Understanding Bayesian Networks

with Examples in R

Marco Scutari

scutari@stats.ox.ac.uk
Department of Statistics
University of Oxford

January 23–25, 2017

Definitions

A Graph and a Probability Distribution

Bayesian networks (BNs) are defined by:

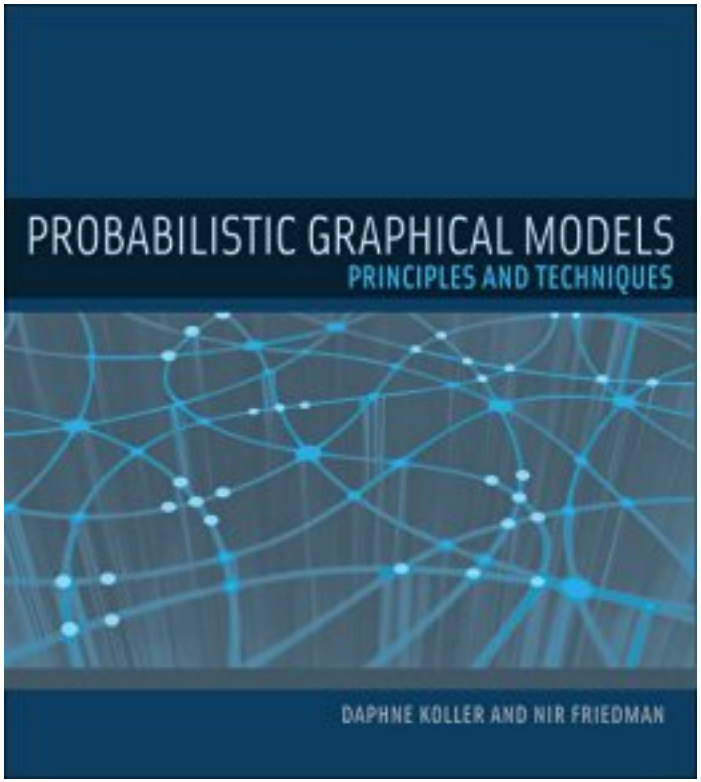
- a network structure, a directed acyclic graph $\mathcal{G} = (\mathbf{V}, A)$, in which each node $v_i \in \mathbf{V}$ corresponds to a random variable X_i;
- a global probability distribution $\mathbf{X}$ with parameters Θ, which can be factorised into smaller local probability distributions according to the arcs $a_{ij} \in A$ present in the graph.

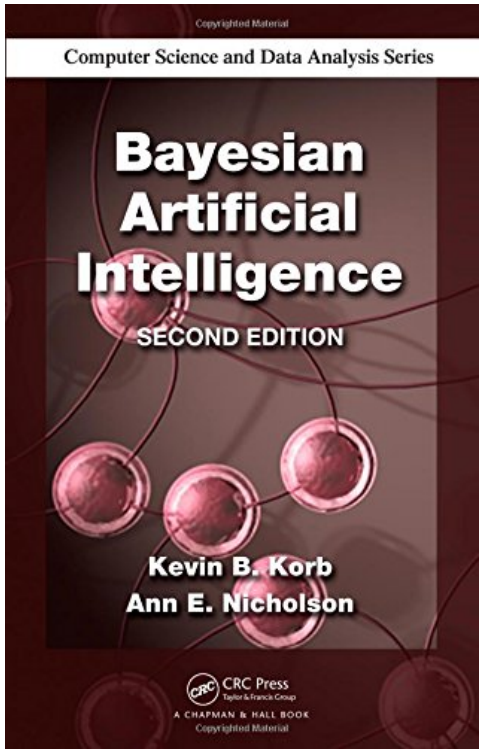
The main role of the network structure is to express the conditional independence relationships among the variables in the model through graphical separation, thus specifying the factorisation of the global distribution:

$$\mathrm{P}(\mathbf{X}) = \prod_{i=1}^{N} \mathrm{P}(X_i \mid \Pi_{X_i}; \Theta_{X_i}) \quad \text{where} \quad \Pi_{X_i} = \{\text{parents of } X_i\}$$

UNIVERSITY OF OXFORD

Where to Look: Book References





(Best perused as ebooks, the Koller & Friedman is $\approx 2^1/_2$ inches thick.)

How to Use: Software References

DISCLAIMER: I am the author of the **bnlearn** R package and I will use it for the most part in this course.

```
install.packages("bnlearn")
```

For displaying graphs, I will use the **Rgraphviz** from BioConductor:

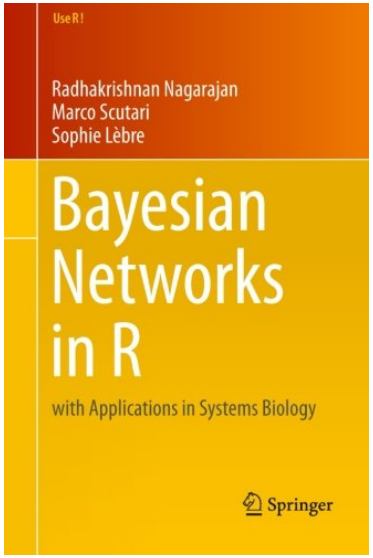
```
source("http://bioconductor.org/biocLite.R")
biocLite(c("graph", "Rgraphviz"))
```

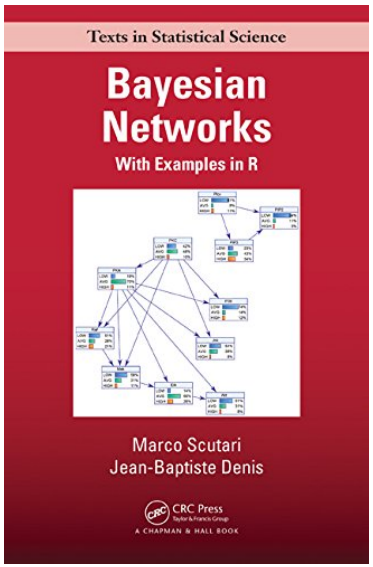
For exact inference on discrete Bayesian networks:

```
source("http://bioconductor.org/biocLite.R")
biocLite(c("graph", "Rgraphviz", "RBGL"))
install.packages("gRain")
```

Other packages from CRAN:

```
install.packages(c("pcalg", "catnet", "abn"))
```





Graphs

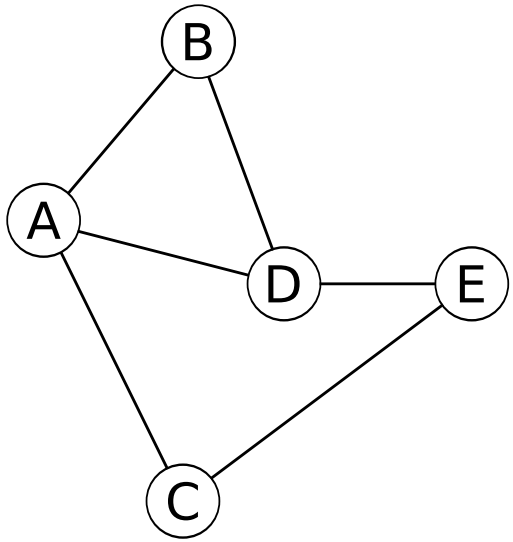
The first component of a BN is a graph. A graph $\mathcal{G}$ is a mathematical object with:

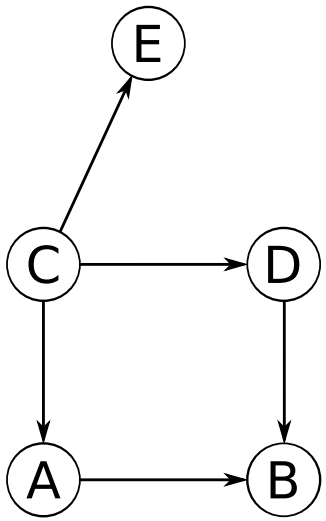
- a set of nodes $\mathbf{V} = \{v_1, \ldots, v_N\}$;
- a set of arcs A which are identified by pairs for nodes in $\mathbf{V}$, e.g. $a_{ij} = (v_i, v_j)$.

Given $\mathbf{V}$, a graph is uniquely identified by A. The arcs in A can be:

- undirected if (v_i, v_j) is an unordered pair and the arc $v_i - v_j$ has no direction;
- directed if $(v_i, v_j) \neq (v_j, v_i)$ is an ordered pair and the arc has a specific direction $v_i \rightarrow v_j$.

The assumption is that there is at most one arc between a pair of nodes.

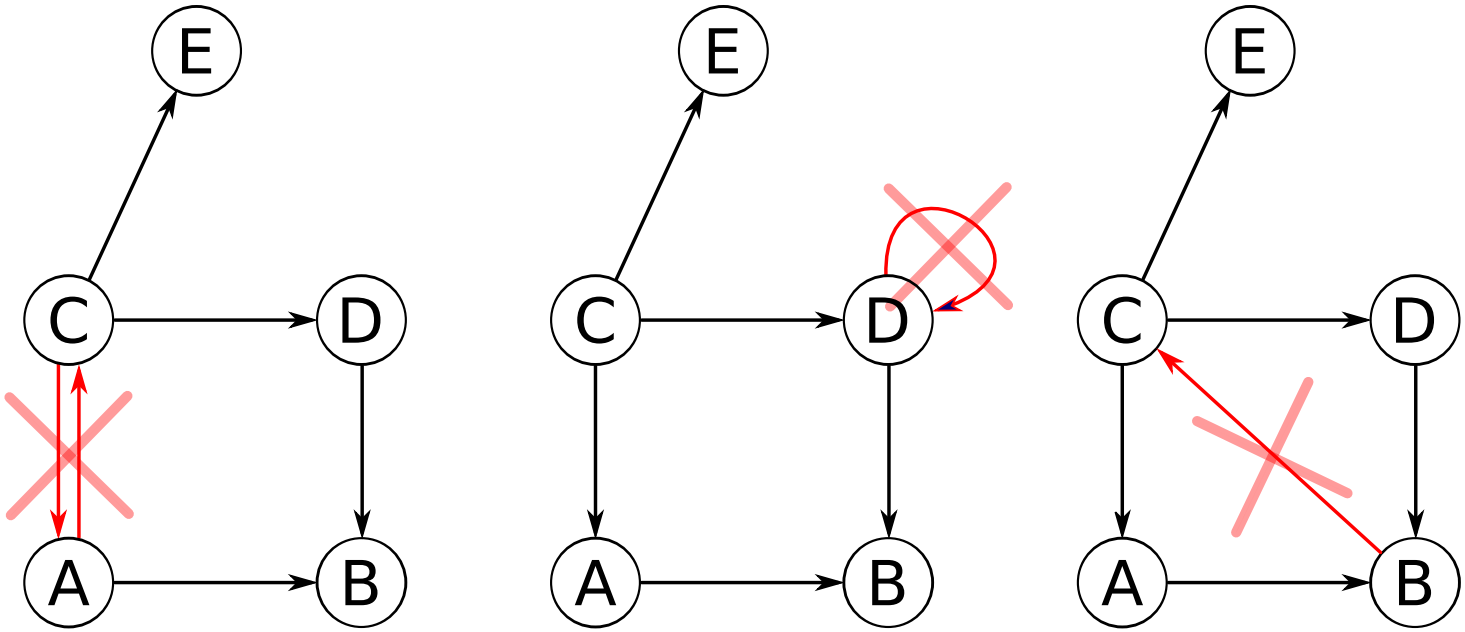




Directed Acyclic Graphs

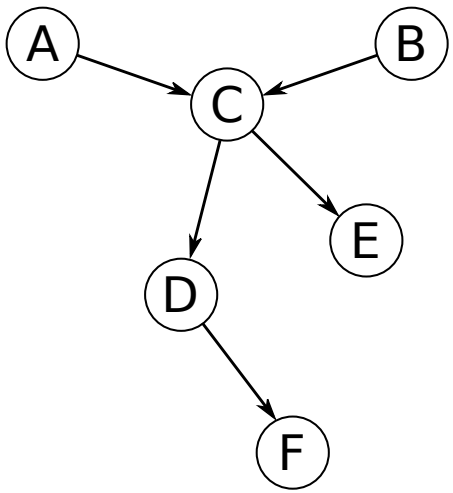
BNs use a specific kind of graph called a directed acyclic graph, that:

- contains only directed arcs;
- does not contain any loop (e.g. an arc $v_i \rightarrow v_i$ from a node to itself);
- does not contain any cycle (e.g. a sequence of arcs $v_i \rightarrow v_j \rightarrow \ldots \rightarrow v_k \rightarrow v_i$ that starts and ends in the same node).



