Journal of Statistical Software

March 2017, Volume 77, Issue 2. *doi: 10.18637/jss.v077.i02*

Bayesian Network Constraint-Based Structure Learning Algorithms: Parallel and Optimized Implementations in the bnlearn R Package

Marco Scutari
University of Oxford

Abstract

It is well known in the literature that the problem of learning the structure of Bayesian networks is very hard to tackle: Its computational complexity is super-exponential in the number of nodes in the worst case and polynomial in most real-world scenarios.

Efficient implementations of score-based structure learning benefit from past and current research in optimization theory, which can be adapted to the task by using the network score as the objective function to maximize. This is not true for approaches based on conditional independence tests, called constraint-based learning algorithms. The only optimization in widespread use, backtracking, leverages the symmetries implied by the definitions of neighborhood and Markov blanket.

In this paper we illustrate how backtracking is implemented in recent versions of the **bnlearn** R package, and how it degrades the stability of Bayesian network structure learning for little gain in terms of speed. As an alternative, we describe a software architecture and framework that can be used to parallelize constraint-based structure learning algorithms (also implemented in **bnlearn**) and we demonstrate its performance using four reference networks and two real-world data sets from genetics and systems biology. We show that on modern multi-core or multiprocessor hardware parallel implementations are preferable over backtracking, which was developed when single-processor machines were the norm.

Keywords: Bayesian networks, structure learning, parallel programming, R.

1. Background and notations

Bayesian networks (BNs) are a class of *graphical models* (Pearl 1988; Koller and Friedman 2009) composed by a set of random variables $\mathbf{X} = \{X_i, i = 1, \ldots, m\}$ and a *directed acyclic graph* (DAG), denoted $G = (\mathbf{V}, A)$ where $\mathbf{V}$ is the *node set* and A is the *arc set*. The

probability distribution of $\mathbf{X}$ is called the *global distribution* of the data, while those associated with individual X_is are called *local distributions*. Each node in $\mathbf{V}$ is associated with one variable, and they are referred to interchangeably. The directed arcs in A that connect them are denoted as "$\rightarrow$" and represent direct stochastic dependencies; so if there is no arc connecting two nodes, the corresponding variables are either marginally independent or conditionally independent given (a subset of) the rest. As a result, each local distribution depends only on a single node X_i and on its parents (i.e., the nodes $X_j, j \neq i$ such that $X_j \rightarrow X_i$, here denoted $\mathbf{\Pi}_{X_i}$):

$$\mathsf{P}\left(\mathbf{X}\right) = \prod_{i=1}^{m} \mathsf{P}\left(X_i \mid \mathbf{\Pi}_{X_i}\right). \tag{1}$$

Common choices for the local and global distributions are multinomial variables (discrete BNs, Heckerman, Geiger, and Chickering 1995); univariate and multivariate normal variables (Gaussian BNs, Geiger and Heckerman 1994); and, less frequently, a combination of the two (conditional Gaussian (CG) BNs, Lauritzen and Wermuth 1989). In the first case, the parameters of interest are the conditional probabilities associated with each variable, represented as *conditional probability tables* (CPTs); in the second, the parameters of interest are the *partial correlation coefficients* between each variable and its parents. As for CG BNs, the parameters of interest are again partial correlation coefficients, computed for each node conditional on its continuous parents for each configuration of the discrete parents.

The key advantage of the decomposition in Equation 1 is to make *local computations* possible for most tasks, using just a few variables at a time regardless of the magnitude of m. A related quantity that works to the same effect is the *Markov blanket* of each node X_i, defined as the set $\mathcal{B}(X_i)$ of nodes which graphically separates X_i from all other nodes $\mathbf{V} \setminus \{X_i, \mathcal{B}(X_i)\}$ (Pearl 1988, p. 97). In BNs such a set is uniquely identified by the parents ($\mathbf{\Pi}_{X_i}$), the children (i.e., the nodes $X_j, j \neq i$ such that $X_i \rightarrow X_j$) and the spouses of X_i (i.e., the nodes that share a child with X_i). By definition, X_i is independent of all other nodes given $\mathcal{B}(X_i)$, thus making them redundant for inference on X_i.

The task of fitting a BN is called *learning* and is generally implemented in two steps.

The first is called *structure learning*, and consists in finding the DAG that encodes the conditional independencies present in the data. This has been achieved in the literature with *constraint-based*, *score-based* and *hybrid* algorithms; for an overview see Koller and Friedman (2009) and Scutari and Denis (2014). Constraint-based algorithms are based on the seminal work of Pearl on causal graphical models and his inductive causation algorithm (IC, Verma and Pearl 1991), which provides a framework for learning the DAG of a BN using conditional independence tests under the assumption that graphical separation and probabilistic independence imply each other (the *faithfulness* assumption). Tests in common use are the mutual information test (for discrete BNs) and the exact Student's t test for correlation (for GBNs). Score-based algorithms represent an application of heuristic optimization techniques: Each candidate DAG is assigned a network score reflecting its goodness of fit, which the algorithm then attempts to maximize. BIC and posterior probabilities arising from various priors are typical choices. Hybrid algorithms use both conditional independence tests and network scores; the former to reduce the space of candidate DAGs and the latter to identify the optimal DAG among them. Some examples are PC (named after its inventors Peter Spirtes and Clark Glymour; Spirtes, Glymour, and Scheines 2000), grow-shrink (GS; Margaritis 2003), Incremental Association (IAMB; Tsamardinos, Aliferis, and Statnikov 2003), interleaved IAMB

(Inter-IAMB; Yaramakala and Margaritis 2005), hill-climbing and tabu search (Russell and Norvig 2009), max-min parents & children (MMPC; Tsamardinos, Brown, and Aliferis 2006) and the semi-interleaved HITON-PC (SI-HITON-PC, from the Greek "hiton" for "blanket"; Aliferis, Statnikov, Tsamardinos, Mani, and Xenofon 2010). These algorithms and more are implemented across several R packages, such as **bnlearn** (all of the above except PC; Scutari 2010), **deal** (hill-climbing; Bøttcher and Dethlefsen 2003), **catnet** and **mugnet** (simulated annealing; Balov and Salzman 2016; Balov 2013), **pcalg** (PC and causal graphical model learning algorithms; Kalisch, Mächler, Colombo, Maathuis, and Bühlmann 2012) and **abn** (hill climbing and exact algorithms; Lewis 2016). Further extensions to model dynamic data are implemented in **ebdbNet** (Rau, Jaffrézic, Foulley, and Doerge 2010), **G1DBN** (Lèbre 2013) and **ARTIVA** (Lèbre, Becq, Devaux, Lelandais, and Stumpf 2010) among others.

The second step is called *parameter learning* and, as the name suggests, deals with the estimation of the parameters of the global distribution. Since the graph structure is known from the previous step, this can be done efficiently by estimating the parameters of the local distributions. With the exception of **bnlearn**, which has a separate `bn.fit` function, R packages automatically execute this step along with structure learning.

Most problems in BN theory have a computational complexity that, in the worst case, scales exponentially with the number of variables. For instance, both structure learning (Chickering 1996; Chickering, Geiger, and Heckerman 1994) and inference (Cooper 1990; Dagum and Luby 1993) are NP-hard and have polynomial complexity even for sparse networks. This is especially problematic in high-dimensional settings such as genetics and systems biology, in which BNs are used for the analysis of gene expressions (Friedman 2004) and protein-protein interactions (Jansen, Yu, Greenbaum, Kluger, Krogan, Chung, Emili, Snyder, Greenblatt, and Gerstein 2003; Sachs, Perez, Pe'er, Lauffenburger, and Nolan 2005); for integrating heterogeneous genetic data (Chang and McGeachie 2011); and to determine disease susceptibility to anemia (Sebastiani, Ramoni, Nolan, Baldwin, and Steinberg 2005) and hypertension (Malovini, Nuzzo, Ferrazzi, Puca, and Bellazzi 2009).

