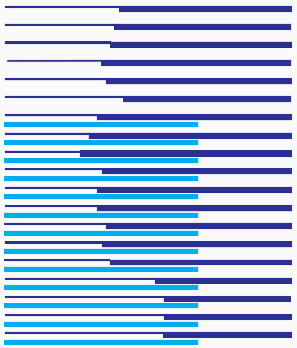


Advanced Probabilistic Modelling

Structure Learning

Marco Scutari

Dalle Molle Institute for
Artificial Intelligence (IDSIA)

Academic Year 2025–2026

Learning a Bayesian Networks

Model selection and estimation are collectively known as learning, and are usually performed as a two-step process:

1. structure learning, learning the graph structure from the data;
2. parameter learning, learning the local distributions implied by the graph structure learned in the previous step.

This workflow is implicitly Bayesian: given a data set $\mathcal{D}$ we have

$$\underbrace{\mathrm{P}(\mathcal{M} \mid \mathcal{D})}_{\text{learning}} = \underbrace{\mathrm{P}(\mathcal{G} \mid \mathcal{D})}_{\text{structure learning}} \cdot \underbrace{\mathrm{P}(\Theta \mid \mathcal{G}, \mathcal{D})}_{\text{parameter learning}}$$

and structure learning is done in practice as

$$\mathrm{P}(\mathcal{G} \mid \mathcal{D}) \propto \mathrm{P}(\mathcal{G})\,\mathrm{P}(\mathcal{D} \mid \mathcal{G}) = \mathrm{P}(\mathcal{G}) \int \mathrm{P}(\mathcal{D} \mid \mathcal{G}, \Theta)\,\mathrm{P}(\Theta \mid \mathcal{G}) d\Theta.$$

Local Distributions: Divide and Conquer

Both are computationally hard, but they are still feasible thanks to the decomposition of $\mathbf{X}$ into local distributions. Under some assumptions, we can use local computations and we never need to manipulate more than one at a time.

Structure learning boils down to

$$\begin{aligned}\mathrm{P}(\mathcal{D} \mid \mathcal{G}) &= \int \prod_{i=1}^{N} \left[\mathrm{P}(X_i \mid \Pi_{X_i}, \Theta_{X_i})\, \mathrm{P}(\Theta_{X_i} \mid \Pi_{X_i})\right] d\Theta \\ &= \prod_{i=1}^{N} \left[\int \mathrm{P}(X_i \mid \Pi_{X_i}, \Theta_{X_i})\, \mathrm{P}(\Theta_{X_i} \mid \Pi_{X_i}) d\Theta_{X_i}\right]\end{aligned}$$

and parameter learning boils down to

$$\mathrm{P}(\Theta \mid \mathcal{G}, \mathcal{D}) = \prod_{i=1}^{N} \mathrm{P}(\Theta_{X_i} \mid \Pi_{X_i}, \mathcal{D}).$$

Prior Elicitation versus Data

For both parameter and structure learning, we can rely either on

- eliciting information from experts, drawing on the available prior knowledge on the variables in $\mathbf{X}$;
- using available data and extract the information the contain.

In structure learning, elicitation involves favouring or penalising the inclusion of specific (patterns of) arcs in the DAG. In parameter learning, it means partially or completely specifying the parameters of local distribution, or constraining them in various ways.

There are pros and cons to either approach:

- it maybe difficult to find experts, or it may be difficult to find data, depending on the phenomenon;
- the data may be noisy or may not fit distributional assumptions;
- it is usually difficult for experts to suggest values for the parameters;
- data may be affected by sampling bias, experts may be affected by personal biases.

Assumptions for Structure Learning from Data

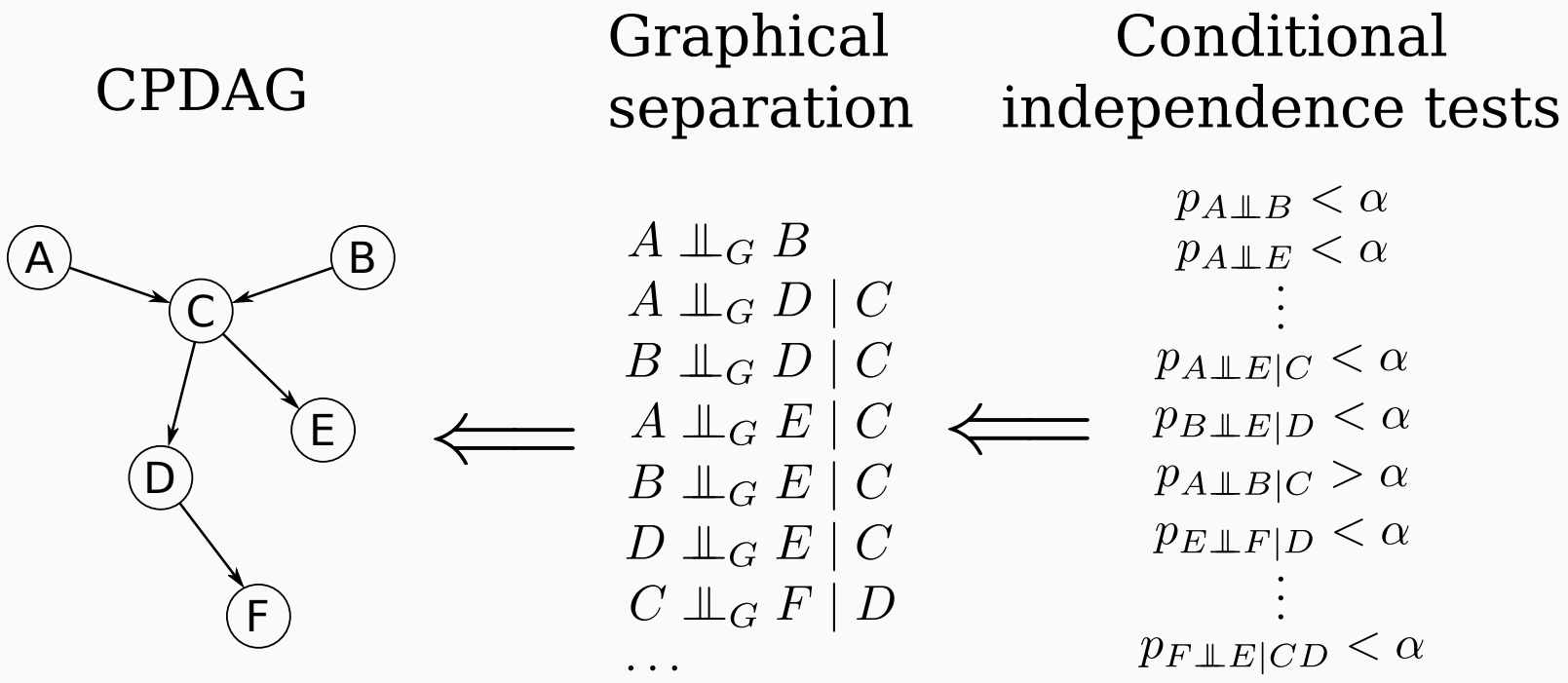
- There must be a one-to-one correspondence between the nodes in the DAG and the random variables in $\mathbf{X}$. There must not be multiple nodes which are deterministic functions of a single variable.
- All the relationships between the variables in $\mathbf{X}$ must be conditional independencies, because they are by definition the only kind of relationships that can be expressed by a BN.
- Every combination of the possible values of the variables in $\mathbf{X}$ must represent a valid, observable (even if really unlikely) event. This assumption implies a strictly positive global distribution, which is needed to a uniquely identifiable model.
- Observations are treated as independent realisations of the set of nodes. If some form of temporal or spatial dependence is present, it must be specifically accounted for in the definition of the network, as in dynamic BNs.

Classes of Structure Learning Algorithms from Data

Despite the (sometimes confusing) variety of theoretical backgrounds and terminology, structure learning algorithms can all be traced to only three approaches:

- Constraint-based algorithms: they use statistical tests to learn conditional independence relationships (called "constraints" in this setting) from the data and assume that the DAG is a perfect map to determine the correct network structure.
- Score-based algorithms: they score each DAG for its goodness of fit, and then find the DAG that maximises that score.
- Hybrid algorithms: conditional independence tests are used to learn at least part of the conditional independence relationships from the data, thus restricting the search space for a subsequent score-based search. The latter determines which edges are actually present in the graph and their direction.

Constraint-Based Structure Learning Algorithms



One way to learn the structure of a BN is to check which conditional independence constraints hold using a suitable conditional independence test. We can identify a single equivalence class in this way.

Assuming a Perfect Map

BNs are defined from graphical separation:

$$\mathbf{A} \perp\!\!\!\perp_G \mathbf{B} \,|\, \mathbf{C} \implies \mathbf{A} \perp\!\!\!\perp_P \mathbf{B} \,|\, \mathbf{C}.$$

However, constraint-based algorithms also imply the reverse:

$$\mathbf{A} \perp\!\!\!\perp_P \mathbf{B} \,|\, \mathbf{C} \iff \mathbf{A} \perp\!\!\!\perp_G \mathbf{B} \,|\, \mathbf{C}.$$

This is a much stronger assumption, which has pros and cons:

- it is impossible to verify;
- but it is a sufficient assumption to uniquely identify Markov blankets, and thus we no longer need to assume that $\mathrm{P}(\mathbf{X})$ is strictly positive everywhere;
- not all $\mathrm{P}(\mathbf{X})$ have a faithful DAG.

The Original: Inductive Causation Algorithm

1. For each pair of variables A and B in $\mathbf{X}$ search for set $\mathbf{S}_{AB} \subset \mathbf{X}$ such that A and B are independent given $\mathbf{S}_{AB}$ and $A, B \notin \mathbf{S}_{AB}$. If there is no such a set, place an undirected arc between A and B.
2. For each pair of non-adjacent variables A and B with a common neighbour C, check whether $C \in \mathbf{S}_{AB}$. If this is not true, set the direction of the arcs $A - C$ and $C - B$ to $A \rightarrow C$ and $C \leftarrow B$.
3. Set the direction of arcs which are still undirected by applying recursively the following two rules:
   - 3.1 if A is adjacent to B and there is a strictly directed path from A to B then set the direction of $A - B$ to $A \rightarrow B$;
   - 3.2 if A and B are not adjacent but $A \rightarrow C$ and $C - B$, then change the latter to $C \rightarrow B$.
4. Return the resulting (partially) directed acyclic graph.

Many newer algorithms: PC, Grow-Shrink, IAMB variants, HITON-PC, HPC.

Conditional Independence Tests: DBNs

Conditional independence tests for $X \perp\!\!\!\perp_P Y | \mathbf{Z}$ are functions of the observed frequencies $\{n_{ijk}, i = 1, \dots, R; j = 1, \dots, C; k = 1, \dots, L\}$.