How the DAG Maps to the Probability Distribution

DAG



$\Longrightarrow$

Graphical separation

$$
\begin{aligned}
&A \perp\!\!\!\perp_G B \\
&A \perp\!\!\!\perp_G D \mid C \\
&B \perp\!\!\!\perp_G D \mid C \\
&A \perp\!\!\!\perp_G E \mid C \\
&B \perp\!\!\!\perp_G E \mid C \\
&D \perp\!\!\!\perp_G E \mid C \\
&C \perp\!\!\!\perp_G F \mid D \\
&\dots
\end{aligned}
$$

$\Longrightarrow$

Probabilistic independence

$$
\begin{aligned}
&A \perp\!\!\!\perp_P B \\
&A \perp\!\!\!\perp_P D \mid C \\
&B \perp\!\!\!\perp_P D \mid C \\
&A \perp\!\!\!\perp_P E \mid C \\
&B \perp\!\!\!\perp_P E \mid C \\
&D \perp\!\!\!\perp_P E \mid C \\
&C \perp\!\!\!\perp_P F \mid D \\
&\dots
\end{aligned}
$$

Formally, the DAG is an independence map of the probability distribution of $\mathbf{X}$, with graphical separation ($\perp\!\!\!\perp_G$) implying probabilistic independence ($\perp\!\!\!\perp_P$).

Maps

Let M be the dependence structure of the probability distribution P of $\mathbf{X}$, that is, the set of conditional independence relationships linking any triplet $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ of subsets of $\mathbf{X}$. A graph G is a dependency map (or D-map) of M if there is a one-to-one correspondence between the random variables in $\mathbf{X}$ and the nodes $\mathbf{V}$ of G such that for all disjoint subsets $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ of $\mathbf{X}$ we have

$$\mathbf{A} \perp\!\!\!\perp_P \mathbf{B} \mid \mathbf{C} \Longrightarrow \mathbf{A} \perp\!\!\!\perp_G \mathbf{B} \mid \mathbf{C}.$$

Similarly, G is an independency map (or I-map) of M if

$$\mathbf{A} \perp\!\!\!\perp_P \mathbf{B} \mid \mathbf{C} \Longleftarrow \mathbf{A} \perp\!\!\!\perp_G \mathbf{B} \mid \mathbf{C}.$$

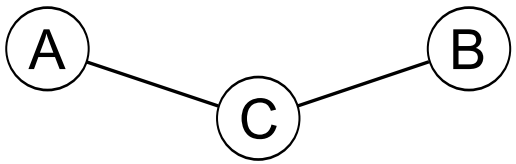
G is said to be a perfect map of M if it is both a D-map and an I-map, that is

$$\mathbf{A} \perp\!\!\!\perp_P \mathbf{B} \mid \mathbf{C} \Longleftrightarrow \mathbf{A} \perp\!\!\!\perp_G \mathbf{B} \mid \mathbf{C},$$

and in this case G is said to be faithful or isomorphic to M.

Graphical Separation in DAGs (Fundamental Connections)

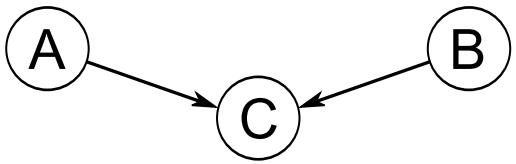
separation (undirected graphs)



$$\mathbf{A} \perp\!\!\!\perp \mathbf{B} \,|\, \mathbf{C}$$

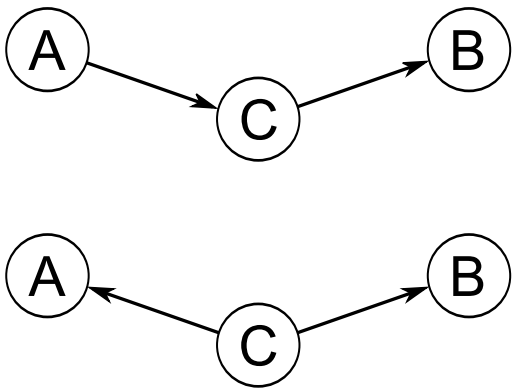
$$\mathrm{P}(\mathbf{A}, \mathbf{B}, \mathbf{C}) = \mathrm{P}(\mathbf{A} \,|\, \mathbf{C})\, \mathrm{P}(\mathbf{B} \,|\, \mathbf{C})\, \mathrm{P}(\mathbf{C})$$

d-separation (directed acyclic graphs)



$$\mathbf{A} \not\perp\!\!\!\perp \mathbf{B} \,|\, \mathbf{C}$$

$$\mathrm{P}(\mathbf{A}, \mathbf{B}, \mathbf{C}) = \mathrm{P}(\mathbf{C} \,|\, \mathbf{A}, \mathbf{B})\, \mathrm{P}(\mathbf{A})\, \mathrm{P}(\mathbf{B})$$



$$\mathbf{A} \perp\!\!\!\perp \mathbf{B} \,|\, \mathbf{C}$$

$$\begin{aligned} \mathrm{P}(\mathbf{A}, \mathbf{B}, \mathbf{C}) &= \\ &= \mathrm{P}(\mathbf{B} \,|\, \mathbf{C})\, \mathrm{P}(\mathbf{C} \,|\, \mathbf{A})\, \mathrm{P}(\mathbf{A}) \\ &= \mathrm{P}(\mathbf{A} \,|\, \mathbf{C})\, \mathrm{P}(\mathbf{B} \,|\, \mathbf{C})\, \mathrm{P}(\mathbf{C}) \end{aligned}$$

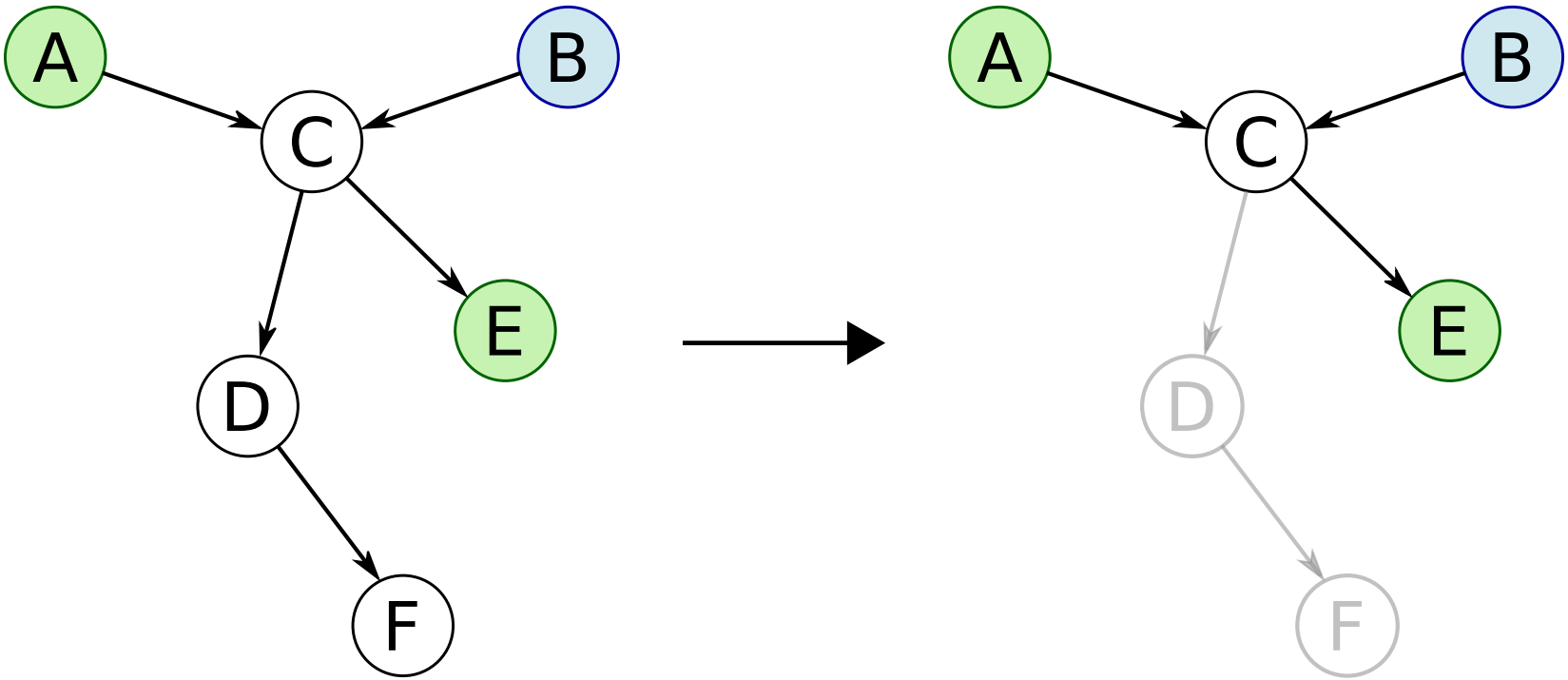
Graphical Separation in DAGs (General Case)

Now, in the general case we can extend the patterns from the fundamental connections and apply them to every possible path between **A** and **B** for a given **C**; this is how d-separation is defined.

> *If* $\mathbf{A}$, $\mathbf{B}$ *and* $\mathbf{C}$ *are three disjoint subsets of nodes in a directed acyclic graph* $\mathcal{G}$, *then* $\mathbf{C}$ *is said to d-separate* $\mathbf{A}$ *from* $\mathbf{B}$, *denoted* $\mathbf{A} \perp\!\!\!\perp_G \mathbf{B} \mid \mathbf{C}$, *if along every path between a node in* $\mathbf{A}$ *and a node in* $\mathbf{B}$ *there is a node* v *satisfying one of the following two conditions:*
>
> 1. v *has converging edges (i.e. there are two edges pointing to* v *from the adjacent nodes in the path) and none of* v *or its descendants (i.e. the nodes that can be reached from* v*) are in* $\mathbf{C}$.
> 2. v *is in* $\mathbf{C}$ *and does not have converging edges.*

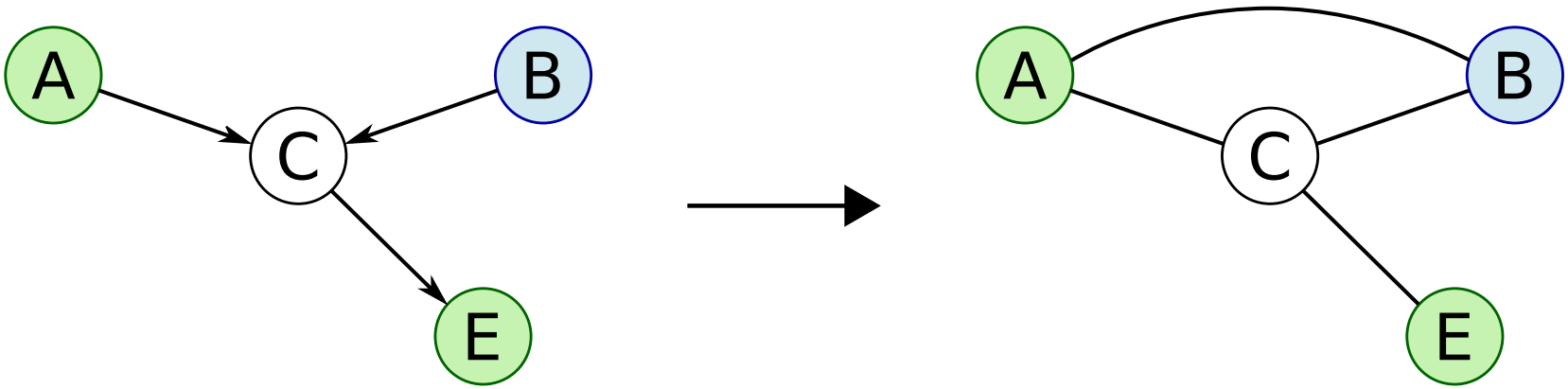
This definition clearly does not provide a computationally feasible approach to assess d-separation; but there are other ways.