Even though algorithms in recent literature are designed taking scalability into account, it is often impractical to learn BNs from data containing more than few hundreds of variables without restrictive assumptions on either the structure of the DAG or the nature of the local distributions. Two ways to address this problem are:

1. *Optimizations:* reducing the number of conditional independence tests and network scores computed from the data, either by skipping redundant ones or by limiting local computations to a few variables.

2. *Parallel implementations:* splitting learning across multiple cores and processors to make better use of modern multi-core and multiprocessor hardware.

As far as score-based learning algorithms are concerned, both possibilities have been explored and a wide range of solutions proposed, from efficient caching using decomposable scores (Daly and Shen 2007), to parallel meta-heuristics (Rauber and Rünger 2010) and integer programming (Cussens 2011). The same cannot be said of constraint-based algorithms; even recent ones such as SI-HITON-PC, while efficient, are still implemented with basic backtracking as the only optimization. We will examine them and their implementations in Section 2, arguing that backtracking provides only modest speed gains and increases the variability of the learned DAGs. As an alternative, in Section 3 we describe a software architecture and

framework that can be used to create parallel implementations of constraint-based algorithms that scale well on large data sets and do not suffer from this problem. In both sections we will focus on the **bnlearn** package because it provides the widest choice of algorithms and implementations among those of interest for this paper.

2. Constraint-based structure learning and backtracking

All constraint-based structure learning algorithms share a common three-phase structure inherited from the IC algorithm through PC and GS; it is illustrated in Algorithm 1. The first, optional, phase consists in learning the Markov blanket of each node to reduce the number of candidate DAGs early on. Any algorithm for learning Markov blankets can be plugged in Step 1 and extended into a full BN structure learning algorithm, as originally suggested in Margaritis (2003) for GS. Once all Markov blankets have been learned, they are checked for consistency (Step 2) using their symmetry; by definition $X_i \in \mathcal{B}(X_j) \Leftrightarrow X_j \in \mathcal{B}(X_i)$. Asymmetries are corrected by treating them as false positives and removing the offending nodes from each other's Markov blankets.

The second phase learns the *skeleton* of the DAG, that is, it identifies which arcs are present in the DAG modulo their direction. This is equivalent to learning the *neighbors* $\mathcal{N}(X_i)$ of each node: its parents and children. As illustrated in Step 3, the absence of a set of nodes $\mathbf{S}_{X_iX_j}$ that separates a particular pair X_i, X_j implies that either $X_i \to X_j$ or $X_j \to X_i$. Separating sets are considered in order of increasing size to keep computations as local as possible. Furthermore, if $\mathcal{B}(X_i)$ and $\mathcal{B}(X_j)$ are available from Steps 1 and 2 the search space can be greatly reduced because $\mathcal{N}(X_i) \subseteq \mathcal{B}(X_i)$. On the one hand, if $X_j \notin \mathcal{B}(X_i)$ by definition X_i is separated from X_j by $\mathbf{S}_{X_iX_j} = \mathcal{B}(X_i)$. On the other hand, if $X_j \in \mathcal{B}(X_i)$ most candidate sets can be disregarded because we know that $\mathbf{S}_{X_iX_j} \subseteq \mathcal{B}(X_i) \setminus X_j$ and $\mathbf{S}_{X_iX_j} \subseteq \mathcal{B}(X_j) \setminus X_i$; both sets are typically much smaller than $\mathbf{V}$. With the exception of the PC algorithm, which is structured exactly as described in Step 3, constraint-based algorithms learn the skeleton by learning each $\mathcal{N}(X_i)$ and then enforcing symmetry (Step 4).

Finally, in the third phase arc directions are established as in Meek (1995). It is important to note that, for some arcs, both directions are equivalent in the sense that they identify equivalent decompositions of the global distribution. Therefore, some arcs will be left undirected and the algorithm will return a *completed partially directed acyclic graph* identifying an *equivalence class* containing multiple DAGs. Such a class is uniquely identified by the skeleton learned in Steps 3 and 4, and by the *v-structures* $\mathcal{V}_l$ of the form $X_i \to X_k \leftarrow X_j$, $X_i \notin \mathcal{N}(X_j)$ learned in Step 5 (Chickering 1995). Additional arc directions are inferred indirectly in Step 6 by ruling out those that would introduce additional v-structures (which would have been identified in Step 5) or cycles (which are not allowed in DAGs).

Even in such a general form, we can see that Algorithm 1 performs many checks for graphical separation that are redundant given the symmetry of the $\mathcal{B}(X_i)$ and the $\mathcal{N}(X_i)$. Intuitively, once we have concluded that $X_j \notin \mathcal{B}(X_i)$ there is no need to check whether $X_i \in \mathcal{B}(X_j)$ in Step 1; and similar considerations can be made for neighbors in Step 3. In practice, this translates to redundant independence tests computed on the data. Therefore, up to version 3.4 **bnlearn** implemented *backtracking* by assuming $X_1, \ldots, X_m$ were processed sequentially and enforcing symmetry by construction (e.g., if $i < j$ then $X_j \notin \mathcal{B}(X_i) \Rightarrow X_i \notin \mathcal{B}(X_j)$ and $X_j \in \mathcal{B}(X_i) \Rightarrow X_i \in \mathcal{B}(X_j)$). While this approach on average reduces the number of tests by a

Algorithm 1 A template for constraint-based structure learning algorithms.

Input: a data set containing the variables $X_i, i = 1, \ldots, m$.
Output: a completed partially directed acyclic graph.

Phase 1: Learning Markov blankets (optional).

1. For each variable X_i, learn its Markov blanket $\mathcal{B}(X_i)$.
2. Check whether the Markov blankets $\mathcal{B}(X_i)$ are symmetric, e.g., $X_i \in \mathcal{B}(X_j) \Leftrightarrow X_j \in \mathcal{B}(X_i)$. Assume that nodes for whom symmetry does not hold are false positives and drop them from each other's Markov blankets.

Phase 2: Learning neighbors.

3. For each variable X_i, learn the set $\mathcal{N}(X_i)$ of its neighbors (i.e., the parents and the children of X_i). Equivalently, for each pair $X_i, X_j, i \neq j$ search for a set $\mathbf{S}_{X_iX_j} \subset \mathbf{V}$ (including $\mathbf{S}_{X_iX_j} = \varnothing$) such that X_i and X_j are independent given $\mathbf{S}_{X_iX_j}$ and $X_i, X_j \notin \mathbf{S}_{X_iX_j}$. If there is no such a set, place an undirected arc between X_i and X_j ($X_i - X_j$). If $\mathcal{B}(X_i)$ and $\mathcal{B}(X_j)$ are available from Steps 1 and 2, the search for $\mathbf{S}_{X_iX_j}$ can be limited to the smallest of $\mathcal{B}(X_i) \setminus X_j$ and $\mathcal{B}(X_j) \setminus X_i$.
4. Check whether the $\mathcal{N}(X_i)$ are symmetric, and correct asymmetries as in Step 2.

Phase 3: Learning arc directions.

5. For each pair of non-adjacent variables X_i and X_j with a common neighbor X_k, check whether $X_k \in \mathbf{S}_{X_iX_j}$. If not, set the direction of the arcs $X_i - X_k$ and $X_k - X_j$ to $X_i \rightarrow X_k$ and $X_k \leftarrow X_j$ to obtain a v-structure $\mathcal{V}_l = \{X_i \rightarrow X_k \leftarrow X_j\}$.
6. Set the direction of arcs that are still undirected by applying the following two rules recursively:
   (a) If X_i is adjacent to X_j and there is a strictly directed path from X_i to X_j (a path leading from X_i to X_j containing no undirected arcs) then set the direction of $X_i - X_j$ to $X_i \rightarrow X_j$.
   (b) If X_i and X_j are not adjacent but $X_i \rightarrow X_k$ and $X_k - X_j$, then change the latter to $X_k \rightarrow X_j$.

factor of 2, it also introduces a false positive or a false negative in the learning process for every type I or type II error in the independence tests. As long as BN learning was only feasible for simple data sets (due to limitations in computational power and algorithm efficiency), and the focus was on probabilistic modeling, the overall error rate was still acceptable; but the increasing prevalence of causal modeling on "small n, large p" data sets in many fields requires a better approach. One such is described in Tsamardinos *et al.* (2006, p. 46) and implemented in **bnlearn** from version 3.5. It modifies Steps 1 and 3 as follows:

- If $X_j \notin \mathcal{B}(X_i), i < j$, then do not consider X_i for inclusion in $\mathcal{B}(X_j)$; and if $X_j \notin \mathcal{N}(X_i)$, then do not consider X_i for inclusion in $\mathcal{N}(X_i)$.