Classic choices are:

- mutual information/log-likelihood ratio

$$\mathrm{MI}(X, Y | \mathbf{Z}) = \sum_{i=1}^{R} \sum_{j=1}^{C} \sum_{k=1}^{L} \frac{n_{ijk}}{n} \log \frac{n_{ijk} n_{++k}}{n_{i+k} n_{+jk}};$$

- and Pearson's X^2 with a χ^2 distribution

$$\mathrm{X}^2(X, Y | \mathbf{Z}) = \sum_{i=1}^{R} \sum_{j=1}^{C} \sum_{k=1}^{L} \frac{\left(n_{ijk} - m_{ijk}\right)^2}{m_{ijk}}, \quad m_{ijk} = \frac{n_{i+k} n_{+jk}}{n_{++k}}.$$

Both have an asymptotic $\chi^2_{(R-1)(C-1)L}$ null distribution.

Conditional independence tests are functions of the partial correlations $\rho_{XY|\,\mathbf{Z}}$ computed from $\Omega = \Sigma^{-1}_{\{X,Y,\mathbf{Z}\}}$.

Classic choices are:

- the exact t test for Pearson's correlation coefficient, defined as

$$\mathrm{t}(X, Y|\,\mathbf{Z}) = \rho_{XY|\,\mathbf{Z}} \sqrt{\frac{n - |\mathbf{Z}| - 2}{1 - \rho^2_{XY|\,\mathbf{Z}}}} \sim t_{n-|\mathbf{Z}|-2};$$

- the asymptotic Fisher's Z test, defined as

$$\mathrm{Z}(X, Y|\,\mathbf{Z}) = \log\left(\frac{1 + \rho_{XY|\,\mathbf{Z}}}{1 - \rho_{XY|\,\mathbf{Z}}}\right) \frac{\sqrt{n - |\mathbf{Z}| - 3}}{2} \sim N(0, 1)$$

  where n is the number of observations and $|\mathbf{Z}|$ is the number of variables in $\mathbf{Z}$.

Conditional Independence Tests: CGBNs (I)

It is more complicated to specify tests for CLGBNs. Going case by case:

- if X, Y and $\mathbf{Z}$ are all categorical, we can use any test for DBNs;
- if X, Y and $\mathbf{Z}$ are all Gaussian, we can use any test for GBNs;
- if X is categorical and Y is Gaussian (or vice versa), the simple test to use is the mutual information

$$\propto \log \frac{\mathrm{P}(Y|\, X, \mathbf{Z})}{\mathrm{P}(Y|\, \mathbf{Z})}$$

  in which both the numerator and the nominator are linear regressions;
- the same is true if X and Y are Gaussian, regardless of $\mathbf{Z}$ the simple test is again the mutual information.

Conditional Independence Tests: Conditional Gaussian (II)

- if X and Y are categorical, and $\mathbf{Z} = \{Z_{c_1}, \dots, Z_{c_l}, Z_{d_1}, \dots, Z_{d_m}\}$ contains both categorical and Gaussian variables, with several applications of Bayes theorem and the chain rule we get

$$\frac{\mathrm{P}(X \mid Z_{d_1:d_m}, Z_{c_1:c_l})}{\mathrm{P}(X \mid Y, Z_{d_1:d_m}, Z_{c_1:c_l})} = \\ = \frac{\prod_{i=1}^{l-1} \mathrm{P}(Z_{c_i} \mid Z_{c_{i+1}:c_l}, X, Z_{d_1:d_m}) \, \mathrm{P}(X, Z_{d_1:d_m})}{\prod_{i=1}^{l-1} \mathrm{P}(Z_{c_i} \mid Z_{c_{i+1}:c_l}, Z_{d_1:d_m}) \, \mathrm{P}(Z_{d_1:d_m})} \times \\ \frac{\prod_{i=1}^{l-1} \mathrm{P}(Z_{c_i} \mid Z_{c_{i+1}:c_l}, X, Y, Z_{d_1:d_m}) \, \mathrm{P}(X, Y, Z_{d_1:d_m})}{\prod_{i=1}^{l-1} \mathrm{P}(Z_{c_i} \mid Z_{c_{i+1}:c_l}, Y, Z_{d_1:d_m}) \, \mathrm{P}(Y, Z_{d_1:d_m})}$$

which is a chain of log-likelihood ratios that can be treated as a mutual information test.

The ASIA Example, Revisited

The `asia` data set is a small synthetic data set from Lauritzen and Spiegelhalter that tries to implement a diagnostic model for lung diseases (tuberculosis, lung cancer or bronchitis) after a visit to Asia.

- `D`: dyspnoea.
- `T`: tuberculosis.
- `L`: lung cancer.
- `B`: bronchitis.
- `A`: visit to Asia.
- `S`: smoking.
- `X`: chest X-ray.
- `E`: tuberculosis versus lung cancer/bronchitis.

```
head(asia)
```

```
   A   S   T  L   B   E   X   D
1 no yes  no no yes  no  no yes
2 no yes  no no  no  no  no  no
3 no  no yes no  no yes yes yes
4 no  no  no no yes  no  no yes
5 no  no  no no  no  no  no yes
6 no yes  no no  no  no  no yes
```

bnlearn: Functions for Constraint-Based Learning

bnlearn implements several constraint-based algorithms, each with its own function: `gs()`, `iamb()`, `mmpc()`, `si.hiton.pc()`, etc.

```
cpdag = si.hiton.pc(asia, undirected = FALSE)
cpdag
```

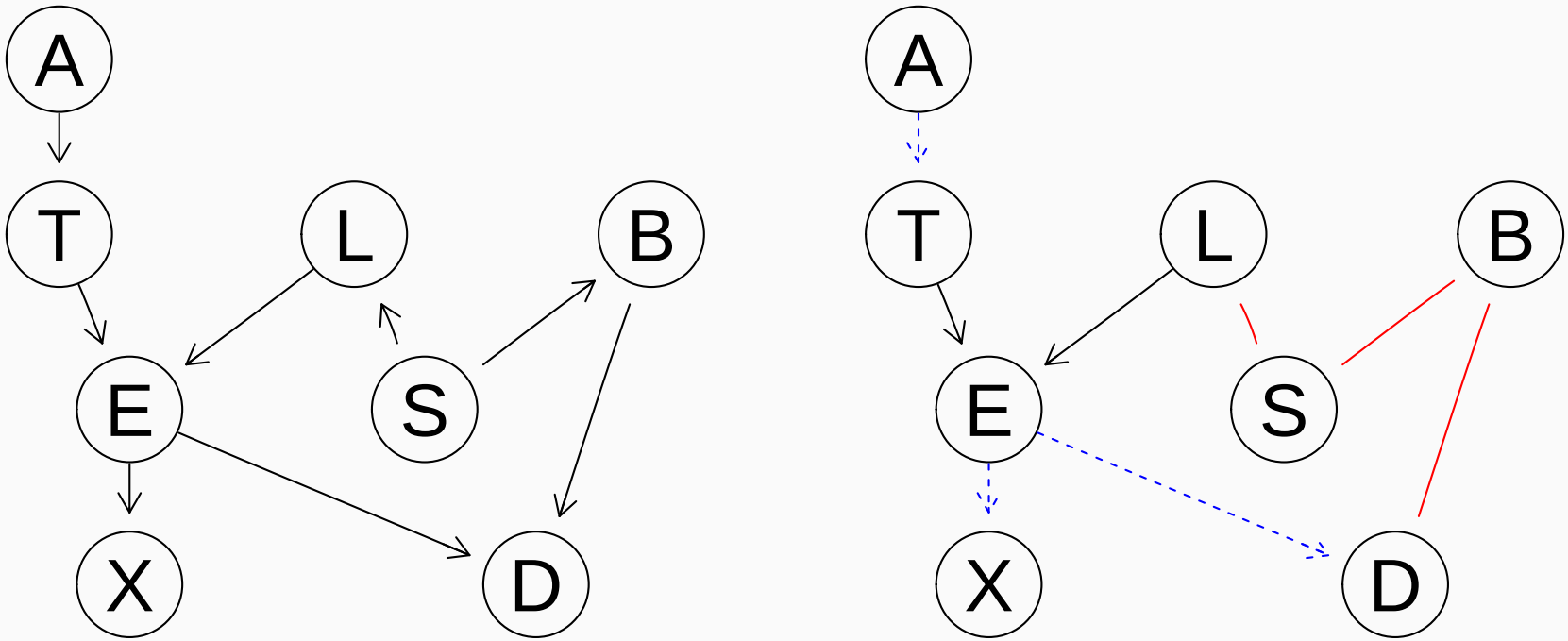
```
  Bayesian network learned via Constraint-based methods

  model:
    [partially directed graph]
  nodes:                                 8
  arcs:                                  5
    undirected arcs:                     3
    directed arcs:                       2
  average markov blanket size:           1.50
  average neighbourhood size:            1.25
  average branching factor:              0.25

  learning algorithm:
                                  Semi-Interleaved HITON-PC
  conditional independence test:
                                 Mutual Information (disc.)
  alpha threshold:                       0.05
  tests used in the learning procedure:  152
```

The arguments for the tuning parameters of constraint-based learning algorithms have the same names in the respective functions:

- the first argument is the data.
- `cluster`: a cluster object from the **parallel** package to perform steps in parallel for different nodes.
- `test`: the label of the test statistic.
- `alpha`: the type-I error threshold for the individual conditional independence tests (i.e. without any multiplicity adjustment).
- `skeleton`: whether to learn just the skeleton instead of the CPDAG.
- `debug`: whether to print out the steps performed by the algorithm.

BNLEARN: Comparing DAGs

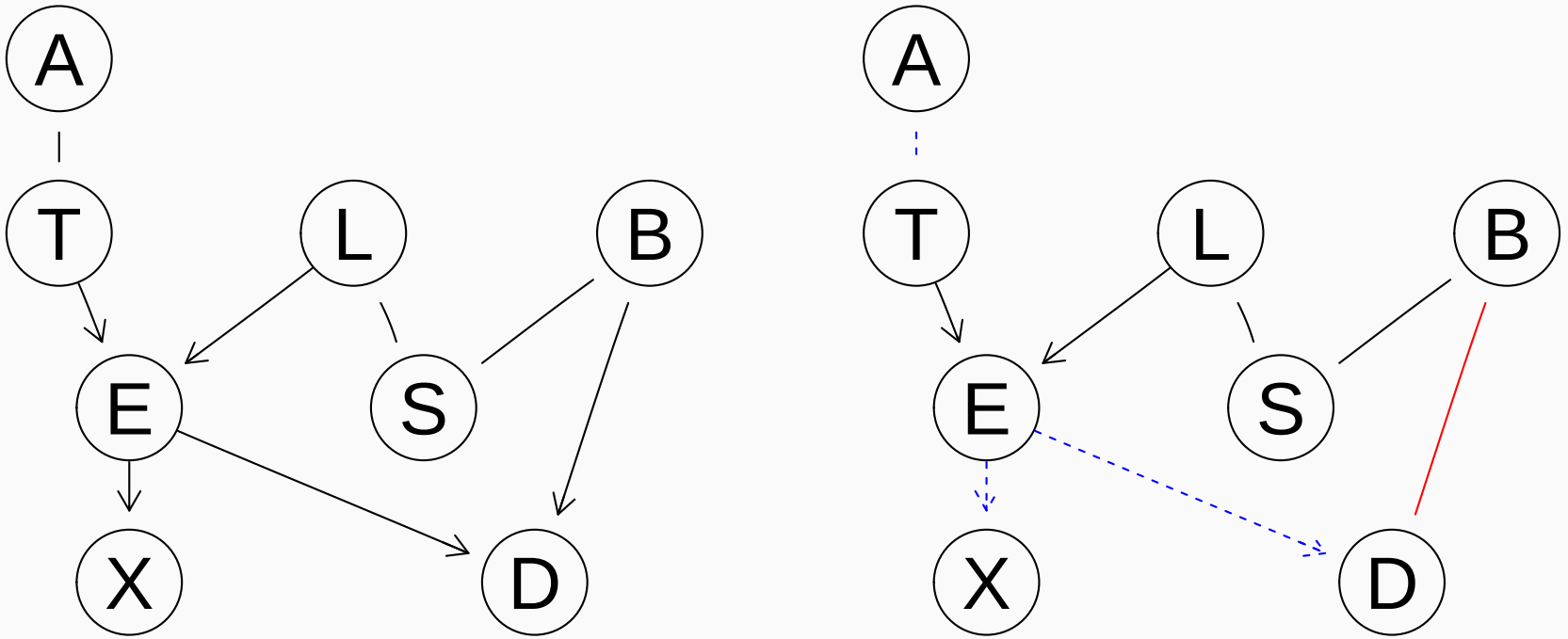
```
asia.dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
cpdag2 = si.hiton.pc(asia, test = "x2", undirected = FALSE)
par(mfrow = c(1, 2))
graph.par(list(nodes = list(fontsize = 10)))
graphviz.compare(asia.dag, cpdag2)
```



Is it really as bad as it looks?