A Simple Algorithm to Check D-Separation (I)



Say we want to check whether A and E are d-separated by B. First, we can drop all the nodes that are not ancestors (i.e. parents, parents' parents, etc.) of A, E and B since each node only depends on its parents.

A Simple Algorithm to Check D-Separation (II)

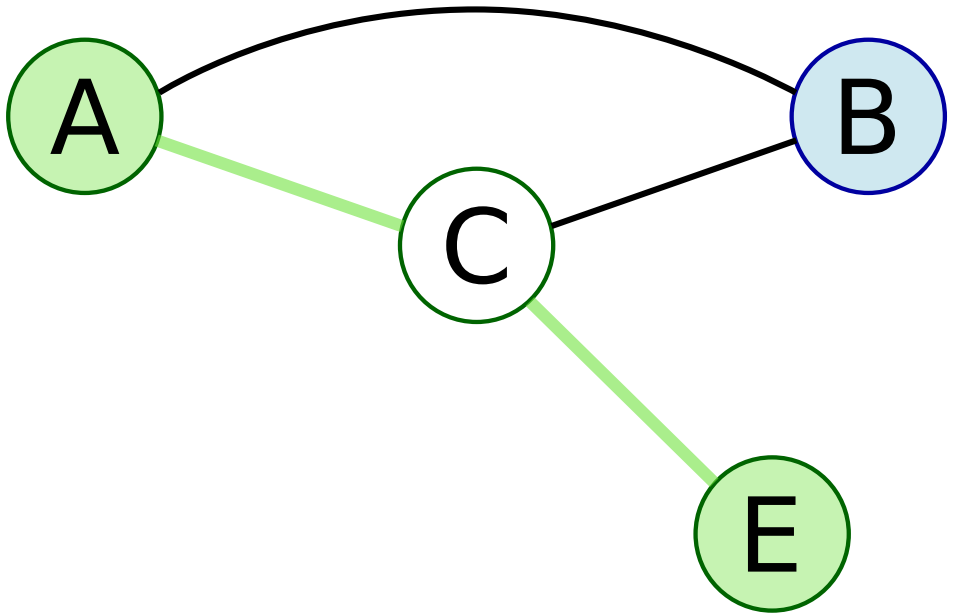


Transform the subgraph into its moral graph by

1. connecting all nodes that have one child in common; and
2. removing all arc directions to obtain an undirected graph.

This transformation has the double effect of making the dependence between parents explicit by "marrying" them and of allowing us to use the classic definition of graphical separation.

A Simple Algorithm to Check D-Separation (III)



Finally, we can just perform e.g. a depth-first or breadth-first search and see if we can find an open path between A and E, that is, a path that is not blocked by B.

The Local Markov Property (I)

If we use d-separation as our definition of graphical separation, assuming that the DAG is an I-map leads to the general formulation of the decomposition of the global distribution $\mathrm{P}(\mathbf{X})$:

$$\mathrm{P}(\mathbf{X}) = \prod_{i=1}^{N} \mathrm{P}(X_i \mid \Pi_{X_i}) \tag{1}$$

into the local distributions for the X_i given their parents Π_{X_i}. If X_i has two or more parents it depends on their joint distribution, because each pair of parents forms a convergent connection centred on X_i and we cannot establish their independence. This decomposition is preferable to that obtained from the chain rule,

$$\mathrm{P}(\mathbf{X}) = \prod_{i=1}^{N} \mathrm{P}(X_i \mid X_{i+1}, \ldots, X_N) \tag{2}$$

because the conditioning sets are typically smaller.

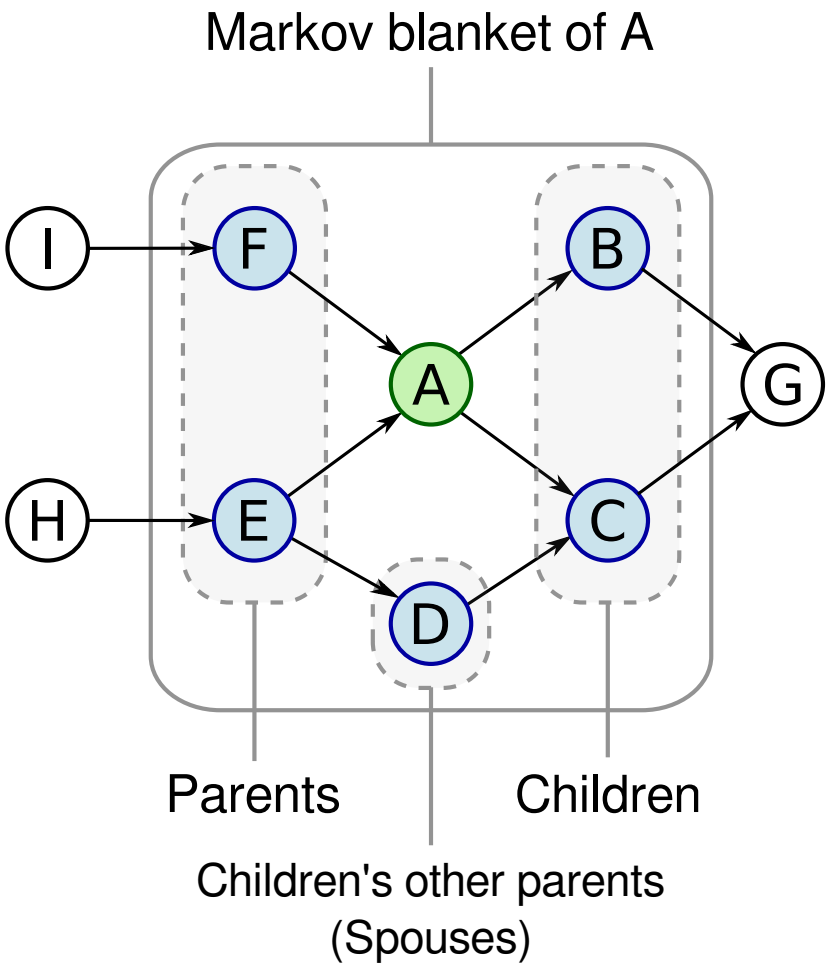
The Local Markov Property (II)

Another result along the same lines is called the local Markov property, which can be combined with the chain rule above to get the decomposition into local distributions.

> *Each node X_i is conditionally independent of its non-descendants (e.g., nodes X_j for which there is no path from X_i to X_j) given its parents.*

Compared to the previous decomposition, it highlights the fact that parents are not completely independent from their children in the BN; a trivial application of Bayes' theorem to invert the direction of the conditioning shows how information on a child can change the distribution of the parent.

Completely D-Separating: Markov Blankets



We can easily use the DAG to solve the feature selection problem. The set of nodes that graphically isolates a target node from the rest of the DAG is called its Markov blanket and includes:

- its parents;
- its children;
- other nodes sharing a child.

Since $\perp\!\!\!\perp_G$ implies $\perp\!\!\!\perp_P$, we can restrict ourselves to the Markov blanket to perform any kind of inference on the target node, and disregard the rest.

Different DAGs, Same Distribution: Topological Ordering

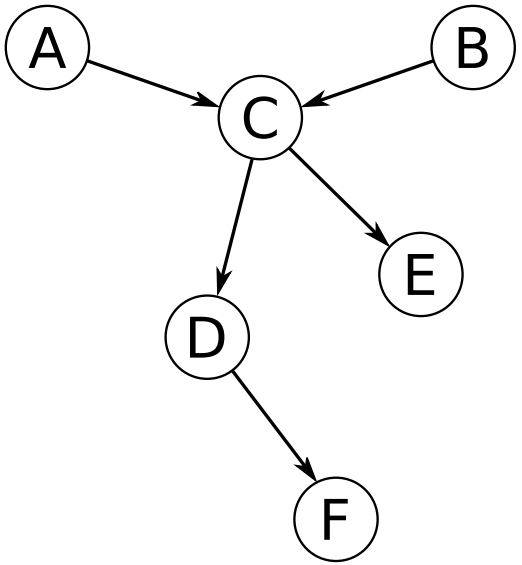
A DAG uniquely identifies a factorisation of $\mathrm{P}(\mathbf{X})$; the converse is not necessarily true. Consider again the DAG on the left:

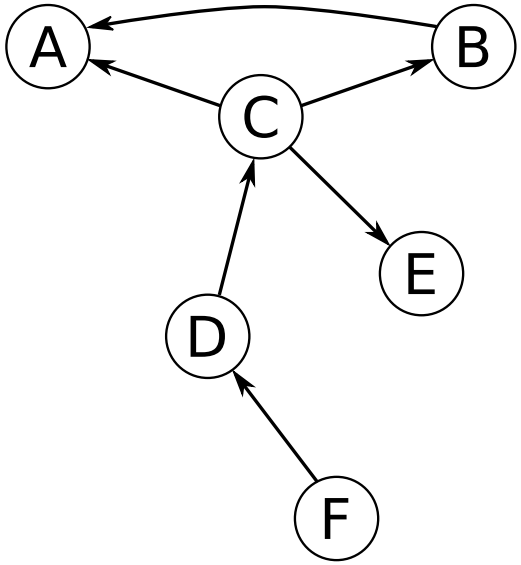
$$\mathrm{P}(\mathbf{X}) = \mathrm{P}(A)\,\mathrm{P}(B)\,\mathrm{P}(C \mid A, B)\,\mathrm{P}(D \mid C)\,\mathrm{P}(E \mid C)\,\mathrm{P}(F \mid D).$$

We can rearrange the dependencies using Bayes theorem to obtain:

$$\mathrm{P}(\mathbf{X}) = \mathrm{P}(A \mid B, C)\,\mathrm{P}(B \mid C)\,\mathrm{P}(C \mid D)\,\mathrm{P}(D \mid F)\,\mathrm{P}(E \mid C)\,\mathrm{P}(F),$$

which gives the DAG on the right, with a different topological ordering.





Different DAGs, Same Distribution: Equivalence Classes

On a smaller scale, even keeping the same underlying undirected graph we can reverse a number of arcs without changing the dependence structure of $\mathbf{X}$. Since the triplets $A \rightarrow B \rightarrow C$ and $A \leftarrow B \rightarrow C$ are probabilistically equivalent, we can reverse the directions of their arcs as we like as long as we do not create any new v-structure ($A \rightarrow B \leftarrow C$, with no arc between A and C).

This means that we can group DAGs into equivalence classes that are uniquely identified by the underlying undirected graph and the v-structures. The directions of other arcs can be either:

- uniquely identifiable because one of the directions would introduce cycles or new v-structures in the graph (compelled arcs);
- completely undetermined.

The result is a completed partially directed graph (CPDAG).