- If $X_j \in \mathcal{B}(X_i)$, then consider X_i for inclusion in $\mathcal{B}(X_j)$ by initializing $\mathcal{B}(X_j) = \{X_i\}$; and if $X_j \in \mathcal{N}(X_i)$ then initialize $\mathcal{N}(X_j) = \{X_i\}$. Note that in both cases X_i can be discarded in the process of learning $\mathcal{B}(X_j)$ and $\mathcal{N}(X_j)$.

Even in this form, backtracking has the undesirable effect of making structure learning depend on the order the variables are stored in the data set, which has been shown to increase errors in the PC algorithm (Colombo and Maathuis 2014). In addition, backtracking provides only a modest speed increase compared to a parallel implementation of Steps 1–4; we will compare the respective running times in Section 3. However, it is easy to implement side-by-side with the original versions of constraint-based structure learning algorithms. Such algorithms are typically described only at the node level, that is, they define how each $\mathcal{B}(X_i)$ and $\mathcal{N}(X_i)$ is learned and then they combine them as described in Algorithm 1. **bnlearn** exports two functions that give access to the corresponding backends: `learn.mb` to learn a single $\mathcal{B}(X_i)$ and `learn.nbr` to learn a single $\mathcal{N}(X_i)$. The old approach to backtracking essentially whitelisted all nodes such that $X_j \in \mathcal{B}(X_i)$ and blacklisted all nodes such that $X_j \notin \mathcal{B}(X_i)$ for each X_i.

```
R> library("bnlearn")
R> data("learning.test", package = "bnlearn")
R> learn.nbr(x = learning.test, method = "si.hiton.pc", node = "D",
+    whitelist = c("A", "C"), blacklist = "B")
```

For example, in the code above we learn $\mathcal{N}(\texttt{D})$ and, assuming we already learned $\mathcal{N}(\texttt{A})$, $\mathcal{N}(\texttt{B})$ and $\mathcal{N}(\texttt{C})$, we `whitelist` and `blacklist` `A`, `B` and `C` depending on whether `D` was one of their neighbors or not. The remaining nodes in the BN are neither whitelisted nor blacklisted and are then tested for conditional independence. By contrast, the current approach initializes $\mathcal{N}(\texttt{D})$ as $\{\texttt{A}, \texttt{C}\}$ but does not whitelist those nodes.

```
R> learn.nbr(x = learning.test, method = "si.hiton.pc", node = "D",
+    blacklist = "B", start = c("A", "C"))
```

As a result, both `A` and `C` can be removed from $\mathcal{N}(\texttt{D})$ by the algorithm. The vanilla, non-optimized equivalent for the same learning algorithm would be

```
R> learn.nbr(x = learning.test, method = "si.hiton.pc", node = "D")
```

which does not include any information from $\mathcal{N}(\texttt{A})$, $\mathcal{N}(\texttt{B})$ or $\mathcal{N}(\texttt{C})$. The syntax for `learn.mb` is identical, and will be omitted for brevity. The only other R package implementing general constraint-based structure learning, **pcalg**, implements the PC algorithm as a monolithic function and does not export the backends which are used to learn the presence of an arc between a pair of nodes. Furthermore, as we noted above, PC is implemented differently from other constraint-based algorithms and is usually modified with different optimizations than backtracking; see, for instance, the interleaving described in Colombo and Maathuis (2014).

In the remainder of this section we will focus on the effects of backtracking on learning the skeleton of the DAG, because Steps 1–4 comprise the vast majority of the overall conditional independence tests and thus control most of the variability of the DAG. To investigate it, we used **bnlearn** and 5 reference BNs of various size and complexity from http://www.bnlearn.com/bnrepository:

- ALARM (Beinlich, Suermondt, Chavez, and Cooper 1989), with 37 nodes, 46 arcs and $p = 509$ parameters. A BN designed by medical experts to provide an alarm message system for intensive care unit patients based on the output a number of vital signs monitoring devices.

- HEPAR II (Onisko 2003), with 70 nodes, 123 arcs and $p = 1453$ parameters. A BN model for the diagnosis of liver disorders from related clinical conditions (e.g., gallstones) and relevant biomarkers (e.g., bilirubin, hepatocellular markers).

- ANDES (Conati, Gertner, VanLehn, and Druzdzel 1997), with 223 nodes, 338 arcs and $p = 1157$ parameters. An intelligent tutoring system based on a student model for the field of classical Newtonian physics, developed at the University of Pittsburgh and at the United States Naval Academy. It handles long-term knowledge assessment, plan recognition, and prediction of students' actions during problem solving exercises.

- LINK (Jensen and Kong 1999), with 724 nodes, 1125 arcs and $p = 14211$ parameters. Developed in the context of linkage analysis in large pedigrees, it models the linkage and the distance between a causal gene and a genetic marker.

- MUNIN (Andreassen, Jensen, Andersen, Falck, Kjærulff, Woldbye, Sørensen, Rosenfalck, and Jensen 1989), with 1041 nodes, 1397 arcs with $p = 80592$ parameters. A BN designed by experts to interpret results from electromyographic examinations and diagnose a large set of common diseases from physical conditions such as atrophy and active nerve fibers.

Simulations were performed on a cluster of 7 dual AMD Opteron 6136, each with 16 cores and 78GB of RAM, with R 3.1.0 and **bnlearn** 3.5. For each BN, we considered 6 different sample sizes ($n = 0.1p, 0.2p, 0.5p, p, 2p, 5p$), chosen as multiples of p to facilitate comparisons between networks of such different complexity; and 4 different constraint-based structure learning algorithms (GS, Inter-IAMB, MMPC, SI-HITON-PC). Since all reference BNs are discrete, we used the asymptotic χ^2 mutual information test with $\alpha = 0.01$. For each combination of BN, sample size and algorithm we repeated the following simulation 20 times. First, we loaded the BN from the RDA file downloaded from the repository (`alarm.rda` below) and generated a sample of the appropriate size with `rbn`.

```
R> load("alarm.rda")
R> sim <- rbn(bn, n = round(0.1 * nparams(bn)))
```

From that sample, we learned the skeleton of the DAG with (`optimized = TRUE`) and without backtracking (`optimized = FALSE`).

```
R> skel.orig <- skeleton(si.hiton.pc(sim, test = "mi", alpha = 0.01,
+    optimized = FALSE))
R> skel.back <- skeleton(si.hiton.pc(sim, test = "mi", alpha = 0.01,
+    optimized = TRUE))
```

Subsequently, we reversed the order of the columns in the data to investigate whether this results in a different skeleton.