Always Compare CPDAGs

```
par(mfrow = c(1, 2))
graph.par(list(nodes = list(fontsize = 10)))
graphviz.compare(cpdag(asia.dag), cpdag2)
```



It is impossible to uniquely identify the direction of some arcs: `cpdag()` makes that apparent and allows for a fair comparison.

bnlearn: The Debugging Output (I)

```
debugging.output = capture.output(
  si.hiton.pc(asia, test = "mc-mi", undirected = FALSE, debug = TRUE)
)
head(debugging.output, n = 17)
```

```
 [1] "----------------------------------------------------------------"
 [2] "* forward phase for node A ."
 [3] "  * checking nodes for association."
 [4] "  > starting with neighbourhood '  '."
 [5] "  * nodes that are still candidates for inclusion."
 [6] "    > T has p-value 0.0472 ."
 [7] "  * nodes that will be disregarded from now on."
 [8] "    > S has p-value 0.171 ."
 [9] "    > L has p-value 0.516 ."
[10] "    > B has p-value 0.0898 ."
[11] "    > E has p-value 0.132 ."
[12] "    > X has p-value 0.225 ."
[13] "    > D has p-value 0.118 ."
[14] "  @ T accepted as a parent/children candidate ( p-value: 0.0472 )."
[15] "  > current candidates are ' T '."
[16] "----------------------------------------------------------------"
[17] "* forward phase for node S ."
```

bnlearn: The Debugging Output (II)

The debugging output is useful to understand the steps the algorithms perform and to investigate where things go wrong.

```
head(grep("phase", debugging.output, value = TRUE), n = 15)
```

```
   [1] "* forward phase for node A ."
   [2] "* forward phase for node S ."
   [3] "* backward phase for candidate node B ."
   [4] "* backward phase for candidate node E ."
   [5] "* backward phase for candidate node X ."
   [6] "* backward phase for candidate node D ."
   [7] "* forward phase for node T ."
   [8] "* backward phase for candidate node X ."
   [9] "* backward phase for candidate node D ."
  [10] "* forward phase for node L ."
  [11] "* backward phase for candidate node B ."
  [12] "* backward phase for candidate node E ."
  [13] "* backward phase for candidate node X ."
  [14] "* backward phase for candidate node D ."
  [15] "* forward phase for node B ."
```

bnlearn: The Debugging Output (III)

```
head(grep("phase|accepted", debugging.output, value = TRUE), n = 20)
```

```
 [1] "* forward phase for node A ."
 [2] "  @ T accepted as a parent/children candidate ( p-value: 0.0472 )."
 [3] "* forward phase for node S ."
 [4] "  @ L accepted as a parent/children candidate ( p-value: 0 )."
 [5] "* backward phase for candidate node B ."
 [6] "  @ B accepted as a parent/children candidate ( p-value: 0 )."
 [7] "* backward phase for candidate node E ."
 [8] "* backward phase for candidate node X ."
 [9] "* backward phase for candidate node D ."
[10] "* forward phase for node T ."
[11] "  @ E accepted as a parent/children candidate ( p-value: 0 )."
[12] "* backward phase for candidate node X ."
[13] "* backward phase for candidate node D ."
[14] "* forward phase for node L ."
[15] "  @ S accepted as a parent/children candidate ( p-value: 0 )."
[16] "* backward phase for candidate node B ."
[17] "* backward phase for candidate node E ."
[18] "  @ E accepted as a parent/children candidate ( p-value: 0 )."
[19] "* backward phase for candidate node X ."
[20] "* backward phase for candidate node D ."
```

In **bnlearn** we can manually reproduce all the steps performed by constraint-based algorithms, either for debugging purposes or for developing new algorithms.

- We can learn the neighbours of a particular node with any algorithm that learns parents and children (HITON and MMPC).

```
learn.nbr(asia, node = "L", method = "si.hiton.pc", test = "mc-mi")
  [1] "S" "E"
```

- We can learn the Markov blanket of a particular node with any algorithm designed to do that (GS and the IAMB variants).

```
learn.nbr(asia, node = "L", method = "si.hiton.pc", test = "mc-mi")
  [1] "S" "E"
```

BNLEARN: CONDITIONAL INDEPENDENCE TESTS

Another very useful function is `ci.test()`, which performs a single marginal or conditional independence test using the same backends as constraint-based algorithms.

```
options(width = 70)
```

```
ci.test(x = "S", y = "E", z = "L", data = asia, test = "mc-mi")
```

```
        Mutual Information (disc., MC)

data:  S ~ E | L
mc-mi = 0.000004, Monte Carlo samples = 5000, p-value = 1
alternative hypothesis: true value is greater than 0
```

Arguments are much the same as before: `test` specifies the test label, `B` the number of permutations. The test is for x $\perp\!\!\!\perp_P$ y | z where z can be either absent (for marginal tests) or a vector of labels (to condition on one or more variables).

Pros and Cons of Constraint-based Algorithms

- They depend heavily on the quality of the conditional independence tests they use: all proofs of correctness assume tests are always right.
  - Asymptotic tests may make algorithms under-perform.
  - Permutation tests are often too slow, but can be made better with sequential permutations and semi-parametric permutations.
  - Shrinkage tests work better than asymptotic test, but not by much.
- They are consistent, but converge may be slow.
- At any single time they evaluate a small subset of variables, which makes them very memory efficient.
- They do not require multiple testing adjustment, they are self-adjusting (nobody knows why exactly, though).
- They are embarrassingly parallel, so they scale extremely well.

An exhaustive search unfeasible in practice, regardless of the goodness-of-fit measure (called network score) used in the process. However, we can use heuristics in combination with decomposable scores

$$\text{Score}(\mathcal{G}) = \sum_{i=1}^{N} \text{Score}(X_i \,|\, \Pi_{X_i})$$

such as

$$\text{BIC}(\mathcal{G}) = \sum_{i=1}^{N} \log \text{P}(X_i \,|\, \Pi_{X_i}) - \frac{|\Theta_{X_i}|}{2} \log n$$

$$\text{BDe}(\mathcal{G}), \text{BGe}(\mathcal{G}) = \sum_{i=1}^{N} \log \left[\int \text{P}(X_i \,|\, \Pi_{X_i}, \Theta_{X_i}) \, \text{P}(\Theta_{X_i} \,|\, \Pi_{X_i}) d\Theta_{X_i} \right]$$

if we only compare BNs that differ in only one local distribution at a time.

The Hill-Climbing Algorithm

1. Choose an initial network structure $\mathcal{G}$, usually (but not necessarily) empty.
2. Compute the score of $\mathcal{G}$, denoted as $Score_{\mathcal{G}} = \text{Score}(\mathcal{G})$.
3. Set $maxscore = Score_{\mathcal{G}}$.
4. Repeat the following steps as long as $maxscore$ increases:
   - 4.1 for every possible arc addition, deletion or reversal not resulting in a cyclic network:
     - 4.1.1 compute the score of the modified network $\mathcal{G}^*$, $Score_{\mathcal{G}^*} = \text{Score}(\mathcal{G}^*)$:
     - 4.1.2 if $Score_{\mathcal{G}^*} > Score_{\mathcal{G}}$, set $\mathcal{G} = \mathcal{G}^*$ and $Score_{\mathcal{G}} = Score_{\mathcal{G}^*}$.
   - 4.2 update $maxscore$ with the new value of $Score_G$.
5. Return the directed acyclic graph $\mathcal{G}$.

Other optimisation algorithms: tabu search, genetics algorithms, linear programming, constrained optimisation and gradient descent.

DBNs: The Bayesian Dirichlet Marginal Likelihood

If the data $\mathcal{D}$ contain no missing values and assuming:

- a Dirichlet conjugate prior ($X_i \,|\, \Pi_{X_i} \sim Mul\,(\Theta_{X_i} \,|\, \Pi_{X_i})$ and $\Theta_{X_i} \,|\, \Pi_{X_i} \sim Dir\,(\alpha_{ijk})$, $\sum_{jk} \alpha_{ijk} = \alpha_i$ the imaginary sample size);
- positivity (all conditional probabilities $\pi_{ijk} > 0$);
- parameter independence (π_{ijk} for different parent configurations) and modularity (same for π_{ijk} in different $\Theta_{X_i} \,|\, \Pi_{X_i}$);

Heckerman et al. derived a closed form expression for $\mathrm{P}(\mathcal{D} \,|\, \mathcal{G})$:

$$\mathrm{BD}(\mathcal{G}, \mathcal{D}; \boldsymbol{\alpha}) = \prod_{i=1}^{N} \mathrm{BD}(X_i, \Pi_{X_i}; \alpha_i) =$$

$$= \prod_{i=1}^{N} \prod_{j=1}^{q_i} \left[\frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + n_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ijk} + n_{ijk})}{\Gamma(\alpha_{ijk})} \right]$$

where r_i is the number of states of X_i; q_i is the number of configurations of Π_{X_i}; $n_{ij} = \sum_k n_{ijk}$; and $\alpha_{ij} = \sum_k \alpha_{ijk}$.

DBNs: Bayesian Dirichlet Equivalent Uniform (BDeu)

The most common BD score assumes $\alpha_{ijk} = \alpha/(r_i q_i)$, $\alpha_i = \boldsymbol{\alpha}$ and is known as BDeu (Bayesian Dirichlet equivalent uniform). The uniform prior over the parameters was justified by the lack of prior knowledge and widely assumed to be non-informative.

However, there is ample evidence that this is a problematic choice:

- The prior is actually not uninformative.
- MAP DAGs selected using BDeu are highly sensitive to the choice of α and can have markedly different number of arcs even for reasonable α.
- In the limits $\alpha \to 0$ and $\alpha \to \infty$ it is possible to obtain both very simple and very complex DAGs, and model comparison may be inconsistent for small $\mathcal{D}$ and small α.
- The sparseness of the MAP network is determined by a complex interaction between α and $\mathcal{D}$.