What Are V-Structures, and What Are Not

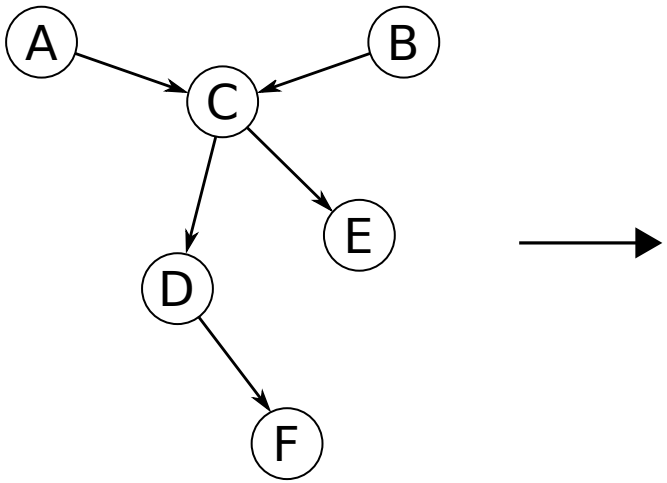
It is important to note that even though $A \to B \leftarrow C$ is a convergent connection, it is not a v-structure if A and C are connected by $A \to C$. As a result, we are no longer able to identify which nodes are the parents in the connection. For example:

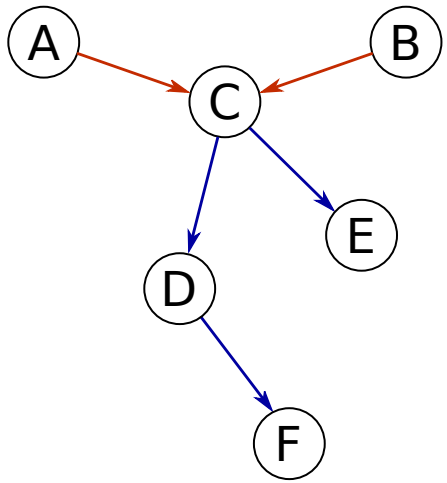
$$\underbrace{\mathrm{P}(A)\,\mathrm{P}(C \mid A)\,\mathrm{P}(B \mid A, C)}_{A \to B \leftarrow C,\, A \to C} = \mathrm{P}(A)\,\frac{\mathrm{P}(C, A)}{\mathrm{P}(A)}\,\frac{\mathrm{P}(B, A, C)}{\mathrm{P}(A, C)} =$$

$$= \mathrm{P}(A)\,\mathrm{P}(B, C \mid A) = \underbrace{\mathrm{P}(A)\,\mathrm{P}(C \mid B, A)\,\mathrm{P}(B \mid A)}_{B \to C \leftarrow A,\, A \to B}. \quad (3)$$

Therefore, the fact that the two parents in a convergent connection are not connected by an arc is crucial in the identification of the correct CPDAG.

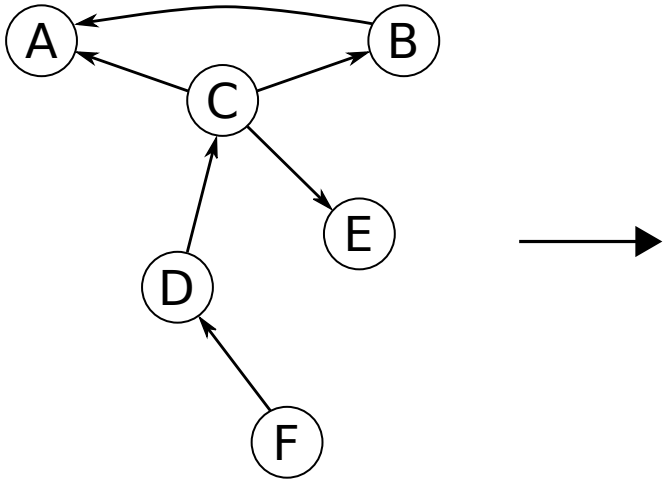
Completed Partially Directed Acyclic Graphs (CPDAGs)

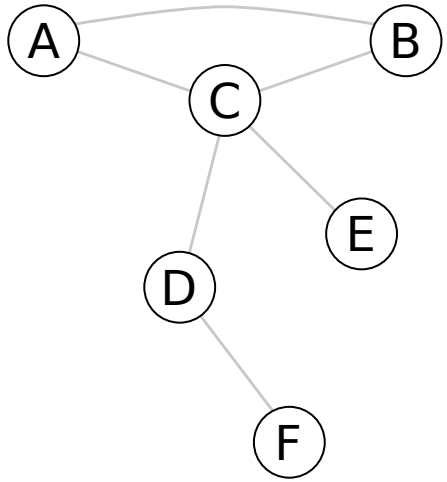




DAG

CPDAG





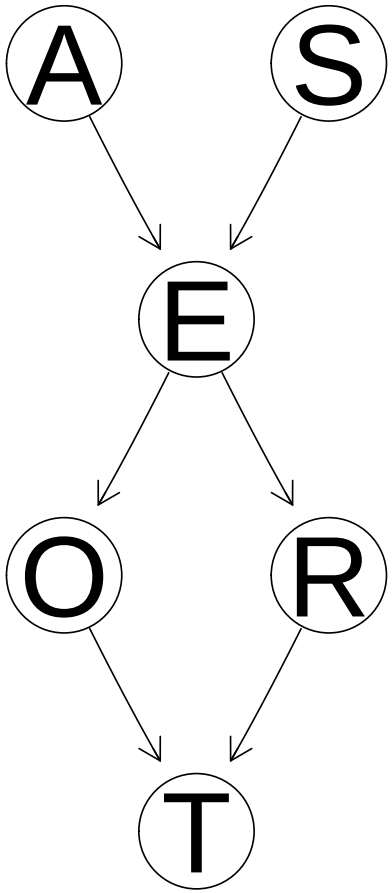
An Example: Train Use Survey

Consider a simple, hypothetical survey whose aim is to investigate the usage patterns of different means of transport, with a focus on cars and trains.

- Age (`A`): *young* for individuals below 30 years old, *adult* for individuals between 30 and 60 years old, and *old* for people older than 60.
- Sex (`S`): *male* or *female*.
- Education (`E`): *up to high school* or *university degree*.
- Occupation (`O`): *employee* or *self-employed*.
- Residence (`R`): the size of the city the individual lives in, recorded as either *small* or *big*.
- Travel (`T`): the means of transport favoured by the individual, recorded either as *car*, *train* or *other*.

The nature of the variables recorded in the survey suggests how they may be related with each other.

The Train Use Survey as a BN (v1)

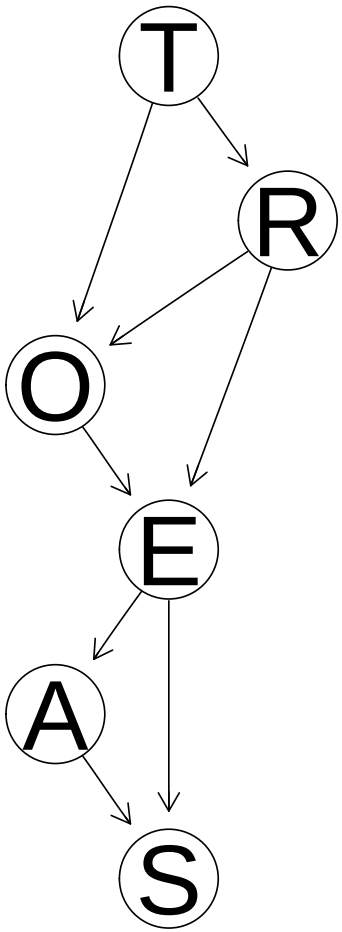


That is a prognostic view of the survey as a BN:

1. the blocks in the experimental design on top (e.g. stuff from the registry office);
2. the variables of interest in the middle (e.g. socio-economic indicators);
3. the object of the survey at the bottom (e.g. means of transport).

Variables that can be thought as "causes" are on above variables that can be considered their "effect", and confounders are on above everything else.

The Train Use Survey as a BN (v2)



That is a diagnostic view of the survey as a BN: it encodes the same dependence relationships as the prognostic view but is laid out to have "effects" on top and "causes" at the bottom.

Depending on the phenomenon and the goals of the survey, one may have a graph that makes more sense than the other; but they are equivalent for any subsequent inference. For discrete BNs, one representation may have fewer parameters than the other.

bnlearn: Creating Graphs (I)

- Setting individual arcs.

```
survey.dag = empty.graph(nodes = c("A", "S", "E", "O", "R", "T"))
survey.dag = set.arc(survey.dag, from = "A", to = "E")
survey.dag = set.arc(survey.dag, from = "S", to = "E")
survey.dag = set.arc(survey.dag, from = "E", to = "O")
survey.dag = set.arc(survey.dag, from = "E", to = "R")
survey.dag = set.arc(survey.dag, from = "O", to = "T")
survey.dag = set.arc(survey.dag, from = "R", to = "T")
```

- Setting the whole arc set at once.

```
arc.set = matrix(c("A", "E",
                   "S", "E",
                   "E", "O",
                   "E", "R",
                   "O", "T",
                   "R", "T"),
            byrow = TRUE, ncol = 2,
            dimnames = list(NULL, c("from", "to")))
arcs(survey.dag) = arc.set
```

bnlearn: Creating Graphs (II)

- Using the adjacency matrix representation of the arc set.

```
amat(survey.dag) =
  matrix(c(0L, 0L, 1L, 0L, 0L, 0L,
           0L, 0L, 1L, 0L, 0L, 0L,
           0L, 0L, 0L, 1L, 1L, 0L,
           0L, 0L, 0L, 0L, 0L, 1L,
           0L, 0L, 0L, 0L, 0L, 1L,
           0L, 0L, 0L, 0L, 0L, 0L),
    byrow = TRUE, nrow = 6, ncol = 6,
    dimnames = list(nodes(survey.dag), nodes(survey.dag)))
```

- Using the formula representation for the Bayesian network.

```
survey.dag = model2network("[A][S][E|A:S][O|E][R|E][T|O:R]")
```

Acyclicity is enforced by all there functions by default, e.g.

```
set.arc(survey.dag, from = "T", to = "E")
## Error in arc.operations(x = x, from = from, to = to, op = "set",
check.cycles = check.cycles, :  the resulting graph contains cycles.
```

bnlearn: BN graph objects

```
survey.dag
```

```
##
##   Random/Generated Bayesian network
##
##   model:
##    [A][S][E|A:S][O|E][R|E][T|O:R]
##   nodes:                                 6
##   arcs:                                  6
##     undirected arcs:                     0
##     directed arcs:                       6
##   average markov blanket size:           2.67
##   average neighbourhood size:            2.00
##   average branching factor:              1.00
##
##   generation algorithm:                  Empty
```

This is what the graph structure of BN looks like when printed: note the model formula, which is the same as that you would pass to `model2network()`. Additional information will be printed as well if the graph is learned from data.

bnlearn: Manipulating Graphs

- Adding, removing and reversing arcs.

```
survey.dag = set.arc(survey.dag, from = "A", to = "O")
survey.dag = drop.arc(survey.dag, from = "E", to = "O")
survey.dag = reverse.arc(survey.dag, from = "R", to = "E")
```

- Finding the skeleton (the underlying undirected graph).

```
skeleton(survey.dag)
```

- Finding the moral graph.

```
moral(survey.dag)
```

- Extracting a subgraph.

```
subgraph(survey.dag)
```

Plus many others...

bnlearn: Investigating Graphs (I)

- Sets of nodes close to a target node (here E).

```
mb(survey.dag, "E")
## [1] "A" "O" "R" "S"
nbr(survey.dag, "E")
## [1] "A" "O" "R" "S"
parents(survey.dag, "E")
## [1] "A" "S"
children(survey.dag, "E")
## [1] "O" "R"
```

- Roots (no parents) and leaves (no children).

```
root.nodes(survey.dag)
## [1] "A" "S"
leaf.nodes(survey.dag)
## [1] "T"
```

bnlearn: Investigating Graphs (II)

- Directed and undirected arcs.

```
directed.arcs(survey.dag)
```

```
##      from to
## [1,] "A"  "E"
## [2,] "S"  "E"
## [3,] "E"  "O"
## [4,] "E"  "R"
## [5,] "O"  "T"
## [6,] "R"  "T"
```

```
undirected.arcs(survey.dag)
```

```
##      from to
```