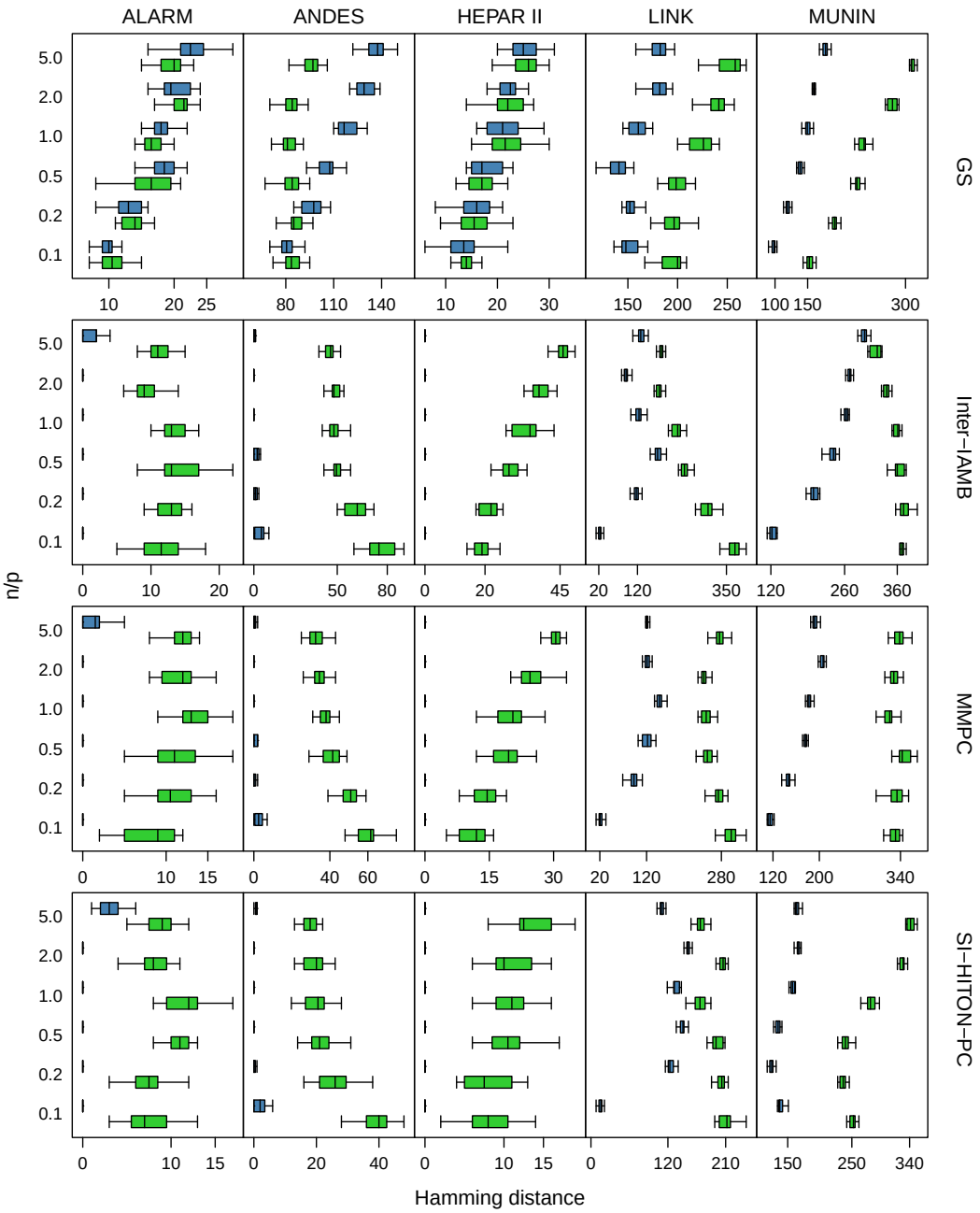

Figure 1: Hamming distance between skeletons learned for the ALARM, ANDES, HEPAR II, LINK and MUNIN reference BNs before and after reversing the ordering of the variables, for various n/p ratios and algorithms. Blue boxplots correspond to structure learning without backtracking, green boxplots to learning with backtracking.

```
R> revsim <- sim[, rev(seq(ncol(sim)))]
```

After learning the skeleton with (`rskel.back`) and without backtracking (`rskel.orig`) from `revsim`, we compared the output with that from `sim` using Hamming distance (Jungnickel 2008).

```
R> rskel.orig <- skeleton(si.hiton.pc(revsim, test = "mi", alpha = 0.01,
+    optimized = FALSE))
R> rskel.back <- skeleton(si.hiton.pc(revsim, test = "mi", alpha = 0.01,
+    optimized = TRUE))
R> hamming(skel.orig, rskel.orig)

[1] 0

R> hamming(skel.back, rskel.back)

[1] 10
```

Ideally, `skel.orig` and `rskel.orig` should be identical and therefore their Hamming distance should be zero. This may not be the case for BNs with deterministic 0-1 nodes, whose structure is unlikely to be learned correctly by any of the considered algorithms; or when conditional independence tests are biased and have low power because of small sample sizes. The difference between the Hamming distance of `skel.orig` and `rskel.orig` and that of `skel.back` and `rskel.back` gives an indication of the dependence on the ordering of the variables introduced by backtracking. It is important to note that different algorithms will also learn the structure of the reference BNs with varying degrees of accuracy, as described in the original papers and in Aliferis *et al.* (2010). However, in this paper we choose to focus on the effect of backtracking (and later of parallelization) as an algorithm-independent optimization technique. Therefore, we compare `skel.orig` with `rskel.orig` and `skel.back` with `rskel.back` instead of comparing all of them to the true skeleton of each reference BN.

The results of this simulation are shown in Figure 1. With the exception of ALARM, ANDES, HEPAR II for the GS algorithm, the Hamming distance between the learned BNs is always greater when backtracking is used. In other words, `hamming(skel.back, rskel.back)` is greater than `hamming(skel.orig, rskel.orig)` for all BNs, algorithms and sample sizes. In fact, Hamming distance does not appear to converge to zero as sample size increases; on the contrary, large samples make even weak dependencies detectable and thus increase the chances of getting different skeletons. This trend is consistently more marked when using backtracking, as is the range of observed Hamming distances in each configuration of BN, sample size and learning algorithm. The combination of these two facts suggests that backtracking does indeed make learning dependent on the order in which the variables are considered; and that it increases the variability of the learned structure.

3. A framework for parallel constraint-based learning

Constraint-based algorithms as described in Algorithm 1 display *coarse-grained parallelism*: different parts need to be synchronized only three times, in Steps 2, 4 and 6. Steps 1, 3 and 5 are *embarrassingly parallel*, because each $\mathcal{B}(X_i)$, each $\mathcal{N}(X_i)$ and each $\mathcal{V}_l$ can be learned independently from the others. In practice, this means changing Step 3 from

```
R> sapply(names(learning.test), function(node) {
+    learn.nbr(learning.test, node = node, method = "si.hiton.pc")
+  })
```

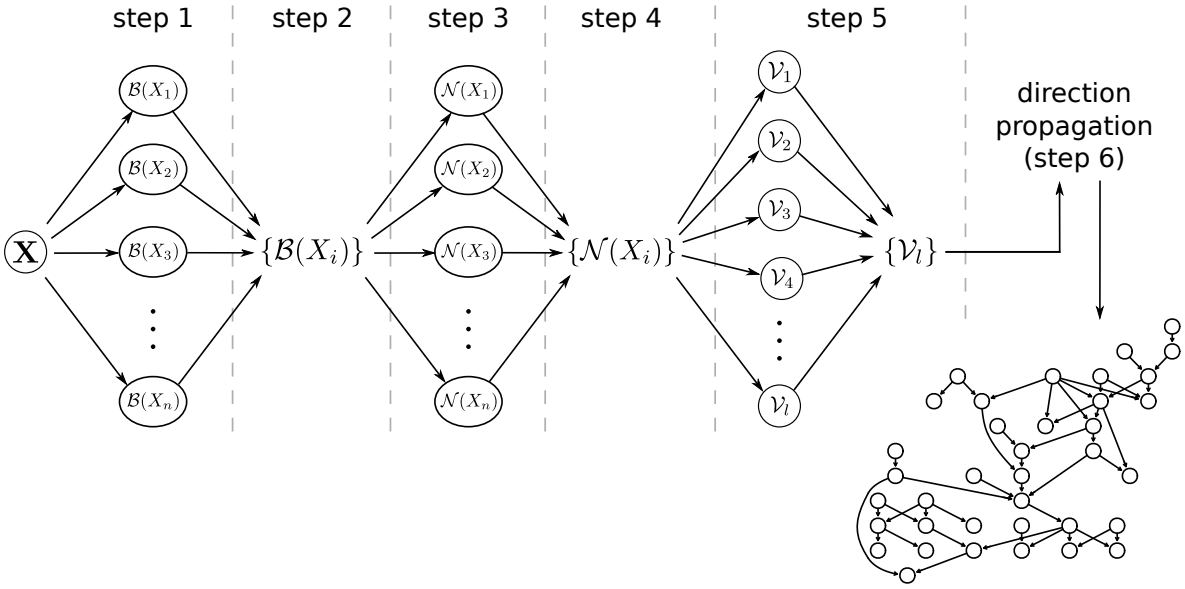

Figure 2: Software architecture for parallel constraint-based structure learning; parallel implementation of Algorithm 1 in **bnlearn**.