Better Than BDeu: Bayesian Dirichlet Sparse (BDs)

If the positivity assumption is violated or the sample size n is small, there may be configurations of some Π_{X_i} that are not observed in $\mathcal{D}$.

$$\mathrm{BDeu}(X_i, \Pi_{X_i}; \alpha) =$$

$$= \prod_{j:n_{ij}=0} \left[\frac{\Gamma(r_i\alpha^*)}{\Gamma(r_i\alpha^*)} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha^*)}{\Gamma(\alpha^*)} \right] \prod_{j:n_{ij}>0} \left[\frac{\Gamma(r_i\alpha^*)}{\Gamma(r_i\alpha^* + n_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha^* + n_{ijk})}{\Gamma(\alpha^*)} \right].$$

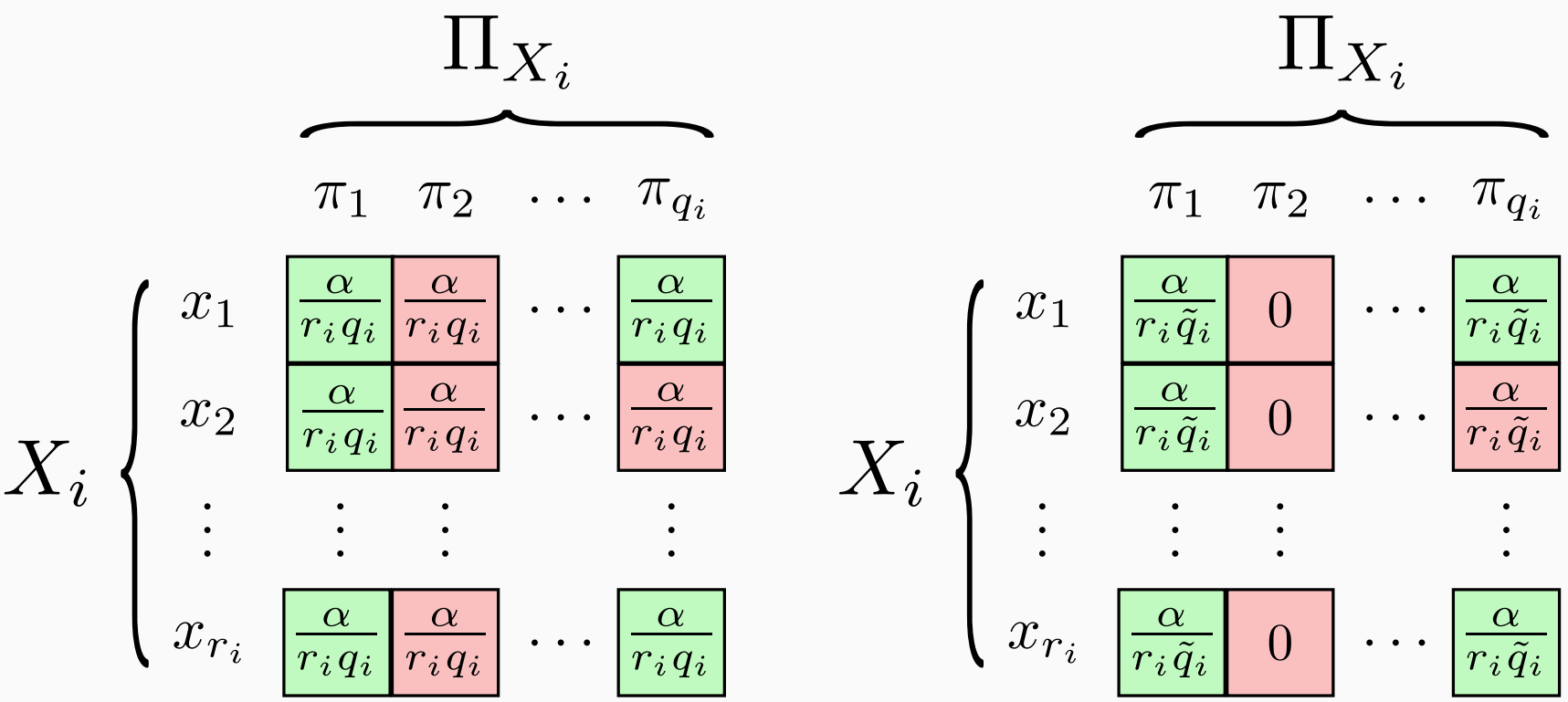
So the effective imaginary sample size decreases as the number of unobserved parents configurations increases, and the MAP estimates of π_{ijk} gradually converge to the ML and favour overfitting.

To address these two undesirable features of BDeu we replace α^* with

$$\tilde{\alpha} = \begin{cases} \alpha/(r_i\tilde{q}_i) & \text{if } n_{ij} > 0 \\ 0 & \text{otherwise} \end{cases}, \quad \tilde{q}_i = \{\text{number of } \Pi_{X_i} \text{ such that } n_{ij} > 0\}$$

and we plug it in BD instead of $\alpha^* = \alpha/(r_i q_i)$ to obtain BDs.

BDeu and BDs Compared



Cells that correspond to $(\mathbf{X}_i, \Pi_{X_i})$ combinations that are not observed in the data are in red, observed combinations are in green.

GBNs: The Bayesian Gaussian Equivalent Score

The Bayesian Gaussian equivalent (BGe) score is defined as the $\mathrm{P}(\mathcal{D} \mid \mathcal{G})$ associated with a normal-Wishart prior $(\boldsymbol{\mu}, W)$ with $\boldsymbol{\mu} \sim N(\boldsymbol{\nu}, \alpha_\mu W)$ and $W \sim Wishart(T, \alpha_w)$:

$$\mathrm{BGe}(X_i, \Pi_{X_i}) = \left(\frac{\alpha_\mu}{N+\alpha_\mu}\right)^{l/2} \frac{\Gamma_l((N+\alpha_w-n+l)/2)}{\pi^{lN/2}\Gamma_l((\alpha_w-n+l)/2)} \frac{|T_{X_i,\Pi_{X_i}}|^{(\alpha_w-n+l)/2}}{|R_{X_i,\Pi_{X_i}}|^{(N+\alpha_w-n+l)/2}}$$

where

$$\Gamma_l\left(\frac{x}{2}\right) = \pi^{l(l-1)/4} \prod_{j=1}^{l} \Gamma\left(\frac{x+1-j}{2}\right),$$

$$R = T + S_N + \frac{N\alpha_w}{N+\alpha_w}(\boldsymbol{\nu} - \bar{\mathbf{x}})(\boldsymbol{\nu} - \bar{\mathbf{x}})^T.$$

(l is defined to be $|X_i \cup \Pi_{X_i}| = |\Pi_{X_i}| + 1$.)

Penalised Likelihoods: AIC and BIC

Penalised likelihoods make for very popular scores. AIC overfits a lot. BIC may under-fit a bit but it is a good default to use. For DBNs, the log-likelihood and the number of parameters associated with a local distribution are:

$$\mathrm{LL}(X_i, \Pi_{X_i}) = \prod_{m=1}^{n} \mathrm{P}(X_i = x_m \,|\, \Pi_{X_i} = \pi_m), \quad |\Theta_{X_i}| = R \times |\Pi_{X_i}|;$$

for GBNs:

$$\mathrm{LL}(X_i, \Pi_{X_i}) = \prod_{m=1}^{n} N(x_m; \boldsymbol{\mu}_{X_i} + \pi_m \boldsymbol{\beta}_{X_i}, \sigma^2_{X_i}), \quad |\Theta_{X_i}| = |\Pi_{X_i}| + 1;$$

for CLGBNS (Δ_{X_i} are the discrete parents, Γ_{X_i} the continuous parents):

$$\mathrm{LL}(X_i, \Pi_{X_i}) = \prod_{m=1}^{n} N(x_m; \boldsymbol{\mu}_{X_i,\delta_m} + \gamma_m \boldsymbol{\beta}_{X_i,\delta_m}, \sigma^2_{X_i,\delta_m}),$$

$$|\Theta_{X_i}| = |\Delta_{X_i}| \times (|\Gamma_{X_i}| + 1).$$

BNLEARN: Hill Climbing with BIC (MARKS)

hc() implements hill-climbing with random restarts, and can use different scores much like functions implementing constraint-based algorithms can use different tests.

```
dag.marks = hc(marks, score = "bic-g")
```

Note that hill-climbing always returns a DAG, not a CPDAG; so the correct way of comparing it with another graph is to take the CPDAG for both.

```
true.dag =
  model2network("[ALG][ANL|ALG][MECH|ALG:VECT][STAT|ALG:ANL][VECT|ALG]")
unlist(compare(dag.marks, true.dag))
```

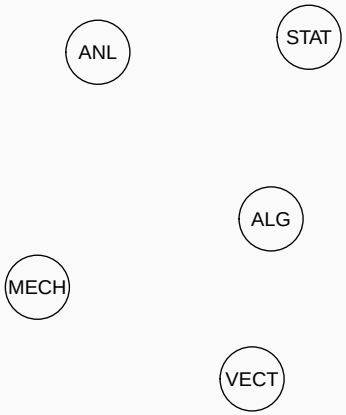
```
 tp fp fn
  3  3  3
```

```
unlist(compare(cpdag(dag.marks), cpdag(true.dag)))
```

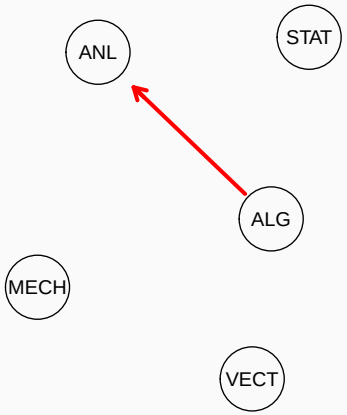
```
 tp fp fn
  6  0  0
```

The Hill-Climbing Algorithm (MARKS)