- Different graph representations.

```
arcs(survey.dag)
amat(survey.dag)
```

- Looking for paths.

```
path(survey.dag, from = "A", to = "T")
```

```
## [1] TRUE
```

bnlearn: D-Separation and Markov Blankets

The `dsep()` and `mb()` functions can be used to show how d-separation and Markov blankets interact in practice. Firstly, node that a node is never part of its own Markov blanket.

```
mbE = mb(survey.dag, "E")
"E" %in% mbE
```

```
## [1] FALSE
```

Secondly, note that the Markov blanket is minimal and that it makes all other nodes independent of the target node.

```
for (node in mbE)
  print(dsep(survey.dag, "E", node, setdiff(mbE, c("E", node))))
```

```
## [1] FALSE
## [1] FALSE
## [1] FALSE
## [1] FALSE
```

```
for (node in setdiff(nodes(survey.dag), c("E", mbE)))
  print(dsep(survey.dag, "E", node, mbE))
```

```
## [1] TRUE
```

bnlearn: Moral Graphs and CPDAGs

There are functions to compute them:

```
moral(survey.dag)
```

```
cpdag(survey.dag)
```

And if we go back to the survey example, we find that all arcs are compelled and that the CPDAG is identical to the original DAG.

```
all.equal(cpdag(survey.dag), survey.dag)
## [1] TRUE
compelled.arcs(survey.dag)
##      from to
## [1,] "A"  "E"
## [2,] "E"  "O"
## [3,] "E"  "R"
## [4,] "O"  "T"
## [5,] "R"  "T"
## [6,] "S"  "E"
```

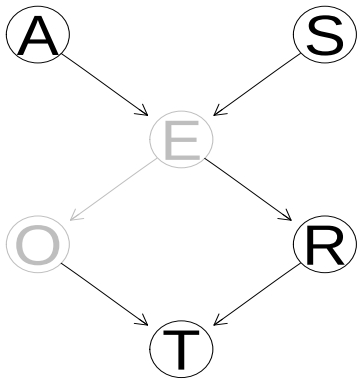
And we can observe that:

```
all.equal(compelled.arcs(survey.dag), directed.arcs(cpdag(survey.dag)))
## [1] TRUE
```

bnlearn: Plotting Graphs

bnlearn uses the functionality implemented in the **Rgraphviz** package to plot graphs, through the `graphviz.plot` function.

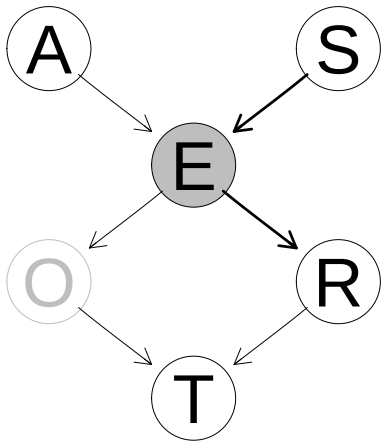
```
hlight = list(nodes = c("E", "O"),
              arcs = c("E", "O"),
              col = "grey",
              textCol = "grey")
pp = graphviz.plot(survey.dag,
       highlight = hlight)
```




```
edgeRenderInfo(pp) =
  list(col = c("S~E" = "black",
               "E~R" = "black"),
       lwd = c("S~E" = 3, "E~R" = 3))
```

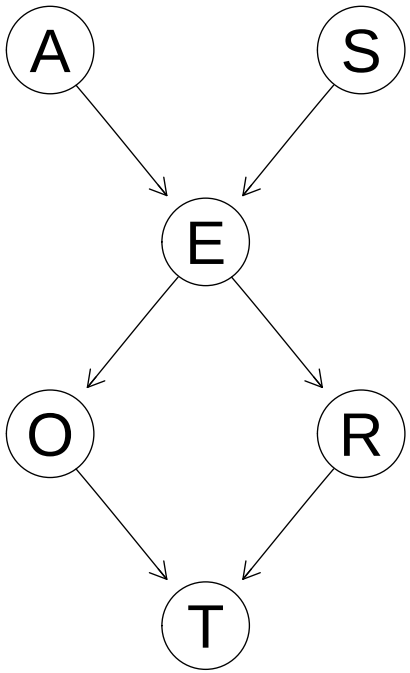
```
nodeRenderInfo(pp) =
  list(col =
    c("S" = "black", "E" = "black",
      "R" = "black"),
       textCol =
    c("S" = "black", "E" = "black",
      "R" = "black"),
       fill = c("E" = "grey"))
```

```
renderGraph(pp)
```

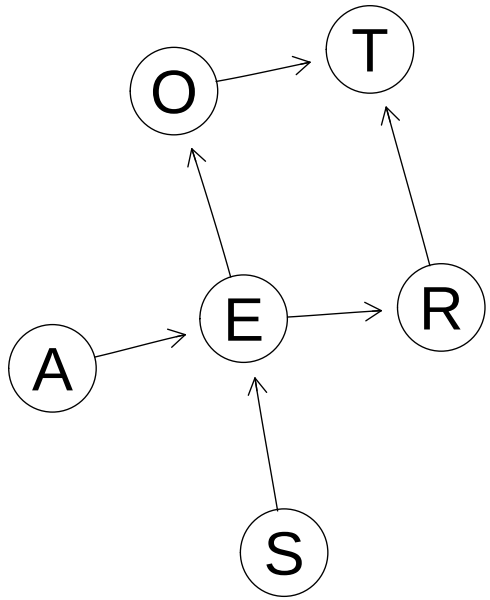


Different Layouts Available in Rgraphviz

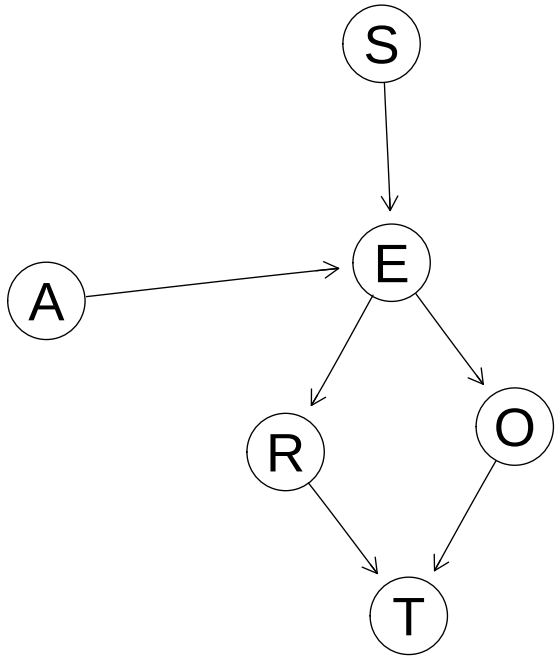
layout = "dot"



layout = "fdp"

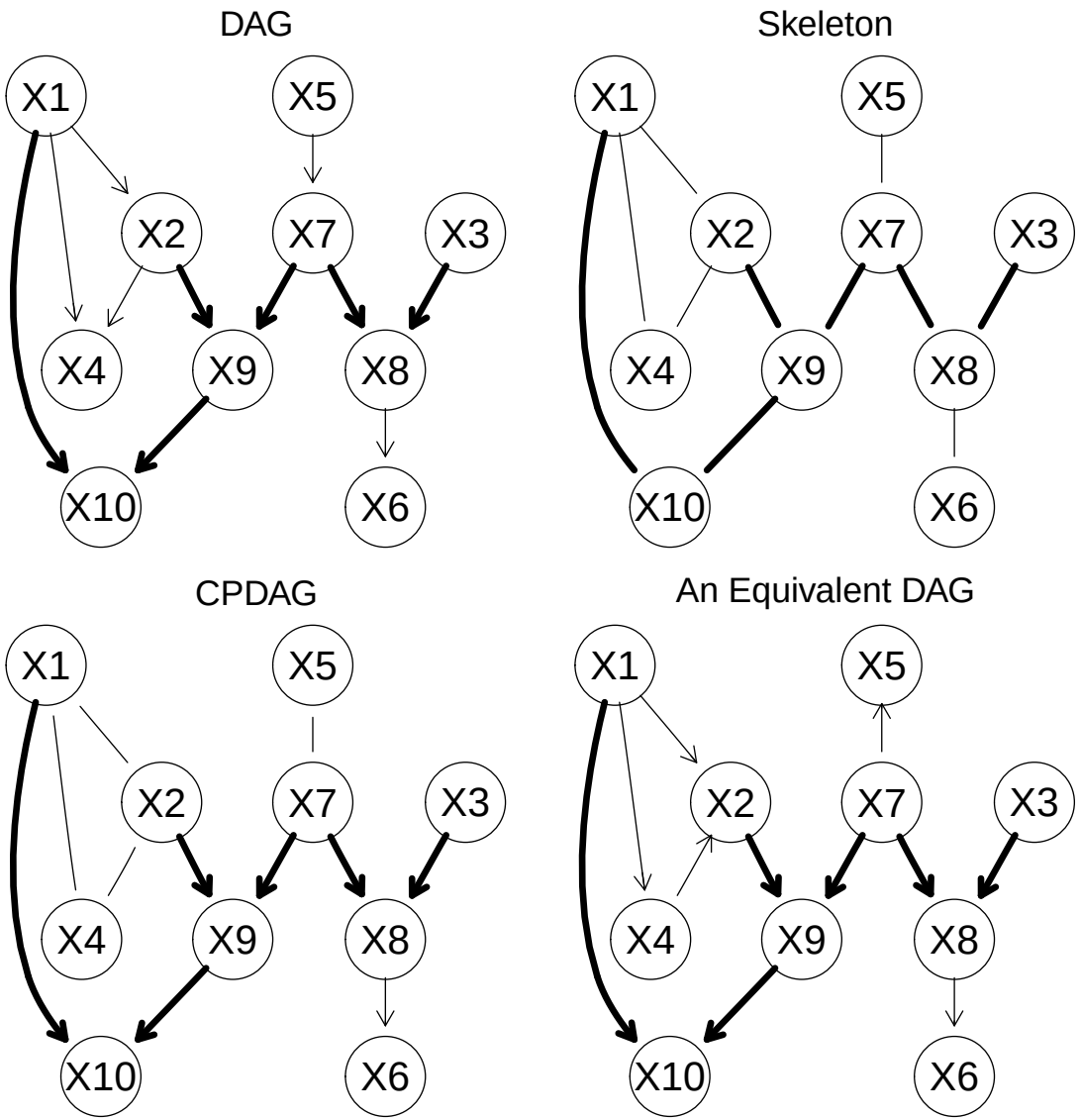


layout = "circo"