to

```
R> library("parallel")
R> cl <- makeCluster(2)
R> clusterEvalQ(cl, library("bnlearn"))
R> parSapply(cl, names(learning.test), function(node) {
+    learn.nbr(learning.test, node = node, method = "si.hiton.pc")
+  })
```

using the functionality provided by the **parallel** package (R Core Team 2016). Step 1 can be modified in the same way for those algorithms that learn the $\mathcal{B}(X_i)$, just calling `learn.mb` instead of `learn.nbr`. Step 6 on the other hand is *inherently sequential* because of its iterative formulation. Parallelizing Algorithm 1 on a step-by-step basis is therefore very convenient. As shown in Figure 2, the implementation still follows the same steps; it performs exactly the same conditional independence tests, thus resulting in the same BN; and it can scale efficiently because computationally intensive steps can be partitioned in as many as parts as there are variables. Splitting the tests in large batches corresponding to the $\mathcal{B}(X_i)$, $\mathcal{N}(X_i)$ and $\mathcal{V}_l$ also reduces the amount of information exchanged by different parts of the implementation, reducing overhead.

Similar changes could in principle be applied to the PC algorithm; different pairs of nodes can be analyzed in parallel and arcs merged into the $\mathcal{N}(X_i)$ at the end of Step 3. However, as was the case for backtracking, the monolithic implementation in **pcalg** would require a complete refactoring beforehand.

3.1. Simulations on the reference BNs

All constraint-based learning algorithms in **bnlearn** have such a parallel implementation made available transparently to the user, who only needs to initialize a '`cluster`' object using **parallel**. The master R process controls the learning process and distributes the $\mathcal{B}(X_i)$, $N(X_i)$ and $\mathcal{V}_l$ to the slave processes, executing only Steps 2, 4 and 6 itself. Consider, for example, the Inter-IAMB algorithm and the data set generated from the ALARM reference BN shipped with **bnlearn**.

```
R> data("alarm", package = "bnlearn")
R> library("parallel")
R> cl <- makeCluster(2)
R> res <- inter.iamb(alarm, cluster = cl)
R> unlist(clusterEvalQ(cl, test.counter()))

[1] 3637 3743

R> stopCluster(cl)
```

After generating a cluster `cl` with 2 slave processes with `makeCluster`, we passed it to `inter.iamb` via the `cluster` argument. As we can see from the output of `clusterEvalQ`, the first slave process performed 3637 (49.3%) conditional tests, and the second 3743 (50.7%). The difference in the number of tests between the two slaves is due to the topology of the BN: the $\mathcal{B}(X_i)$ and $\mathcal{N}(X_i)$ have different sizes and therefore require different numbers of tests to learn. This in turn also affects the number of tests required to learn the v-structures $\mathcal{V}_l$.

Increasing the number of slave processes reduces the number of tests performed by each of them, further increasing the overall performance of the algorithm.

```
R> cl <- makeCluster(3)
R> res <- inter.iamb(alarm, cluster = cl)
R> unlist(clusterEvalQ(cl, test.counter()))

[1] 2218 2479 2683

R> stopCluster(cl)
R> cl <- makeCluster(4)
R> res <- inter.iamb(alarm, cluster = cl)
R> unlist(clusterEvalQ(cl, test.counter()))

[1] 1737 1900 1719 2024

R> stopCluster(cl)
```

The raw and normalized running times of the algorithms used in Section 2 are reported in Figure 3 for clusters of 1, 2, 3, 4, 6 and 8 slaves; values are averaged over 10 runs for each configuration using generated data sets of size 20000. Only the results for LINK and MUNIN are shown, as they are the largest reference BNs considered in this paper. For ALARM, HEPAR II and ANDES, and for smaller sample sizes, running times are too short to make

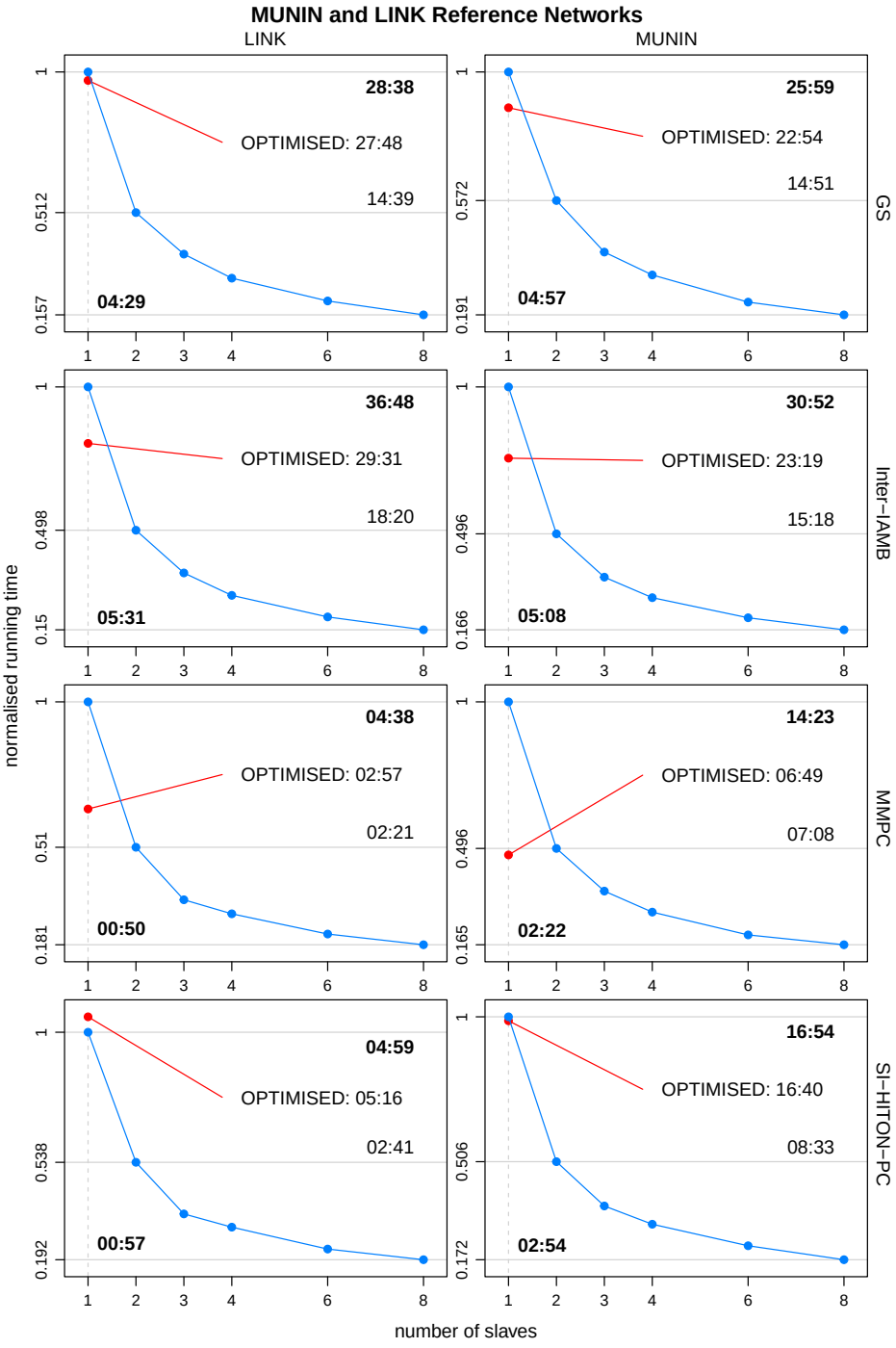

Figure 3: Normalized running times for learning the skeletons of the MUNIN and LINK reference BNs with the GS, Inter-IAMB, MMPC and SI-HITON-PC algorithms. Raw running times are reported for backtracking and for parallel learning with 1, 2 and 8 slave processes.