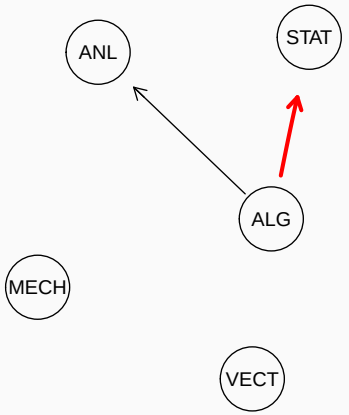
Initial BIC score: −1807.528



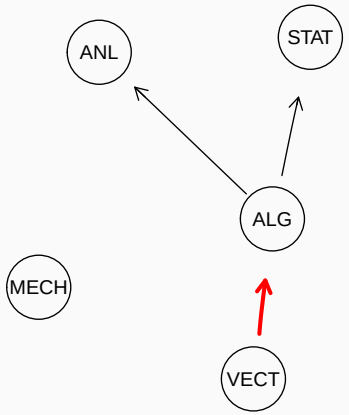
Current BIC score: −1778.804



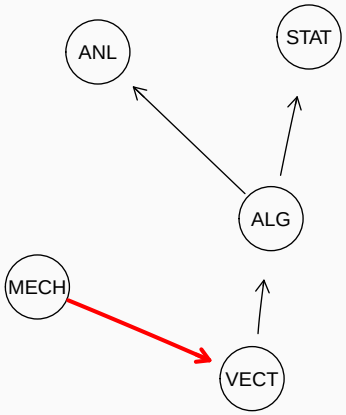
Current BIC score: −1755.383



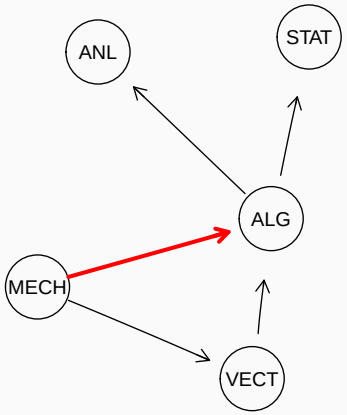
Current BIC score: −1737.176



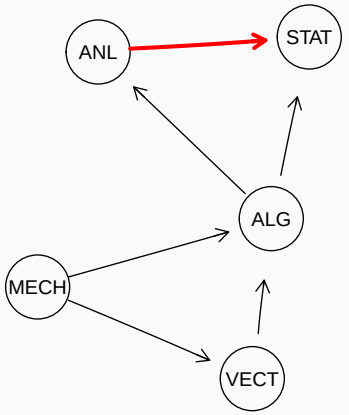
Current BIC score: −1723.325



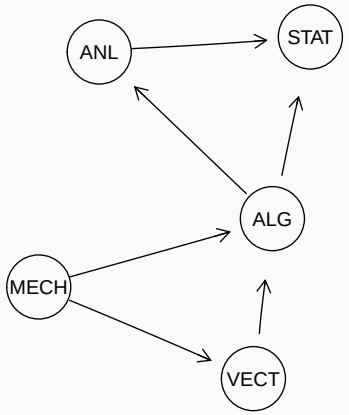
Current BIC score: −1720.901



Current BIC score: −1720.150



Final BIC score: −1720.150



- `compare()` takes two graphs (DAGs, CPDAGs, UGs) and returns a list containing `tp` (true positives), `fp` (false positives) and `fn` (false negatives); directed and undirected arcs are considered different.

```
unlist(compare(dag.marks, true.dag))
```

```
tp fp fn
 3  3  3
```

- `hamming()` computes the Hamming distance between the skeletons of the graphs (zero means a perfect match).

```
hamming(dag.marks, true.dag)
```

```
[1] 0
```

- `shd()` computes the Structural Hamming distance between two CPDAGs, which is similar to the Hamming distance but with a penalty of $1/2$ for directed-undirected arc differences.

```
shd(dag.marks, true.dag)
```

```
[1] 0
```

bnlearn: Hill Climbing with Random Restarts (ASIA)

In addition to scores and their tuning parameters (here `iss` for the imaginary sample size of BDeu), `hc()` has arguments `restart` for the number of random restarts and `perturb` for the number of perturbed arcs in the new starting DAG.

```
asia.restart = hc(asia, score = "bde", iss = 1, restart = 10, perturb = 5)
```

```
debugging.output =
  capture.output(hc(asia, score = "bde", iss = 1, restart = 10,
     perturb = 5, debug = TRUE))
head(grep("^\\* (best|restart)", debugging.output, value = TRUE), n = 10)
```

```
  [1] "* best operation was: adding B -> D ."
  [2] "* best operation was: adding L -> E ."
  [3] "* best operation was: adding E -> X ."
  [4] "* best operation was: adding S -> B ."
  [5] "* best operation was: adding T -> E ."
  [6] "* best operation was: adding E -> D ."
  [7] "* best operation was: adding S -> L ."
  [8] "* best operation was: adding L -> E ."
  [9] "* best operation was: adding E -> X ."
 [10] "* best operation was: adding S -> L ."
```

Why Do We Want Random Restarts?

Random restarts reduce the probability of getting stuck in a local maximum by jumping away from it. The DAG we jump to is created by perturbing the DAG that was identified as a local maximum, that is, by changing a number of its arcs to create a new DAG.

```
head(grep("^\\* (current score|doing)", debugging.output, value = TRUE), 14)
```

```
 [1] "* current score: -15225 "
 [2] "* current score: -14043 "
 [3] "* current score: -12955 "
 [4] "* current score: -12026 "
 [5] "* current score: -11579 "
 [6] "* current score: -11348 "
 [7] "* current score: -11217 "
 [8] "* current score: -11096 "
 [9] "* doing a random restart, 9 of 10 left."
[10] "* current score: -12150 "
[11] "* current score: -11220 "
[12] "* current score: -11099 "
[13] "* current score: -11096 "
[14] "* doing a random restart, 8 of 10 left."
```

bnlearn: Hill-Climbing With Preseeded Networks

Another way to avoid getting stuck in local maxima is to start the search from a different network. The default is to start from the empty DAG.

```
capture.output(hc(asia, score = "bde", iss = 1, debug = TRUE))[c(2, 6:7)]
```

```
  [1] "* starting from the following network:"
  [2] "  model:"
  [3] "   [A][S][T][L][B][E][X][D] "
```

However, we can specify an alternative starting DAG with the `start` argument. Here we generate one at random with `random.graph()`.

```
capture.output(hc(asia, score = "bde", iss = 1,
  start = random.graph(names(asia)), debug = TRUE))[c(2, 6:7)]
```

```
  [1] "* starting from the following network:"
  [2] "  model:"
  [3] "   [A][S][T][L|A:S:T][E|A:S][B|S:T:L][X|S:E][D|A:L] "
```

The principle is the same as, say, starting k-means from different sets of centroids and keeping the clustering that fits the data best.

BNLEARN: TABU SEARCH

In addition to `hc()`, **bnlearn** implements `tabu()` with arguments `tabu` (the length of the tabu list) and `max.tabu` (the maximum number of iterations `tabu()` can perform without improving the best network score).

```
debugging.output =
  capture.output(tabu(asia, score = "bde", iss = 1, tabu = 10,
     max.tabu = 5, debug = TRUE))
head(grep("^\\* (best operation|network)", debugging.output, value = TRUE), 10)
```

```
 [1] "* best operation was: adding B -> D ."
 [2] "* best operation was: adding L -> E ."
 [3] "* best operation was: adding E -> X ."
 [4] "* best operation was: adding S -> B ."
 [5] "* best operation was: adding T -> E ."
 [6] "* best operation was: adding E -> D ."
 [7] "* best operation was: adding S -> L ."
 [8] "* network score did not increase (for 1 times), looking for a minimal decreas
 [9] "* best operation was: reversing S -> L ."
[10] "* network score did not increase (for 2 times), looking for a minimal decreas
```

Pros and Cons of Score-based Algorithms

- Convergence to the global maximum (the best structure) is not guaranteed for finite samples, the search may get stuck in a local maximum.
- They are more stable than constraint-based algorithms.
- They require a definition of both the global and the local distributions, and a matching decomposable, network score. This means, for instance, that we cannot use them with ordinal variables because it is difficult to specify the global distribution. On the other hand, there are trend tests to use for conditional independence.
- Most scores have tuning parameters, whereas conditional independence tests (mostly) do not; and algorithms have tuning parameters as well. This usually means a grid of values to be tested under cross-validation to select the optimal learning strategy.

Hybrid Structure Learning Algorithms

Hybrid algorithms combine constraint-based and score-based algorithms to complement the respective strengths and weaknesses; they are considered the state of the art in current literature.

They work by alternating the following two steps:

- learn some conditional independence constraints to restrict the number of candidate networks;
- find the network that maximises some score function and that satisfies those constraints and define a new set of constraints to improve on.

These steps can be repeated several times (until convergence), but one or two times is usually enough.

The Sparse Candidate Algorithm and MMHC

1. Choose a network structure $\mathcal{G}$, usually (but not necessarily) empty.
2. Repeat the following steps until convergence:
   - 2.1 **restrict:** select a set $\mathbf{C}_i$ of candidate parents for each node $X_i \in \mathbf{X}$, which must include the parents of X_i in $\mathcal{G}$;
   - 2.2 **maximise:** find the network structure $\mathcal{G}^*$ that maximises $\mathrm{Score}(\mathcal{G}^*)$ among the networks in which the parents of each node X_i are included in the corresponding set $\mathbf{C}_i$;
   - 2.3 set $\mathcal{G} = \mathcal{G}^*$.
3. Return the directed acyclic graph $\mathcal{G}$.

If we iterate only once, using MMPC for the restrict phase and hill-climbing for the maximise phase we obtain the Max-Min Hill-Climbing (MMHC) algorithm as a particular case.

rsmax2() implements a single step of the Sparse Candidate algorithm: it runs the restrict and maximise phases only once.

```
asia.rsmax2 =
  rsmax2(asia, restrict = "si.hiton.pc",  maximize = "tabu",
    restrict.args = list(test = "x2", alpha = 0.01),
    maximize.args = list(score = "bic", tabu = 10))
```

Its main arguments are:

- restrict: constraint-based algorithm to use in the restrict phase;
- restrict.args: its optional arguments;
- maximize: score-based algorithm to use in the maximise phase;
- maximize.args: its optional arguments.

The following two commands are equivalent:

```
rsmax2(asia, restrict = "mmpc", maximize = "hc")
mmhc(asia)
```

And from the debugging output we can see that is the case:

```
debugging.output = capture.output(print(mmhc(asia, debug = TRUE)))
grep("restrict|maximize|method:", debugging.output, value = TRUE)
```

```
  [1] "* restrict phase, using the Max-Min Parent Children algorithm."
  [2] "* maximize phase, using the Hill-Climbing algorithm."
  [3] "  constraint-based method:                  "
  [4] "  score-based method:                       Hill-Climbing "
```

```
debugging.output =
  capture.output(print(rsmax2(asia, restrict = "mmpc", maximize = "hc",
    debug = TRUE)))
grep("restrict|maximize|method:", debugging.output, value = TRUE)
```

```
  [1] "* restrict phase, using the Max-Min Parent Children algorithm."
  [2] "* maximize phase, using the Hill-Climbing algorithm."
  [3] "  constraint-based method:                  "
  [4] "  score-based method:                       Hill-Climbing "
```

Pros and Cons of Hybrid Algorithms

- You can mix and match conditional independence tests and network scores with structure learning algorithms, since the latter do not depend on the nature of the data. They can range from frequentist to Bayesian to information-theoretic and anything in between (within reason).
- Constraint-based algorithms are usually faster, score-based algorithms are more stable. Hybrid algorithms are at least as good as score-based algorithms, and often a bit faster.
- Tuning parameters can be difficult to tune for some configurations of algorithms, tests and scores.