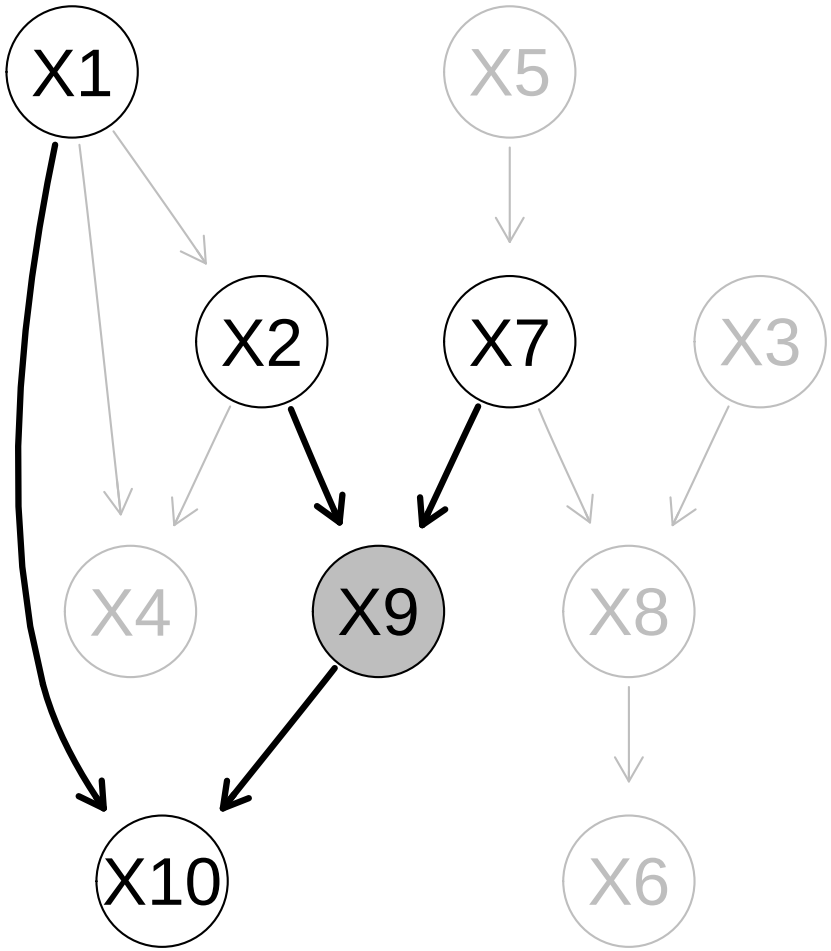


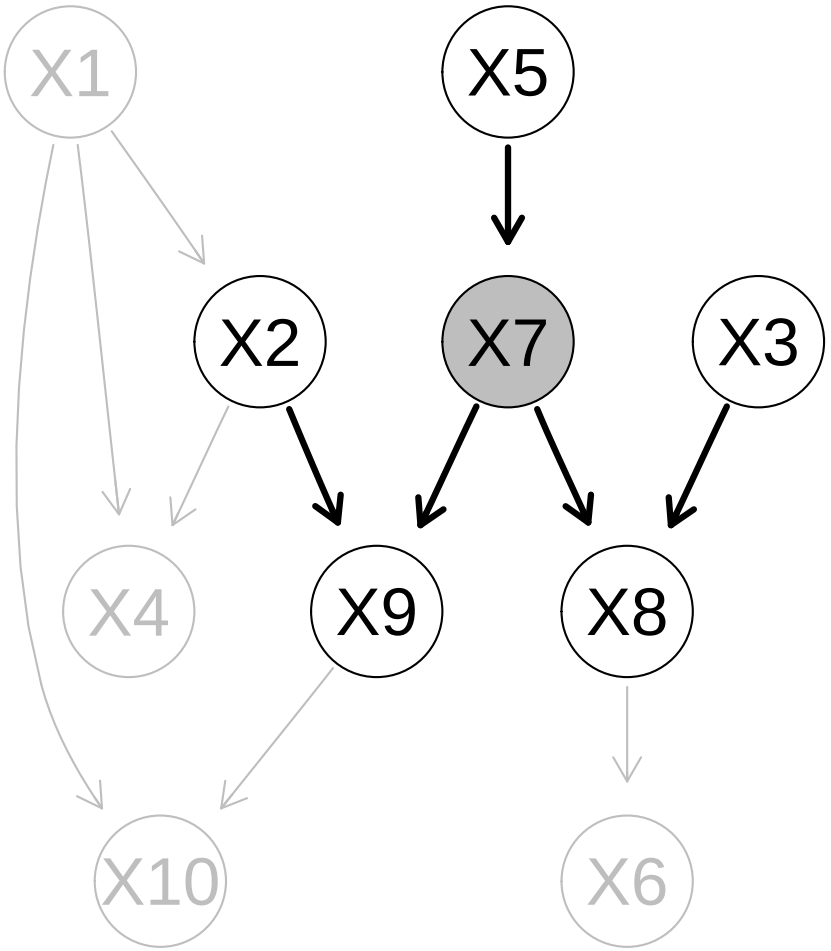
NOTE: unlike **igraph** we cannot rearrange the layout of the nodes, which makes plotting graphs with the same node positions but different arcs very difficult.

Another Example, from the C&H Book (I)



Another Example, from the C&H Book (II)





Another Example, from the C&H Book (III)

We can verify again that the Markov blanket contains the children, the parents and the spouses of the node it is centred on; and that it does not contain that node.

```
M = paste("[X1][X3][X5][X6|X8][X2|X1][X7|X5][X4|X1:X2]",
      "[X8|X3:X7][X9|X2:X7][X10|X1:X9]", sep = "")
dag = model2network(M)
mb(dag, node = "X9")
```

```
## [1] "X1"  "X10" "X2"  "X7"
```

```
par.X9 = parents(dag, node = "X9")
ch.X9 = children(dag, node = "X9")
sp.X9 = sapply(ch.X9, parents, x = dag)
sp.X9 = sp.X9[sp.X9 != "X9"]
unique(c(par.X9, ch.X9, sp.X9))
```

```
## [1] "X2"  "X7"  "X10" "X1"
```

Another Example, from the C&H Book (IV)

We can also check that Markov blankets are symmetric: if A is in the Markov blanket of B, then B is in the Markov blanket of A.

```
sapply(nodes(dag), function(node) node %in% mb(dag, node = "X9"))
```

```
##    X1   X10    X2    X3    X4    X5    X6    X7    X8    X9
##  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
sapply(nodes(dag), function(node) "X9" %in% mb(dag, node = node))
```

```
##    X1   X10    X2    X3    X4    X5    X6    X7    X8    X9
##  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

This is a consequence of the fact that if A is a parent of B, then B is a child of A; and if A is a spouse of B, then B is a spouse of A.

What About the Probability Distributions?

The second component of a BN is the probability distribution $\mathrm{P}(\mathbf{X})$. The choice should such that the BN:

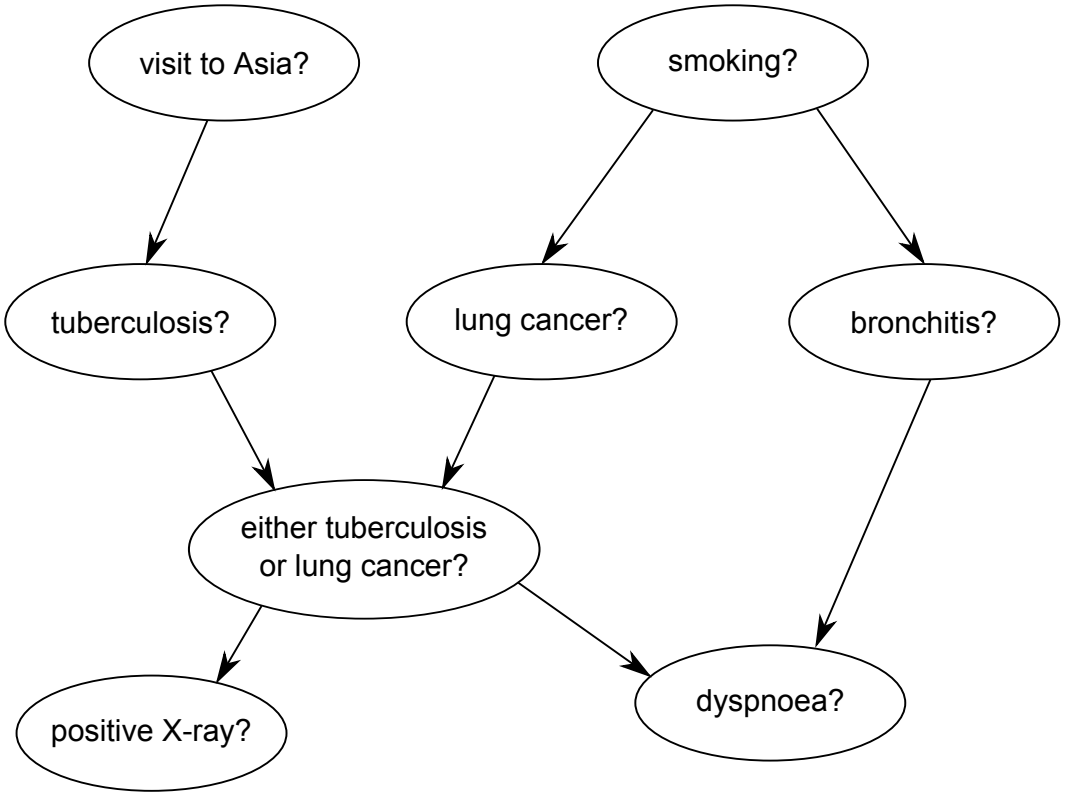
- can be learned efficiently from data;
- is flexible (distributional assumptions should not be too strict);
- is easy to query to perform inference.

The three most common choices in the literature (by far), are:

- discrete BNs (DBNs), in which $\mathbf{X}$ and the $X_i \mid \Pi_{X_i}$ are multinomial;
- Gaussian BNs (GBNs), in which $\mathbf{X}$ is multivariate normal and the $X_i \mid \Pi_{X_i}$ are univariate normal;
- conditional linear Gaussian BNs (CLGBNs), in which $\mathbf{X}$ is a mixture of multivariate normals and the $X_i \mid \Pi_{X_i}$ are either multinomial, univariate normal or mixtures of normals.

It has been proved in the literature that exact inference is possible in these three cases, hence their popularity.

Discrete BNs

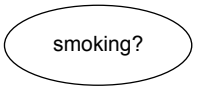


A classic example of DBN is the ASIA network from Lauritzen & Spiegelhalter (1988), which includes a collection of binary variables. It describes a simple diagnostic problem for tuberculosis and lung cancer.

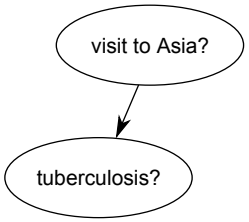
Total parameters of $\mathbf{X}$:
$2^8 - 1 = 255$

Conditional Probability Tables (CPTs)

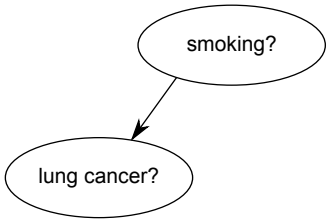
yes	no
0.01	0.99



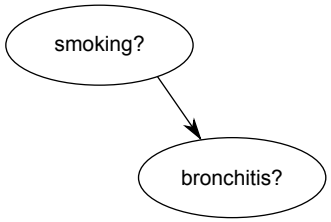
yes	no
0.50	0.50



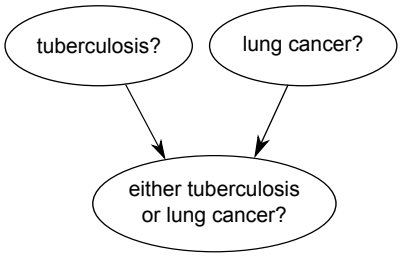
	yes	no
yes	0.05	0.95
no	0.01	0.99



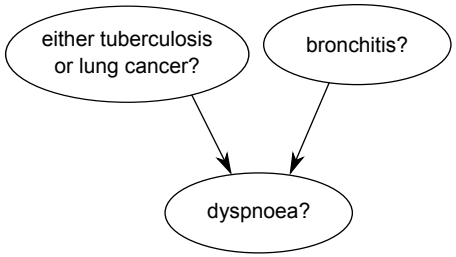
	yes	no
yes	0.10	0.90
no	0.01	0.99



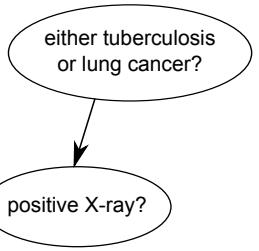
	yes	no
yes	0.60	0.40
no	0.30	0.70



	yes	no
yes:yes	1	0
yes:no	1	0
no:yes	1	0
no:no	0	1



	yes	no
yes:yes	0.90	0.10
yes:no	0.70	0.30
no:yes	0.80	0.20
no:no	0.10	0.90



	yes	no
yes	0.98	0.02
no	0.05	0.95

The local distributions $X_i \mid \Pi_{X_i}$ take the form of conditional probability tables for each node given all the configurations of the values of its parents.

Overall parameters of the $X_i \mid \Pi_{X_i}$: 18

bnlearn: Creating a Discrete BN (ASIA)