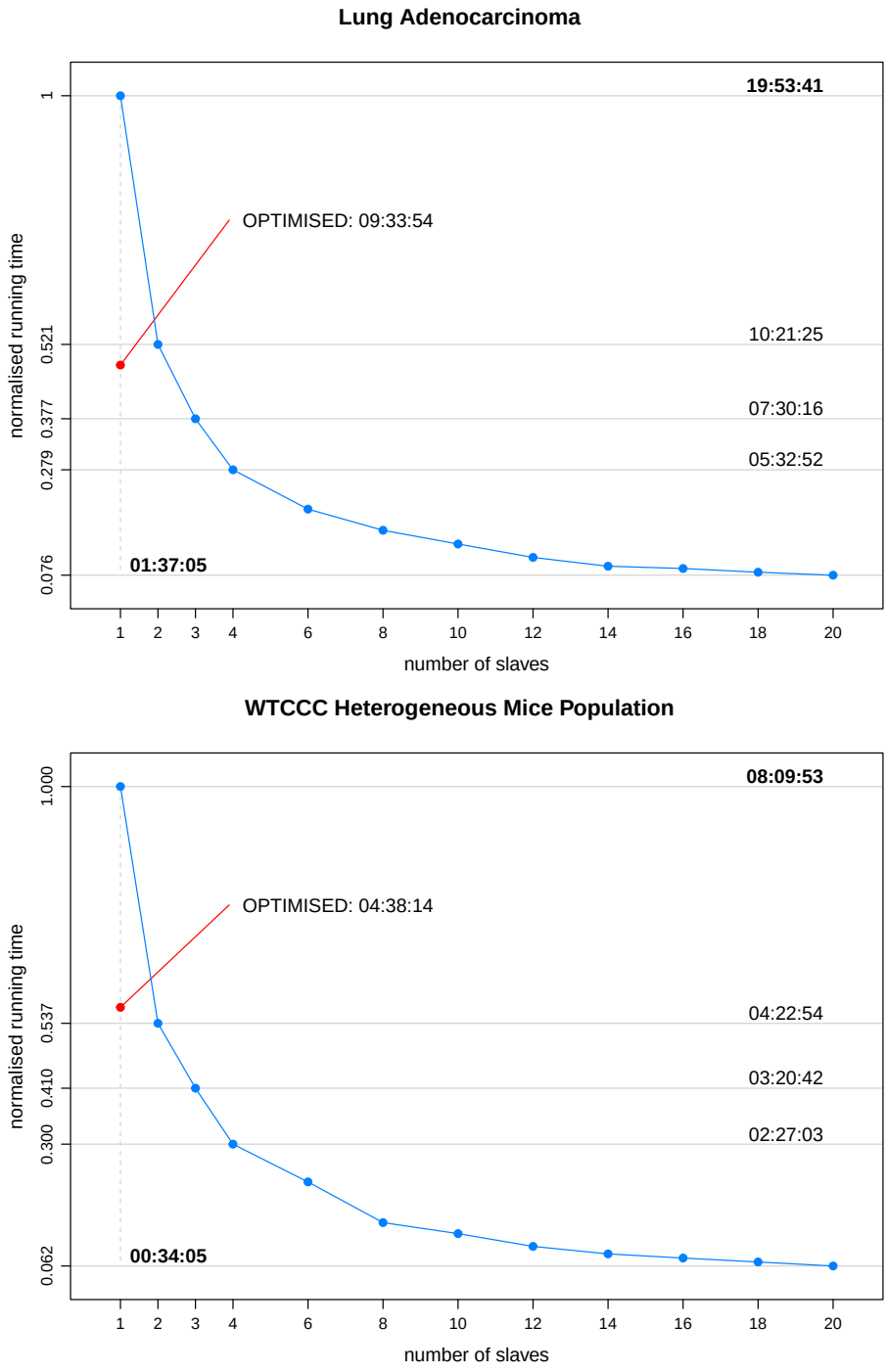

Figure 4: Running times for learning the skeletons underlying the lung adenocarcinoma (Beer *et al.* 2002) and mice (Valdar *et al.* 2006) data sets with SI-HITON-PC. Raw running times are reported for backtracking and for parallel learning with 1, 2, 3, 4 and 20 slave processes.

parallel learning meaningful for at least MMPC and SI-HITON-PC. It is clear from the figure that the gains in running time follow the *law of diminishing returns*: Adding more slaves produces smaller and smaller improvements. Furthermore, tests are never split uniformly among the slave processes and therefore slaves that have fewer tests to perform are left waiting for others to complete (see, for instance, the R code snippets above). Even so, the parallel implementations in **bnlearn** scale efficiently up to 8 slaves. In the absence of any overhead we would expect the average normalized running time to be approximately $1/8 = 0.125$; observed values are in the range $[0.157, 0.191]$. The difference, which is in the range $[0.032, 0.066]$, can be attributed to a combination of communication and synchronization costs as discussed above. Optimized learning is at best competitive with 2 slaves (MMPC, MUNIN), and at worst may actually degrade performance (LINK, SI-HITON-PC).

3.2. Simulations on the real-world data

To provide a more realistic benchmarking on large-scale biological data, we applied SI-HITON-PC on the lung adenocarcinoma gene expression data (86 observations, 7131 variables representing expression levels) from Beer *et al.* (2002); and on the Wellcome Trust Case Control Consortium (WTCCC) heterogeneous mice sequence data (1940 observations, 4053 variables representing allele counts) from Valdar *et al.* (2006). The former is a landmark study in predicting patient survival after an early-stage lung adenocarcinoma diagnosis. Building a gene network from such data to explore interactions and the presence of regulator genes is a common task in systems biology literature, hence the interest in benchmarking BN structure learning. The latter is a reference data set produced by WTCCC to study genome-wide high-resolution mapping of quantitative trait loci using mice as animal models for human diseases. In this context BNs have been used to investigate dependence patterns between single nucleotide polymorphisms (Morota, Valente, Rosa, Weigel, and Gianola 2012).

Both data sets are publicly available and have been preprocessed to remove highly correlated variables ($\mathsf{COR} > 0.95$) and to impute missing values with the **impute** package (Hastie, Tibshirani, Narasimhan, and Chu 2016). The adenocarcinoma data set has a sample size which is extremely small compared to the number of variables, which is common in systems biology. On the other hand, the mice data has a sample size that is typical of large genome-wide association studies. We ran SI-HITON-PC using 1, 2, 3, 4, 6, 8, 10, 12, 14, 16, 18 and 20 slaves, averaging over 5 runs in each case; the other algorithms we considered in Section 2 did not scale well enough to handle either data set. Variables were treated as continuous, and independence was tested using the Student's t test for correlation with $\alpha = 0.01$.

As we can see in Figure 4, we observe a low overhead even for 20 slave processes, with normalized running times of 0.062 (mice) and 0.076 (adenocarcinoma) which are very close to the theoretical $1/20 = 0.05$. Similar considerations can be made across the whole range of 2 to 20 slaves, with a measured overhead between 0.02 and 0.08. Surprisingly, overhead seems to decrease in absolute terms with the number of slaves, from 0.04 (adenocarcinoma) and 0.08 (mice) for 3 slaves to 0.012 (adenocarcinoma) and 0.026 (mice) for 20 slaves. However, clusters with 2 slaves have a smaller overhead (0.021 and 0.037) than those with 3 or 4 slaves. We note that overhead is comparable to that of the reference BNs in Section 2, suggesting that it does not strongly depend on the size of the BN; and that the widely different sample sizes of the two data sets also seem to have little effect. Again, the running time of the optimized implementation is comparable with that of 2 slaves.