A Final Comparison

In this particular case, hill-climbing with random restarts wins the day.

```
true.dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
unlist(compare(cpdag(asia.rsmax2), cpdag(true.dag)))
```

```
  tp fp fn
   4  4  1
```

```
shd(asia.rsmax2, true.dag)
```

```
  [1] 4
```

```
unlist(compare(cpdag(asia.restart), cpdag(true.dag)))
```

```
  tp fp fn
   7  1  0
```

```
shd(asia.restart, true.dag)
```

```
  [1] 1
```

```
unlist(compare(cpdag(cpdag2), cpdag(true.dag)))
```

```
  tp fp fn
   4  4  1
```

```
shd(cpdag2, true.dag)
```

```
  [1] 4
```

The most common choice for $\mathrm{P}(\mathcal{G})$ is the uniform distribution because it is feels like it is uninformative. However, it is problematic because:

- Score-based structure learning algorithms typically generate new candidate DAGs by a single arc additions, deletions or reversals:

$$\frac{\mathrm{P}(\mathcal{G} \cup \{X_j \to X_i\} \mid \mathcal{D})}{\mathrm{P}(\mathcal{G} \mid \mathcal{D})} = \frac{\mathrm{P}(\mathcal{G} \cup \{X_j \to X_i\})}{\mathrm{P}(\mathcal{G})} \frac{\mathrm{P}(\mathcal{D} \mid \mathcal{G} \cup \{X_j \to X_i\})}{\mathrm{P}(\mathcal{D} \mid \mathcal{G})}.$$

  The prior always simplifies, and that implies all operations have a prior probability of $1/3$. The probability that two nodes up connected is then $2/3$.

- Two arcs are correlated in the prior if they are incident on a common node, so false positives and false negatives can potentially propagate through $\mathrm{P}(\mathcal{G})$ and lead to further errors in learning $\mathcal{G}$.

- DAGs that are completely unsupported by the data have most of the probability mass for large enough N.

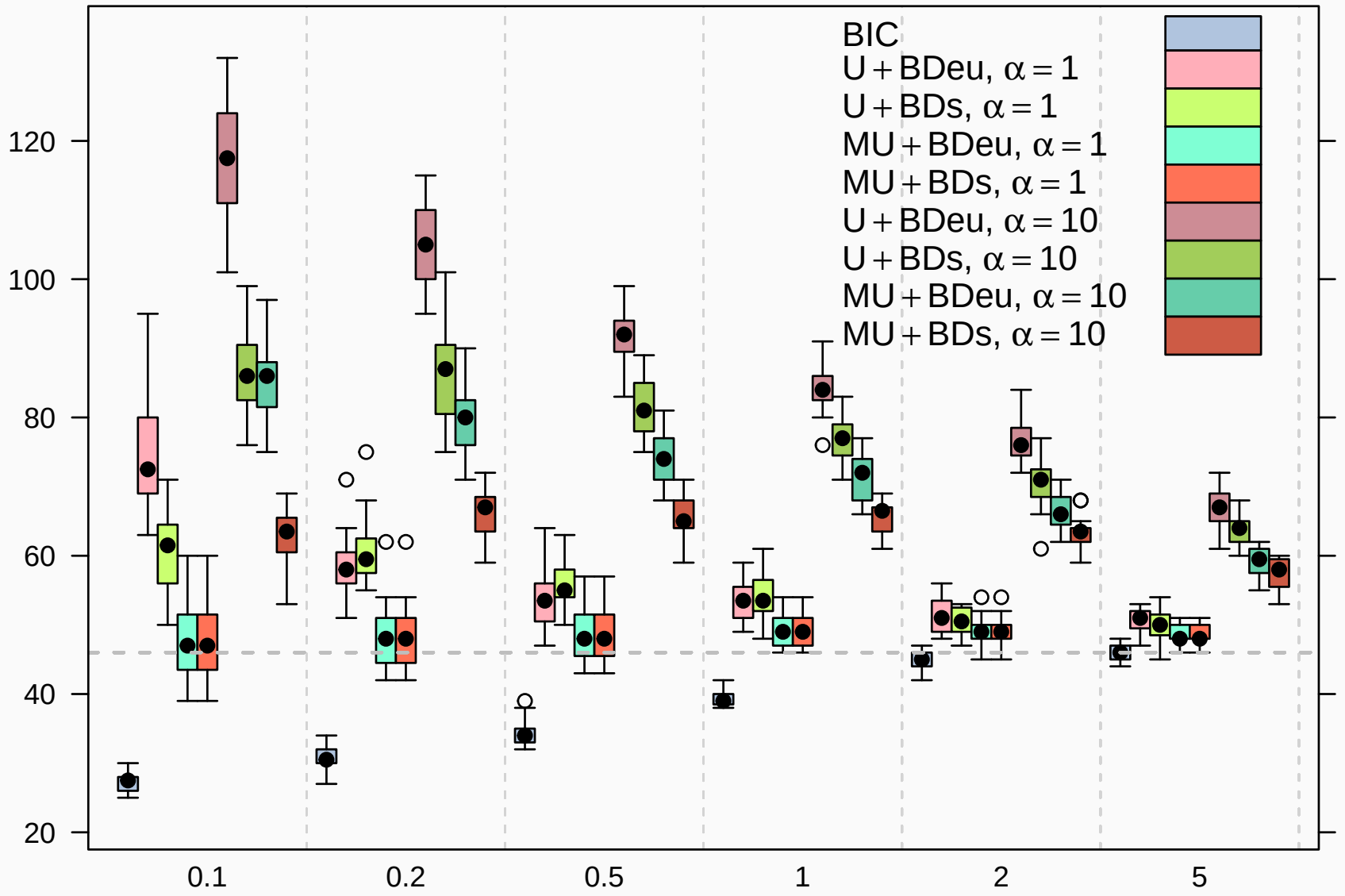
Sparse Graphical Priors

We want our BNs to be sparse: we should express this fact in $\mathrm{P}(\mathcal{G})$. The simplest option is to use a marginal uniform prior that:

- Does not favour arc inclusion, which should have probability $\leqslant 1/2$.
- Does not favour the propagation of errors in structure learning because arcs are independent from each other.
- Is computationally trivial to use: including an arc with probability $\leqslant 1/4$, its reverse with the same probability, and not including an arc with probability $\geqslant 1/2$.

A DAG can contain $\frac{N(N-1)}{2}$ arcs. We can set the probability of inclusion to $c\frac{2}{N-1}, c \in [1, 3]$ to have $O(cN)$ expected arcs in the prior, which often works even better.

BDe and BDs Scores, Uniform and Marginal Uniform Priors



Informative Graphical Priors

When we have reliable information elicited from experts, we may also want to use an informative prior to force structure learning to include (or not) specific arcs patterns. Some examples:

- limiting the number of parents for some or all nodes;
- setting different probabilities for inclusion for different arcs (the Castelo & Siebes prior);
- whitelisting or blacklisting specific arcs;
- setting the topological ordering of the topological ordering, effectively allowing only one direction for each arc.

All these informative priors concentrate the prior probability mass on DAGs we know from experts to be the most sensible. They can also rule out a large portion of the possible DAGs (giving them a zero probability), reducing the search space by orders of magnitude.

BNLEARN: Graphical Priors with Hill-Climbing

```
hc(marks, maxp = 3)
```

```
wl = data.frame(
  from = c("ANL", "ANL"),
  to = c("ALG", "MECH")
)
bl = data.frame(
  from = c("MECH", "MECH"),
  to = c("ALG", "VECT")
)
hc(marks, whitelist = wl, blacklist = wl)
```

```
hc(marks, score = "bge", prior = "marginal", beta = 2 / (ncol(marks) - 1))
beta = data.frame(from = c("MECH", "ALG"), to = c("ALG", "MECH"),
          prob = c(0.2, 0.6))
hc(marks, score = "bge", prior = "cs", beta = beta)
```

```
bl = tiers2blacklist(list(c("ANL", "ALG"), c("VECT", "STAT"), "MECH"))
hc(marks, blacklist = bl)
bl = ordering2blacklist(c("ANL", "ALG", "VECT", "STAT", "MECH"))
hc(marks, blacklist = bl)
```

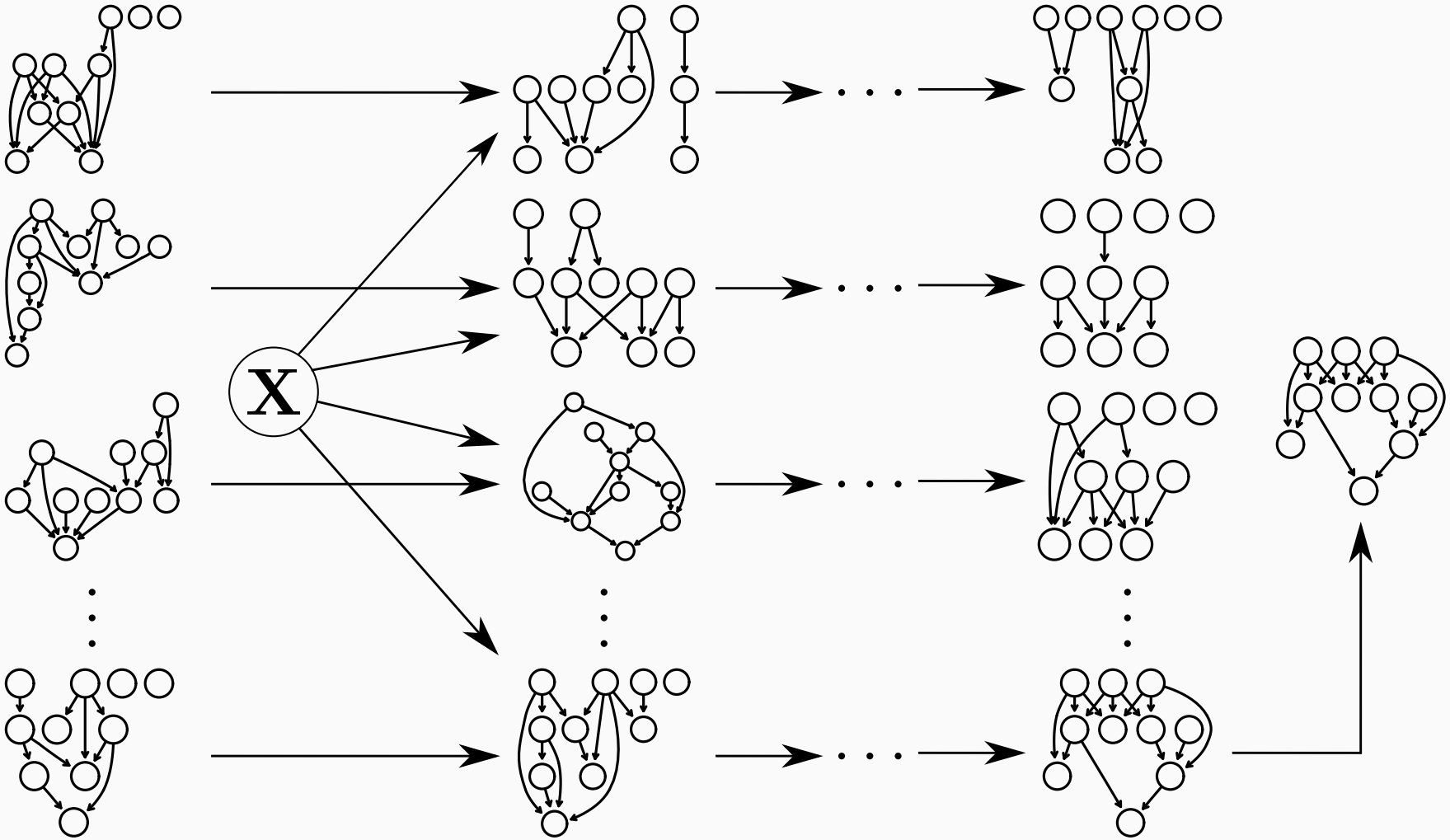
Model Averaging: We Should Always Do That!