```
asia.dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")

lv = c("yes", "no")

A.prob = array(c(0.01, 0.99), dim = 2, dimnames = list(A = lv))
S.prob = array(c(0.01, 0.99), dim = 2, dimnames = list(A = lv))
T.prob = array(c(0.05, 0.95, 0.01, 0.99), dim = c(2, 2),
          dimnames = list(T = lv, A = lv))
L.prob = array(c(0.1, 0.9, 0.01, 0.99), dim = c(2, 2),
         dimnames = list(L = lv, S = lv))
B.prob = array(c(0.6, 0.4, 0.3, 0.7), dim = c(2, 2),
         dimnames = list(B = lv, S = lv))
D.prob = array(c(0.9, 0.1, 0.7, 0.3, 0.8, 0.2, 0.1, 0.9), dim = c(2, 2, 2),
         dimnames = list(D = lv, B = lv, E = lv))
E.prob = array(c(1, 0, 1, 0, 1, 0, 0, 1), dim = c(2, 2, 2),
         dimnames = list(E = lv, T = lv, L = lv))
X.prob = array(c(0.98, 0.02, 0.05, 0.95), dim = c(2, 2),
         dimnames = list(X = lv, E = lv))

cpt = list(A = A.prob, S = S.prob, T = T.prob, L = L.prob, B = B.prob,
        D = D.prob, E = E.prob, X = X.prob)
bn = custom.fit(asia.dag, cpt)
```

bnlearn: Conditional Probability Tables (I)

```
bn$D
```

```
##
##   Parameters of node D (multinomial distribution)
##
## Conditional probability table:
##
## , , E = yes
##
##       B
## D      yes  no
##   yes 0.9 0.7
##   no  0.1 0.3
##
## , , E = no
##
##       B
## D      yes  no
##   yes 0.8 0.1
##   no  0.2 0.9
```

bnlearn: Creating a Discrete BN (Survey)

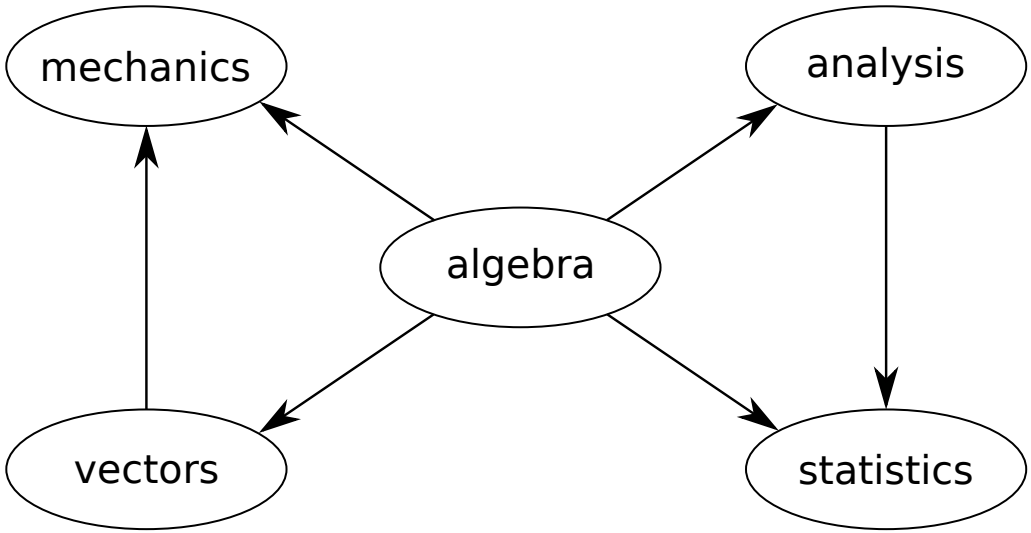
```
A.lv = c("young", "adult", "old")
S.lv = c("M", "F")
E.lv = c("high", "uni")
O.lv = c("emp", "self")
R.lv = c("small", "big")
T.lv = c("car", "train", "other")
A.prob = array(c(0.30, 0.50, 0.20), dim = 3, dimnames = list(A = A.lv))
S.prob = array(c(0.60, 0.40), dim = 2, dimnames = list(S = S.lv))
O.prob = array(c(0.96, 0.04, 0.92, 0.08), dim = c(2, 2),
               dimnames = list(O = O.lv, E = E.lv))
R.prob = array(c(0.25, 0.75, 0.20, 0.80), dim = c(2, 2),
               dimnames = list(R = R.lv, E = E.lv))
E.prob = array(c(0.75, 0.25, 0.72, 0.28, 0.88, 0.12, 0.64,
               0.36, 0.70, 0.30, 0.90, 0.10), dim = c(2, 3, 2),
               dimnames = list(E = E.lv, A = A.lv, S = S.lv))
T.prob = array(c(0.48, 0.42, 0.10, 0.56, 0.36, 0.08, 0.58,
               0.24, 0.18, 0.70, 0.21, 0.09), dim = c(3, 2, 2),
               dimnames = list(T = T.lv, O = O.lv, R = R.lv))

cpt = list(A = A.prob, S = S.prob, E = E.prob, O = O.prob,
           R = R.prob, T = T.prob)
bn = custom.fit(survey.dag, cpt)
```

bnlearn: Conditional Probability Tables (II)

```
bn$T
```

```
##
##   Parameters of node T (multinomial distribution)
##
## Conditional probability table:
##
## , , R = small
##
##         O
## T         emp self
##   car   0.48 0.56
##   train 0.42 0.36
##   other 0.10 0.08
##
## , , R = big
##
##         O
## T         emp self
##   car   0.58 0.70
##   train 0.24 0.21
##   other 0.18 0.09
```

Gaussian BNs



A classic example of GBN is the MARKS networks from Mardia, Kent & Bibby (1979), which describes the relationships between the marks on 5 math-related topics.

Assuming $X \sim N(\boldsymbol{\mu}, \Sigma)$, we can compute $\Omega = \Sigma^{-1}$. Then $\Omega_{ij} = 0$ implies $X_i \perp\!\!\!\perp_P X_j \mid \mathbf{X} \setminus \{X_i, X_j\}$. The absence of an arc $X_i \to X_j$ in the DAG implies $X_i \perp\!\!\!\perp_G X_j \mid \mathbf{X} \setminus \{X_i, X_j\}$, which in turn implies $X_i \perp\!\!\!\perp_P X_j \mid \mathbf{X} \setminus \{X_i, X_j\}$.

Total parameters of $\mathbf{X}$: $5 + 15 = 20$

Partial Correlations and Linear Regressions

The local distributions $X_i \mid \Pi_{X_i}$ take the form of linear regression models with the Π_{X_i} acting as regressors and with independent error terms.

$$\begin{aligned}
\mathsf{ALG} &= 50.60 + \varepsilon_{\mathsf{ALG}} \sim N(0, 112.8) \\
\mathsf{ANL} &= -3.57 + 0.99\mathsf{ALG} + \varepsilon_{\mathsf{ANL}} \sim N(0, 110.25) \\
\mathsf{MECH} &= -12.36 + 0.54\mathsf{ALG} + 0.46\mathsf{VECT} + \varepsilon_{\mathsf{MECH}} \sim N(0, 195.2) \\
\mathsf{STAT} &= -11.19 + 0.76\mathsf{ALG} + 0.31\mathsf{ANL} + \varepsilon_{\mathsf{STAT}} \sim N(0, 158.8) \\
\mathsf{VECT} &= 12.41 + 0.75\mathsf{ALG} + \varepsilon_{\mathsf{VECT}} \sim N(0, 109.8)
\end{aligned}$$

(That is because $\Omega_{ij} \propto \beta_j$ for X_i, so $\beta_j > 0$ if and only if $\Omega_{ij} > 0$. Also $\Omega_{ij} \propto \rho_{ij}$, the partial correlation between X_i and X_j, so we are implicitly assuming all probabilistic dependencies are linear.)

Overall parameters of the $X_i \mid \Pi_{X_i}$: $11 + 5 = 16$

bnlearn: Creating a Gaussian BN

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

ALG.dist = list(coef = c("(Intercept)" = 50.60), sd = 10.62)
ANL.dist = list(coef = c("(Intercept)" = -3.57, ALG = 0.99), sd = 10.5)
MECH.dist =
  list(coef = c("(Intercept)" = -12.36, ALG = 0.54, VECT = 0.46), sd = 13.97)
STAT.dist =
  list(coef = c("(Intercept)" = -11.19, ALG = 0.76, ANL = 0.31), sd = 12.61)
VECT.dist = list(coef = c("(Intercept)" = 12.41, ALG = 0.75), sd = 10.48)

ldist = list(ALG = ALG.dist, ANL = ANL.dist, MECH = MECH.dist,
             STAT = STAT.dist, VECT = VECT.dist)
bn = custom.fit(marks.dag, ldist)
```

Note that we specify the regression coefficients and the standard deviation of the residuals in keeping with the parameterisation used by R.

bnlearn: Local Linear Regressions

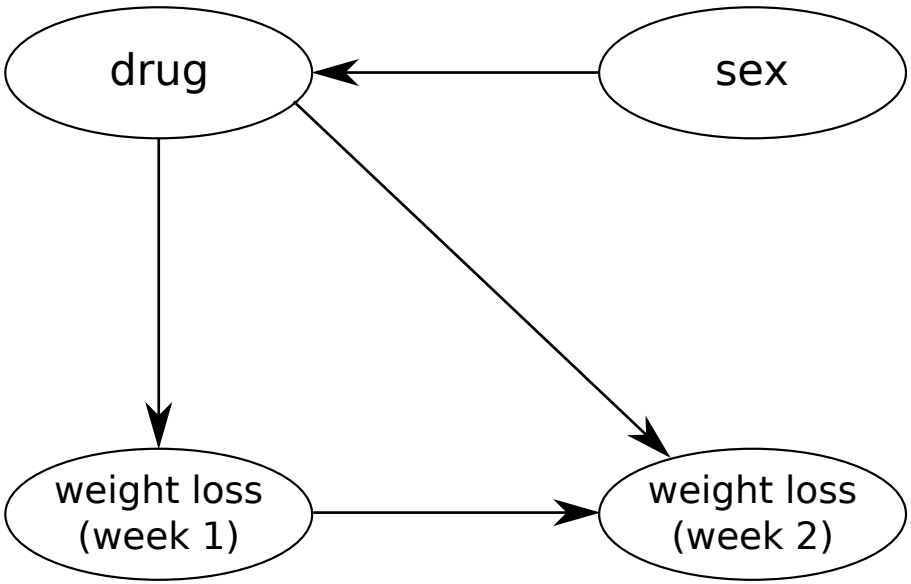
```
bn[c("MECH", "STAT")]
```

```
## $MECH
##
##   Parameters of node MECH (Gaussian distribution)
##
## Conditional density: MECH | ALG + VECT
## Coefficients:
## (Intercept)          ALG         VECT
##      -12.36         0.54         0.46
## Standard deviation of the residuals: 14
##
## $STAT
##
##   Parameters of node STAT (Gaussian distribution)
##
## Conditional density: STAT | ALG + ANL
## Coefficients:
## (Intercept)          ALG          ANL
##      -11.19         0.76         0.31
## Standard deviation of the residuals: 12.6
```

Conditional Linear Gaussian BNs

CLGBNs contain both discrete and continuous nodes, and combine DBNs and GBNs as follows to obtain a mixture-of-Gaussians network:

- continuous nodes cannot be parents of discrete nodes;
- the local distribution of each discrete node is a CPT;
- the local distribution of each continuous node is a set of linear regression models, one for each configurations of the discrete parents, with the continuous parents acting as regressors.



One of the classic examples is the RATS' WEIGHTS network from Edwards (1995), which describes weight loss in a drug trial performed on rats.