4. Discussion and conclusions

In this paper we described a software architecture and framework to create parallel implementations of constraint-based structure learning algorithms for BNs and its implementation in **bnlearn**. Since all these algorithms trace their roots to the IC algorithm from Verma and Pearl (1991), they share a common layout and can be parallelized in the same way. In particular, several steps are embarrassingly parallel and can be trivially split in independent parts to be executed simultaneously. This is important for two reasons. Firstly, it limits the amount of overhead in the parallel computations due to the need of keeping different parts of the algorithms in sync. As we have seen in Section 3, this allows the parallel implementations to scale efficiently in the number of slave processes. Secondly, it implies that the parallel implementation of each algorithm performs the same conditional independence tests as the original. This is in contrast with backtracking, which is the only widespread way of improving the sequential performance of constraint-based algorithms. Different approaches to backtracking have different speed-quality tradeoffs, which motivated the adoption of that currently implemented in **bnlearn**. The simulations in Section 2 suggest that backtracking can increase the variability of the DAGs learned from the data. At the same time, speed gains are competitive at most with the parallel implementation with 2 slave processes. Since most computers in recent times come with at least 2 cores, it is possible to outperform backtracking even on commodity hardware while retaining the lower variability of the non-optimized implementations. Furthermore, even for the largest number of processes considered in this paper (8 for the reference BNs, 20 for the real-world data), the overhead introduced by communication and synchronization between the slaves is low; the highest observed value is 0.08. This suggests that the proposed software architecture as implemented in **bnlearn** and **parallel** scales efficiently for the range of sample sizes and number variables considered in Section 3. Finally, it is important to note that these considerations arise from both discrete and Gaussian BNs and a variety of constraint-based structure learning algorithms.

As for future research, there are several possible ways in which the current implementation may be studied and improved. First of all, overhead might be reduced by replacing `parSapply` with a function that allocates the $\mathcal{B}(X_i)$ and $\mathcal{N}(X_i)$ dynamically to the slaves as they become idle. Assuming the underlying BN is sparse, which is often formalized with a bound on the size of the $\mathcal{B}(X_i)$, this is likely to provide little practical benefit as the overhead is already low compared to the gains provided by parallelism. However, there are some specific settings such as gene regulatory networks (e.g., Babu and Teichmann 2002) in which this assumption is known not to hold; improvements may then be substantial. Such a setup could be based either on the `mcparallel` and `mccollect` functions in the **parallel** package, which unfortunately are not available on Windows, or by avoiding **parallel** entirely to use the **Rmpi** package directly (Yu 2002). Synchronization in Steps 2 and 4 is required to obtain a consistent BN and thus precludes the use of partial update techniques such as that described in Ahmed, Aly, Gonzalez, Narayanamurthy, and Smola (2012).

It would also be interesting to consider how the overhead scales in the sample size and in the complexity of the BN. On average, the number of conditional independence tests required by constraint-based algorithms scales quadratically in the number of variables; and the tests themselves are typically linear in complexity in the sample size. Increasing or decreasing the latter should have little impact on the overhead of parallel learning, because data need to be copied only once to the slaves and that copy could be avoided altogether by using a

shared-memory architecture. The results in Section 3.2 suggest this is indeed the case, and the worst-case overhead is also similar to that of reference BNs in Section 3.1. No locking or synchronization is needed since the data are never modified by the algorithms. On the other hand, the number of variables in the BN can affect overhead in various ways. If the BN is small, differences in the learning times of the $\mathcal{B}(X_i)$ and $\mathcal{N}(X_i)$ are more likely to leave slave processes idle even with the dynamic allocation scheme described above. However, if the BN is large the size of the $\mathcal{B}(X_i)$ and $\mathcal{N}(X_i)$ may vary dramatically thus introducing significant overhead. In both cases the number of variables can only be used a rough proxy for the complexity of the BN, which depends mainly on its topology; and imposing sparsity assumptions on the structure of the BN can be used as a tool to control overhead by keeping the $\mathcal{B}(X_i)$ and $\mathcal{N}(X_i)$ small and of comparable size.

References

Ahmed A, Aly M, Gonzalez J, Narayanamurthy SM, Smola AJ (2012). "Scalable Inference in Latent Variable Models." In *Proceeddings of the 5th ACM International Conference on Web Search and Data Mining*, pp. 123–132. ACM.

Aliferis CF, Statnikov A, Tsamardinos I, Mani S, Xenofon XD (2010). "Local Causal and Markov Blanket Induction for Causal Discovery and Feature Selection for Classification Part I: Algorithms and Empirical Evaluation." *Journal of Machine Learning Research*, **11**, 171–234. doi:10.1145/956750.956838.

Andreassen S, Jensen FV, Andersen SK, Falck B, Kjærulff U, Woldbye M, Sørensen AR, Rosenfalck A, Jensen F (1989). "MUNIN – An Expert EMG Assistant." In JE Desmedt (ed.), *Computer-Aided Electromyography and Expert Systems*, Clinical Neurophysiology Updates, pp. 255–277. Elsevier Science Ltd.

Babu MM, Teichmann SA (2002). "Evolution of Transcription Factors and the Gene Regulatory Network in Escherichia Coli." *Nucleic Acids Research*, **31**(4), 1234–1244.

Balov N (2013). ***mugnet**: Mixture of Gaussian Bayesian Network Model*. R package version 1.01.3, URL https://CRAN.R-project.org/package=mugnet.

Balov N, Salzman P (2016). ***catnet**: Categorical Bayesian Network Inference*. R package version 1.15, URL https://CRAN.R-project.org/package=catnet.

Beer DG, Kardia SLR, Huang CC, Giordano TJ, Levin AM, Misek DE, Lin L, Chen G, Gharib TG, Thomas DG, Lizyness ML, Kuick R, Hayasaka S, Taylor JMG, Iannettoni MD, Orringer MB, Hanash S (2002). "Gene-Expression Profiles Predict Survival of Patients with Lung Adenocarcinoma." *Nature Medicine*, **8**, 816–824. doi:10.1038/sj.onc.1206529.

Beinlich IA, Suermondt HJ, Chavez RM, Cooper GF (1989). "The ALARM Monitoring System: A Case Study with Two Probabilistic Inference Techniques for Belief Networks." In J Hunter, J Cookson, J Wyatt (eds.), *Proceedings of the 2nd European Conference on Artificial Intelligence in Medicine (AIME)*, pp. 247–256. Springer-Verlag.

Bøttcher SG, Dethlefsen C (2003). "**deal**: A Package for Learning Bayesian Networks." *Journal of Statistical Software*, **8**(20), 1–40. doi:10.18637/jss.v008.i20.

Chang HH, McGeachie M (2011). "Phenotype Prediction by Integrative Network Analysis of SNP and Gene Expression Microarrays." In *Proceedings of the 33rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 6849–6852. IEEE Press, New York.

Chickering DM (1995). "A Transformational Characterization of Equivalent Bayesian Network Structures." In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI95)*, pp. 87–98.

Chickering DM (1996). "Learning Bayesian Networks is NP-Complete." In D Fisher, H Lenz (eds.), *Learning from Data: Artificial Intelligence and Statistics V*, pp. 121–130. Springer-Verlag.

Chickering DM, Geiger D, Heckerman D (1994). "Learning Bayesian Networks is NP-Hard." *Technical report*, Microsoft Research, Redmond. Available as Technical Report MSR-TR-94-17.

Colombo D, Maathuis MH (2014). "Order-Independent Constraint-Based Causal Structure Learning." *Journal of Machine Learning Research*, **15**, 3741–3782. doi:10.1007/978-3-319-11433-0_32.