Structure learning from limited amounts of data is inherently noisy. Perturbing the data, learning multiple DAGs and then averaging them is most effective in removing that noise. There are two main ways of doing that:

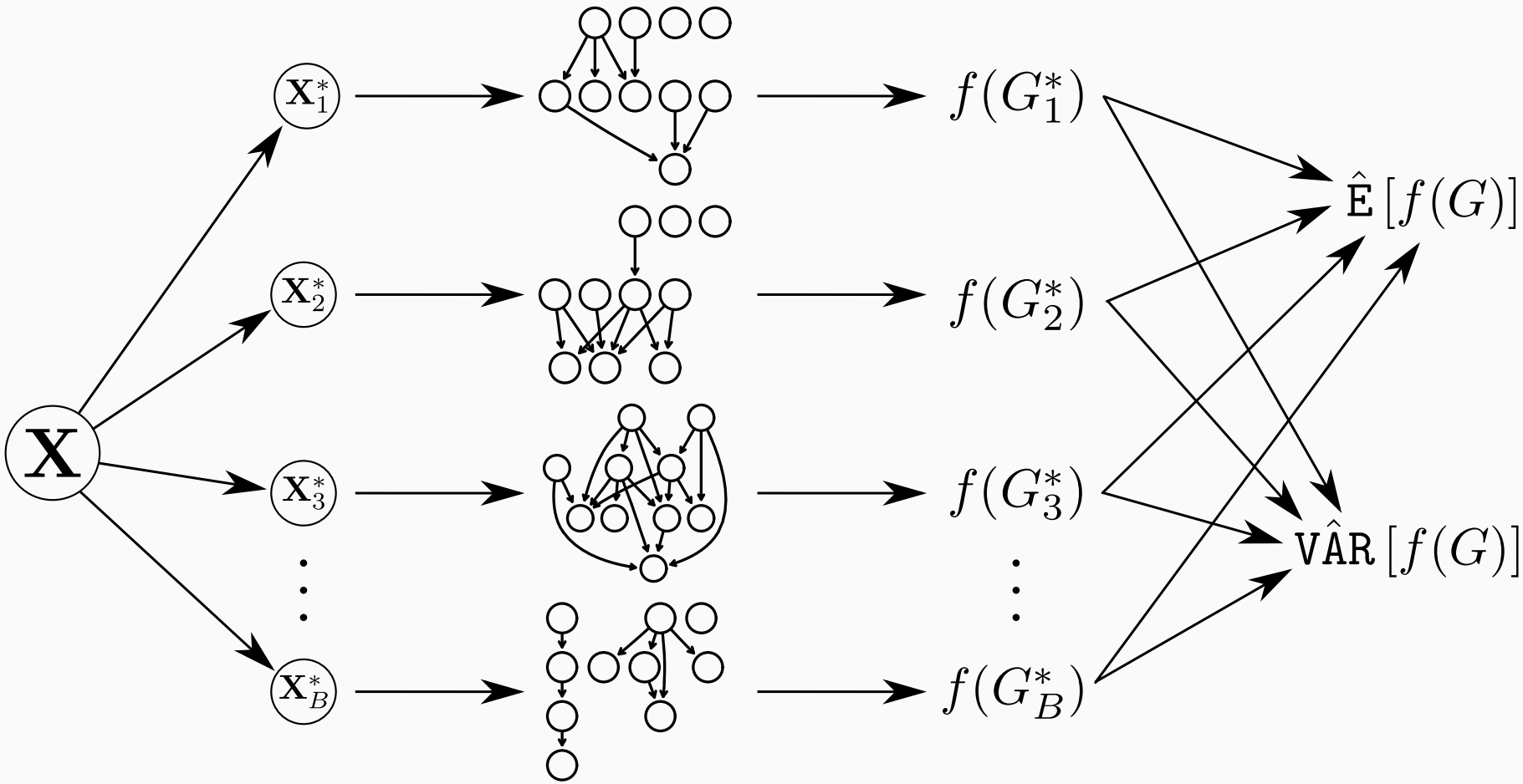
1. Searching from different starting points increases our coverage of the space of the possible DAGs and uses all observations.
2. Learning multiple DAGs from bootstrap samples (that is, bagging) perturbs the data reducing the impact of outliers.
3. Both.

The frequency with which an arc appears is a measure of the strength of the dependence. In other words, it quantifies our confidence that the arc is real. This is separate from the magnitude or the sign of the statistical effect the arcs represents.

Multiple Starting Points and the Same Data



Same Starting Point and Bootstrapped Data



BNLEARN: Model Averaging (I)

```
nodes = names(marks)
start = random.graph(nodes = nodes, method = "ic-dag", num = 500, every = 50)
netlist = lapply(start,
  function(dag) hc(marks, start = dag)
)
str = custom.strength(netlist, nodes = nodes)
head(str)
```

```
    from   to strength direction
1 MECH VECT    1.000       0.5
2 MECH  ALG    1.000       0.5
3 MECH  ANL    0.086       0.5
4 MECH STAT    0.066       0.5
5 VECT MECH    1.000       0.5
6 VECT  ALG    1.000       0.5
```

```
str = boot.strength(marks, algorithm = "hc")
head(str)
```

```
    from   to strength direction
1 MECH VECT    0.830     0.521
2 MECH  ALG    0.825     0.509
3 MECH  ANL    0.095     0.737
4 MECH STAT    0.030     0.583
5 VECT MECH    0.830     0.479
6 VECT  ALG    0.940     0.500
```

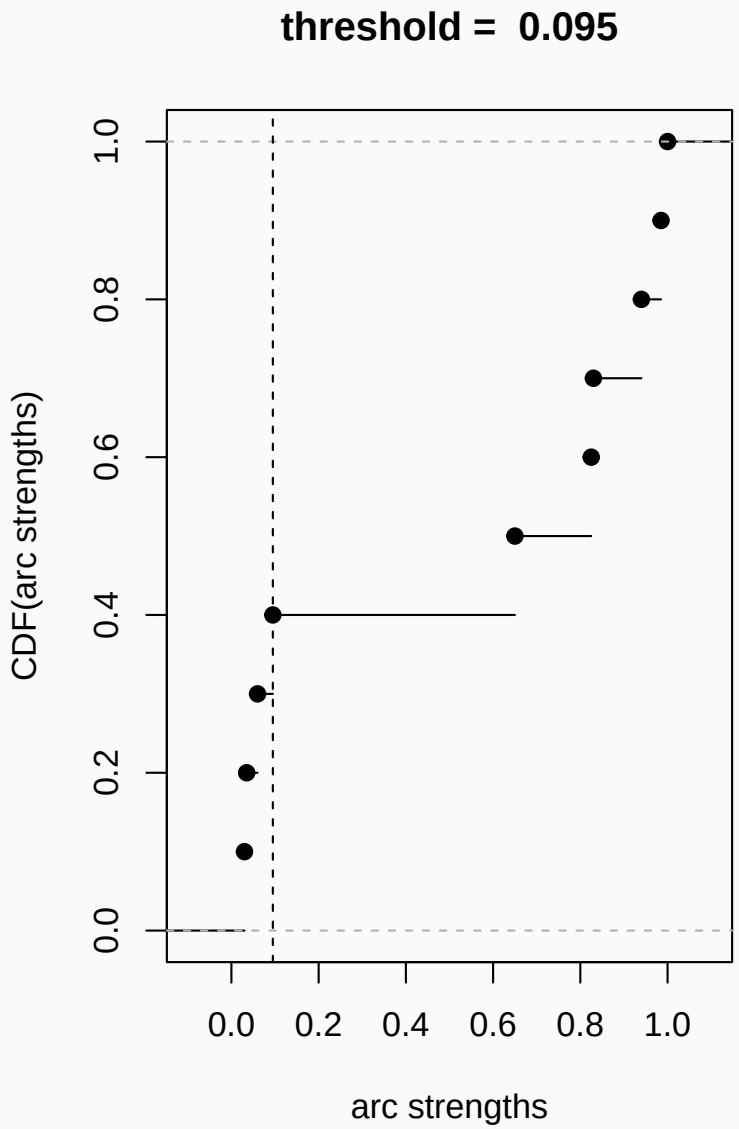
BNLEARN: Model Averaging (II)

We need a threshold to decide which arcs are strong enough to be included in the consensus DAG that is the output of model averaging: we can either estimate it from the data

```
averaged.network(str)
```

or pick one ourselves.

```
averaged.network(str, threshold = 0.90)
```



Who Learns the Best Network Structures

Bayesian network Structure learning is defined by the combination of a statistical criterion and an algorithm that determines how the criterion is applied to the data. What we can say about the algorithms?

Q1 *Which of constraint-based and score-based algorithms provide the most accurate structural reconstruction?*

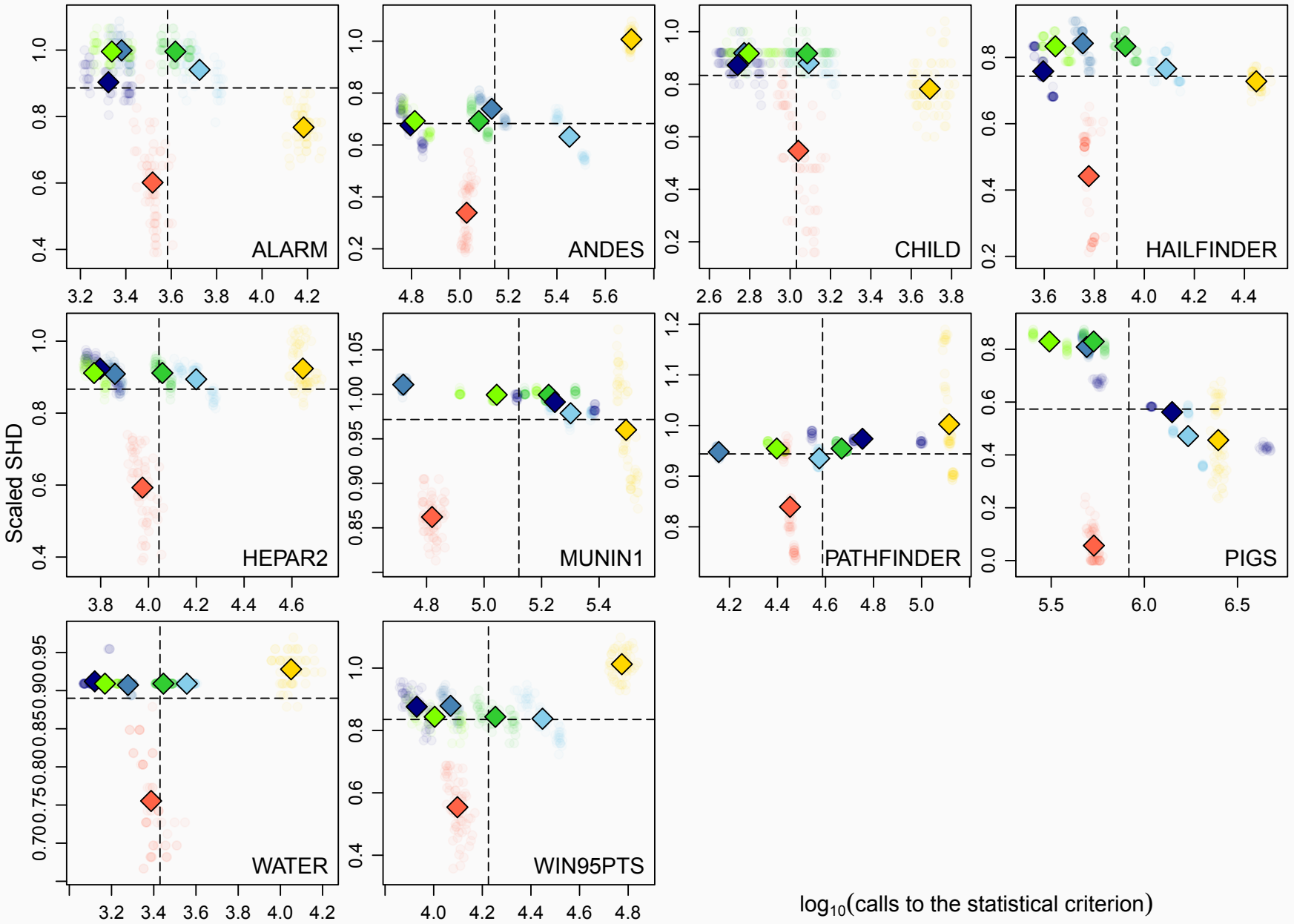
Q2 *Are hybrid algorithms more accurate than constraint-based or score-based algorithms?*