Conati C, Gertner AS, VanLehn K, Druzdzel MJ (1997). "On-Line Student Modeling for Coached Problem Solving Using Bayesian Networks." In A Jameson, C Paris, C Tasso (eds.), *User-Modeling – Proceedings of the Sixth International Conference UM97 Chia Laguna, Sardinia, Italy June 2–5 1997*, pp. 231–242. Springer-Verlag. doi:10.1007/978-3-7091-2670-7_24.

Cooper GF (1990). "The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks." *Artificial Infelligence*, **42**(2–3), 393–405. doi:10.1016/0004-3702(90)90060-d.

Cussens J (2011). "Bayesian Network Learning with Cutting Planes." In FG Cozman, A Pfeffer (eds.), *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI 2011)*, pp. 153–160. AUAI Press.

Dagum P, Luby M (1993). "Approximating Probabilistic Inference in Bayesian Belief Networks is NP-Hard." *Artificial Intelligence*, **60**(1), 141–153. doi:10.1016/0004-3702(93)90036-b.

Daly R, Shen Q (2007). "Methods to Accelerate the Learning of Bayesian Network Structures." In *Proceedings of the 2007 UK Workshop on Computational Intelligence*. Imperial College, London.

Friedman N (2004). "Inferring Cellular Networks Using Probabilistic Graphical Models." *Science*, **303**(5659), 799–805. doi:10.1126/science.1094068.

Geiger D, Heckerman D (1994). "Learning Gaussian Networks." *Technical report*, Microsoft Research, Redmond. Available as Technical Report MSR-TR-94-10.

Hastie T, Tibshirani R, Narasimhan B, Chu G (2016). ***impute**: Imputation for Microarray Data*. R package version 1.48.0, URL http://Bioconductor.org/packages/release/bioc/html/impute.html.

Heckerman D, Geiger D, Chickering DM (1995). "Learning Bayesian Networks: The Combination of Knowledge and Statistical Data." *Machine Learning*, **20**(3), 197–243. doi:10.1023/a:1007469629108. Available as Technical Report MSR-TR-94-09.

Jansen R, Yu H, Greenbaum D, Kluger Y, Krogan NJ, Chung S, Emili A, Snyder M, Greenblatt JF, Gerstein M (2003). "A Bayesian Networks Approach for Predicting Protein-Protein Interactions from Genomic Data." *Science*, **302**(5644), 449–453. doi:10.1126/science.1087361.

Jensen CS, Kong A (1999). "Blocking Gibbs Sampling for Linkage Analysis in Large Pedigrees with Many Loops." *The American Journal of Human Genetics*, **65**(3), 885–901. doi:10.1086/302524.

Jungnickel D (2008). *Graphs, Networks and Algorithms*. 3rd edition. Springer-Verlag.

Kalisch M, Mächler M, Colombo D, Maathuis MH, 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.

Koller D, Friedman N (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Lauritzen SL, Wermuth N (1989). "Graphical Models for Associations between Variables, Some of Which Are Qualitative and Some Quantitative." *The Annals of Statistics*, **17**(1), 31–57. doi:10.1214/aos/1176347003.

Lèbre S (2013). ***G1DBN**: A Package Performing Dynamic Bayesian Network Inference*. R package version 3.1.1, URL https://CRAN.R-project.org/package=G1DBN.

Lèbre S, Becq J, Devaux F, Lelandais G, Stumpf MPH (2010). "Statistical Inference of the Time-Varying Structure of Gene-Regulation Networks." *BMC Systems Biology*, **4**(130), 1–16. doi:10.4137/grsb.s4509.

Lewis F (2016). ***abn**: Data Modelling with Additive Bayesian Networks*. R package version 1.0, URL https://CRAN.R-project.org/package=abn.

Malovini A, Nuzzo A, Ferrazzi F, Puca A, Bellazzi R (2009). "Phenotype Forecasting with SNPs Data Through Gene-Based Bayesian Networks." *BMC Bioinformatics*, **10**(Suppl 2), S7. doi:10.1186/1471-2105-10-s2-s7.

Margaritis D (2003). *Learning Bayesian Network Model Structure from Data*. Ph.D. thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh. Available as Technical Report CMU-CS-03-153.

Meek C (1995). "Causal Inference and Causal Explanation with Background Knowledge." In P Besnard, S Hanks (eds.), *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI 2011)*, pp. 403–410. Morgan Kaufmann.

Morota G, Valente BD, Rosa GJM, Weigel KA, Gianola D (2012). "An Assessment of Linkage Disequilibrium in Holstein Cattle Using a Bayesian Network." *Journal of Animal Breeding and Genetics*, **129**(6), 474–487. doi:10.1111/jbg.12002.

Onisko A (2003). *Probabilistic Causal Models in Medicine: Application to Diagnosis of Liver Disorders*. Ph.D. thesis, Institute of Biocybernetics and Biomedical Engineering, Polish Academy of Science, Warsaw.

Pearl J (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Rau A, Jaffrézic F, Foulley JL, Doerge R (2010). "An Empirical Bayesian Method for Estimating Biological Networks from Temporal Microarray Data." *Statistical Applications in Genetics and Molecular Biology*, **9**(1). doi:10.1007/978-1-60761-580-4_10.

Rauber T, Rünger G (2010). *Parallel Programming For Multicore and Cluster Systems*. Springer-Verlag. doi:10.1007/978-3-642-04818-0.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Russell SJ, Norvig P (2009). *Artificial Intelligence: A Modern Approach*. 3rd edition. Prentice Hall.

Sachs K, Perez O, Pe'er D, Lauffenburger DA, Nolan GP (2005). "Causal Protein-Signaling Networks Derived from Multiparameter Single-Cell Data." *Science*, **308**(5721), 523–529. doi:10.1126/science.1105809.

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, Denis JB (2014). *Bayesian Networks with Examples in R*. Chapman & Hall.

Sebastiani P, Ramoni MF, Nolan V, Baldwin CT, Steinberg M (2005). "Genetic Dissection and Prognostic Modeling of Overt Stroke in Sickle Cell Anemia." *Nature Genetics*, **37**(4), 435–440. doi:10.1038/ng1533.

Spirtes P, Glymour C, Scheines R (2000). *Causation, Prediction, and Search*. MIT Press. doi:10.1007/978-1-4612-2748-9.

Tsamardinos I, Aliferis CF, Statnikov A (2003). "Algorithms for Large Scale Markov Blanket Discovery." In I Russell, SM Haller (eds.), *Proceedings of the 16th International Florida Artificial Intelligence Research Society Conference*, pp. 376–381. AAAI Press.

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.

Valdar W, Solberg LC, Gauguier D, Burnett S, Klenerman P, Cookson WO, Taylor MS, Rawlins JN, Mott R, Flint J (2006). "Genome-Wide Genetic Association of Complex Traits in Heterogeneous Stock Mice." *Nature Genetics*, **8**, 879–887. doi:10.1038/ng1840.

Verma TS, Pearl J (1991). "Equivalence and Synthesis of Causal Models." *Uncertainty in Artificial Intelligence*, **6**, 255–268. doi:10.1016/0004-3702(87)90012-9.

Yaramakala S, Margaritis D (2005). "Speculative Markov Blanket Discovery for Optimal Feature Selection." In *ICDM '05: Proceedings of the Fifth IEEE International Conference on Data Mining*, pp. 809–812. IEEE Computer Society.

Yu H (2002). "**Rmpi**: Parallel Statistical Computing in R." *R News*, **2**(2), 10–14. doi:10.1016/s0167-8191(02)00175-8.

Affiliation:

Marco Scutari
Department of Statistics
University of Oxford
1 South Parks Road
OX1 3TG Oxford, United Kingdom
E-mail: scutari@stats.ox.ac.uk
URL: http://www.bnlearn.com/about/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
March 2017, Volume 77, Issue 2 *Submitted:* 2014-06-30
doi:10.18637/jss.v077.i02 *Accepted:* 2015-10-02