Q3 *Are score-based algorithms slower than constraint-based and hybrid algorithms?*

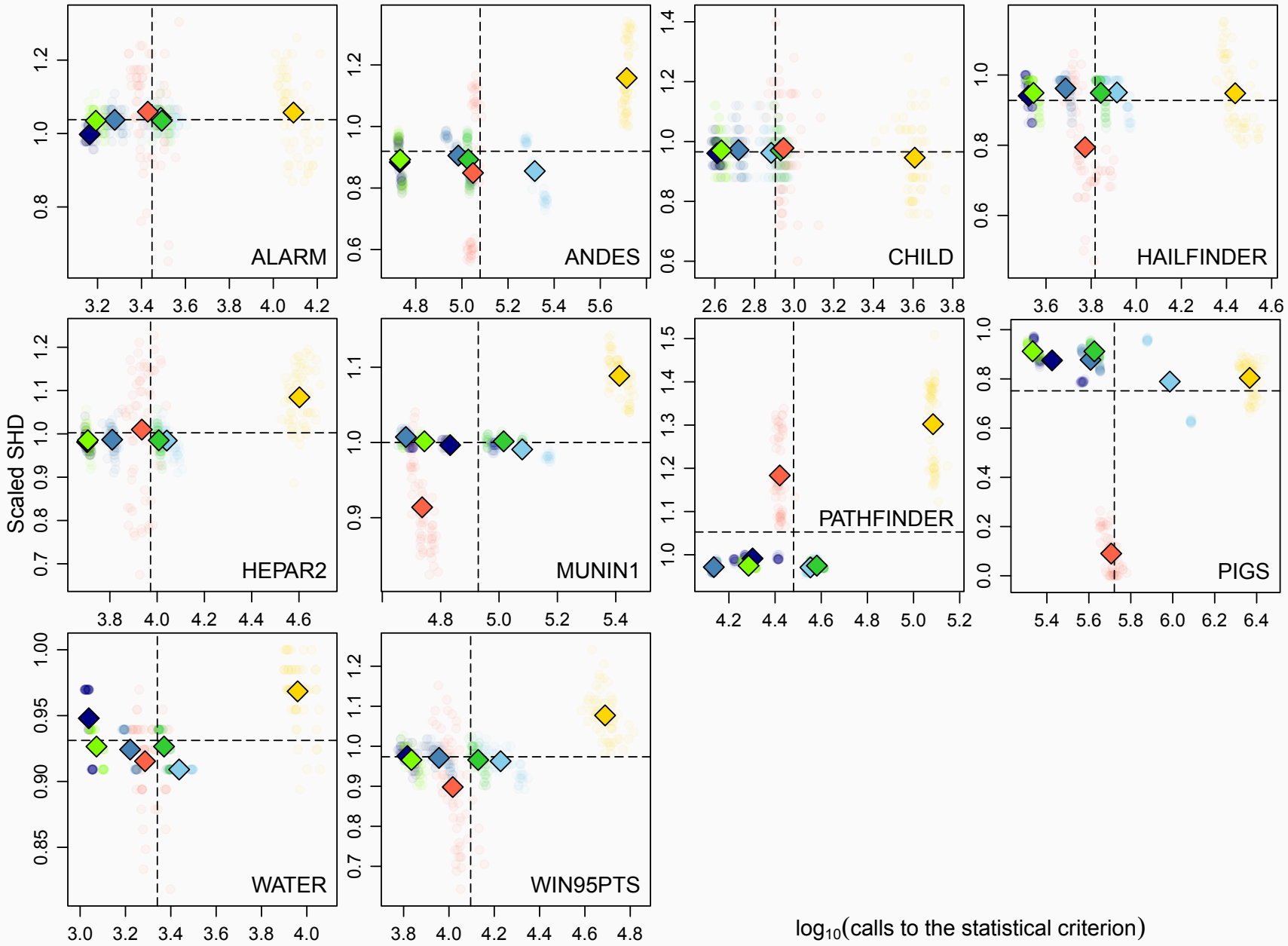
Q4 *Are hybrid algorithms faster than constraint-based or score-based algorithms?*

Q5 *Do the different classes of algorithms present any systematic difference in either speed or accuracy when learning small networks and large networks?*

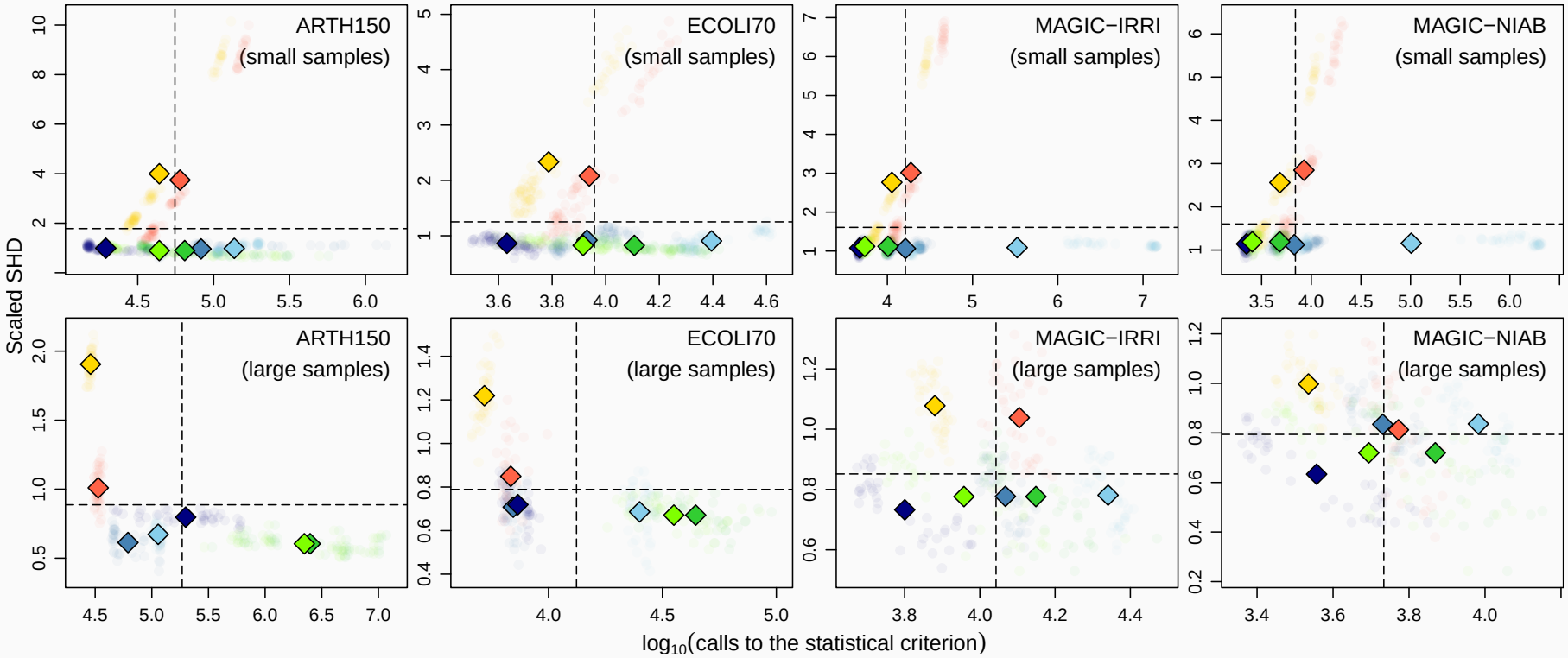
Discrete Bayesian Networks (Large Samples)



Discrete Bayesian Networks (Small Samples)



Gaussian Bayesian Networks



Constraint-based algorithms are in blue shades, hybrid algorithms in green shades, score-based in warm colours. Tabu search is red, the PC algorithm is navy blue.

Conclusions

We can say that, broadly speaking:

- **Q1** Constraint-based algorithms are more accurate than score-based algorithms for small sample sizes.
- **Q2** They are as accurate as hybrid algorithms.
- **Q3** Tabu search, as a score-based algorithm, is faster than constraint-based algorithms more often than not.
- **Q4** Hybrid algorithms are not faster overall than constraint-based or score-based algorithms. In fact, there is no consistent ordering of the algorithms from these classes.
- **Q5** No systematic difference in the ranking of different classes of algorithms in terms of speed and accuracy was observed for any class of algorithms for small networks compared to large networks.

This in contrast with the general view in the older literature that only studied trivially small problems.

Relevant Functions in bnlearn

- structure learning algorithms: `pc.stable()`, `hc()`, `tabu()`, `rsmax2()`, etc.
- graphical distances: `shd()`, `hamming()` and `compare()`.
- graphical comparisons: `graphviz.compare()`
- model averaging: `boot.strength()`, `custom.strength()` and `averaged.network()`.

Summary

- Learning the structure of a BN is the first and most crucial step in learning a BN, whether from data or from expert knowledge.
- There are three classes of algorithms to learn the structure of a BN from data: constraint-based, score-based and hybrid.
- The algorithms in these three classes are defined without requiring any specific type of data, which means that it is possible to mix and match tests and scores with algorithms.
- Different classes of algorithms have different strengths and weaknesses. Score-based algorithms are in more common use in practice.
- Scores, tests and algorithms all have tuning parameters and it is usually not clear how their choice impacts the learned networks and how much.
- There is no "best" algorithm: different algorithms will be "best" with different data sets and for different tasks.

Thanks!

Any questions?