Mixtures of Linear Regressions

The resulting local distribution for the first weight loss for drugs D_1, D_2 and D_3 is:

$$\begin{aligned} W_{1,D_1} &= 7 + \varepsilon_{D_1} \sim N(0, 2.5) \\ W_{1,D_2} &= 7.50 + \varepsilon_{D_2} \sim N(0, 2) \\ W_{1,D_3} &= 14.75 + \varepsilon_{D_3} \sim N(0, 11) \end{aligned}$$

with just the intercepts since the node has no continuous parents. The local distribution for the second loss is:

$$\begin{aligned} W_{2,D_1} &= 1.02 + 0.89\beta_{W_1} + \varepsilon_{D_1} \sim N(0, 3.2) \\ W_{2,D_2} &= -1.68 + 1.35\beta_{W_1} + \varepsilon_{D_2} \sim N(0, 4) \\ W_{2,D_3} &= -1.83 + 0.82\beta_{W_1} + \varepsilon_{D_3} \sim N(0, 1.9) \end{aligned}$$

Overall, they look like random effect models with random intercepts and random slopes.

bnlearn: Creating a Conditional Linear Gaussian BN

```
rats.dag = model2network("[SEX][DRUG|SEX][WL1|DRUG][WL2|WL1:DRUG]")
SEX.lv = c("M", "F")
DRUG.lv = c("D1", "D2", "D3")

SEX.prob = array(c(0.5, 0.5), dim = 2, dimnames = list(SEX = SEX.lv))
DRUG.prob = array(c(0.3333, 0.3333, 0.3333, 0.3333, 0.3333, 0.3333),
                  dim = c(3, 2), dimnames = list(DRUG = DRUG.lv, SEX = SEX.lv))
WL1.coef = matrix(c(7, 7.50, 14.75), nrow = 1, ncol = 3,
                dimnames = list("(Intercept)", NULL))
WL1.dist = list(coef = WL1.coef, sd = c(1.58, 0.447, 3.31))
WL2.coef = matrix(c(1.02, 0.89, -1.68, 1.35, -1.83, 0.82), nrow = 2, ncol = 3,
                dimnames = list(c("(Intercept)", "WL1")))
WL2.dist = list(coef = WL2.coef, sd = c(1.78, 2, 1.37))

ldist = list(SEX = SEX.prob, DRUG = DRUG.prob, WL1 = WL1.dist, WL2 = WL2.dist)
bn = custom.fit(rats.dag, ldist)
```

The regression coefficients are stored in a matrix with one conditional regression in each column, so that each column corresponds to one configuration of the discrete parents and each row to one of the continuous parents.

bnlearn: Mixtures of Linear Regressions

```
bn$WL2

##
##   Parameters of node WL2 (conditional Gaussian distribution)
##
## Conditional density: WL2 | DRUG + WL1
## Coefficients:
##                   0      1      2
## (Intercept)    1.02  -1.68  -1.83
## WL1            0.89   1.35   0.82
## Standard deviation of the residuals:
##    0     1     2
## 1.78  2.00  1.37
## Discrete parents' configurations:
##    DRUG
## 0    D1
## 1    D2
## 2    D3
```

Limitations of These Probability Distributions

- No real-world, multivariate data set follows a multivariate Gaussian distribution; even if the marginal distributions are normal, not all dependence relationships are linear.
- Computing partial correlations is problematic in most large data sets (and in a lot of small ones, too) because of singularities.
- Parametric assumptions for mixed data have strong limitations, as they impose constraints on which arcs may be present in the graph (e.g. a continuous node cannot be the parent of a discrete node).
- Discretisation is a common solution to the above problems, but it may discard useful information and it is tricky to get right (i.e. choosing a set of intervals such that the dependence relationships involving the original variable are preserved). On the other hand, dependencies are no longer required to be linear.
- Ordinal variables are treated as categorical, again losing information.

Equivalence and Singularity

Assuming the DAG is an I-map means that serial and divergent connections result in equivalent factorisations of the variables involved. It is easy to show that

$$\underbrace{\mathrm{P}(X_i)\,\mathrm{P}(X_j \mid X_i)\,\mathrm{P}(X_k \mid X_j)}_{\text{serial connection}} = \mathrm{P}(X_j, X_i)\,\mathrm{P}(X_k \mid X_j) =$$

$$= \underbrace{\mathrm{P}(X_i \mid X_j)\,\mathrm{P}(X_j)\,\mathrm{P}(X_k \mid X_j)}_{\text{divergent connection}}.$$

Then $X_i \rightarrow X_j \rightarrow X_k$ and $X_i \leftarrow X_j \rightarrow X_k$ are equivalent. This is true, however, only if the global distribution is positive everywhere because it may not be possible to reverse the direction of the conditioning:

$$\mathrm{P}(X_i \mid X_j) \neq \frac{\mathrm{P}(X_i, X_j)}{\mathrm{P}(X_j)} \qquad \text{if} \qquad \mathrm{P}(X_j) = 0.$$

Summary

- Bayesian networks are a combination of a DAG and a global distribution, both defined on the same variables.
- Bayesian networks provide a systematic decomposition of the global distribution into lower-dimensional local distributions, in a divide-and-conquer way.
- Bayesian network provide a principled solution to the problem of feature selection using Markov blankets.
- Three distributional assumptions are common: discrete, Gaussian, and conditional linear Gaussian.

Fundamentals of Inference

Events, Evidence and Queries

Probabilistic reasoning on BNs works in the framework of Bayesian statistics and focuses on the computation of posterior probabilities or densities.

For example, suppose we have learned a BN $\mathcal{B}$ with DAG G and parameters Θ. We want to use $\mathcal{B}$ to investigate the effects of a new piece of evidence $\mathbf{E}$ using the knowledge encoded in $\mathcal{B}$, that is, to investigate the posterior distribution

$$\mathrm{P}(\mathbf{X} \mid \mathbf{E}, \mathcal{B}) = \mathrm{P}(\mathbf{X} \mid \mathbf{E}, G, \Theta).$$

Questions that can be asked are called queries and are typically an event of interest. The two most common queries are conditional probability (CPQ) and maximum a posteriori (MAP) queries, also known as most probable explanation (MPE) queries.

Types of Evidence

- Hard evidence: an instantiation of one or more variables in the BN. In other words,

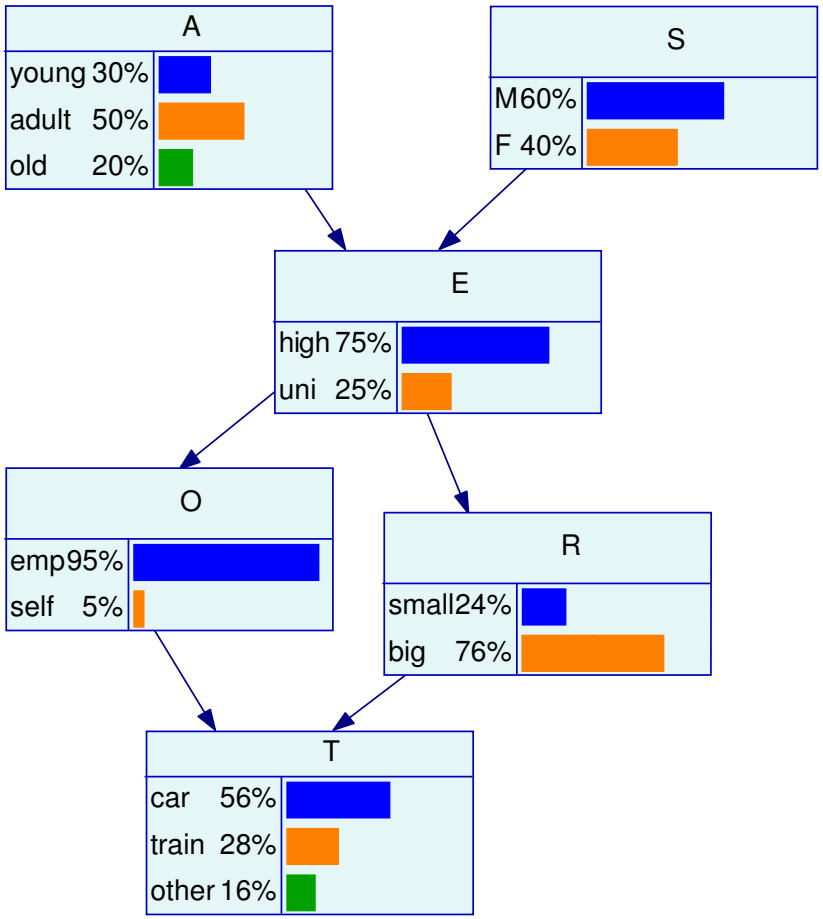
$$\mathbf{E} = \{X_{i_1} = e_1, X_{i_2} = e_2, \ldots, X_{i_k} = e_k\},$$

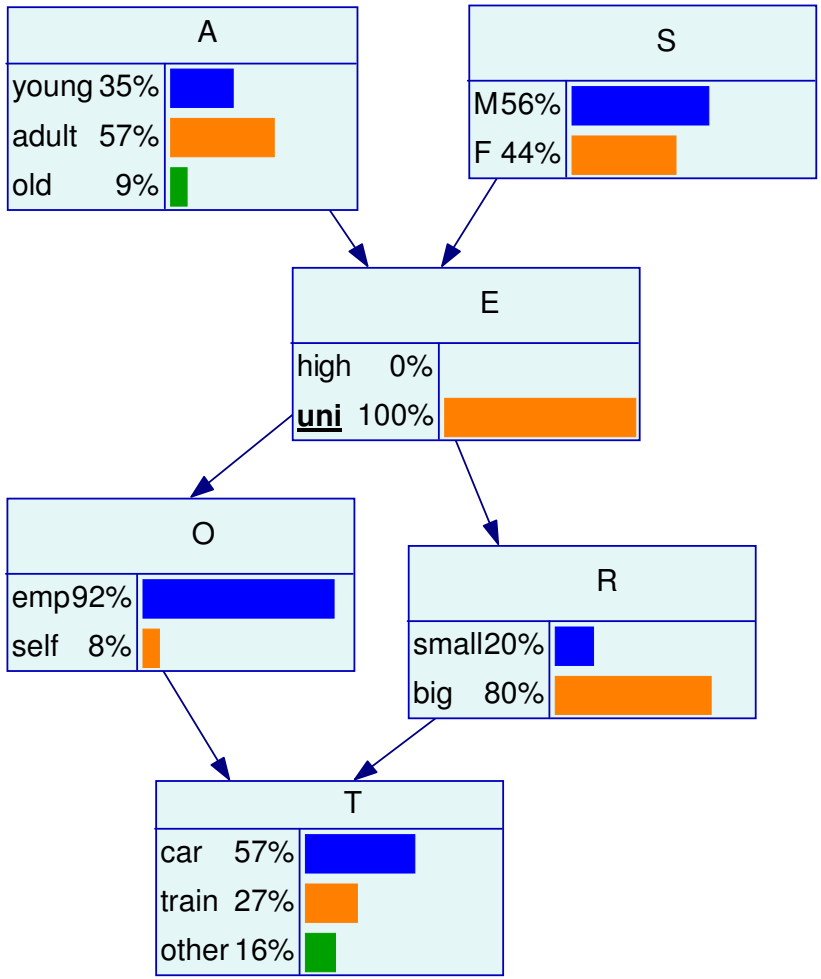
  which ranges from the value of a single variable X_i to a complete specification for $\mathbf{X}$ (such a new partial or complete observation).
- Soft evidence: a new distribution for one or more variables in the network. Since both the network structure and the distributional assumptions are treated as fixed, soft evidence is usually specified as a new set of parameters,

$$\mathbf{E} = \left\{X_{i_1} \sim (\Theta_{X_{i_1}}), X_{i_2} \sim (\Theta_{X_{i_2}}), \ldots, X_{i_k} \sim (\Theta_{X_{i_k}})\right\}.$$

  This new distribution may be, for instance, the null distribution in a hypothesis testing problem.

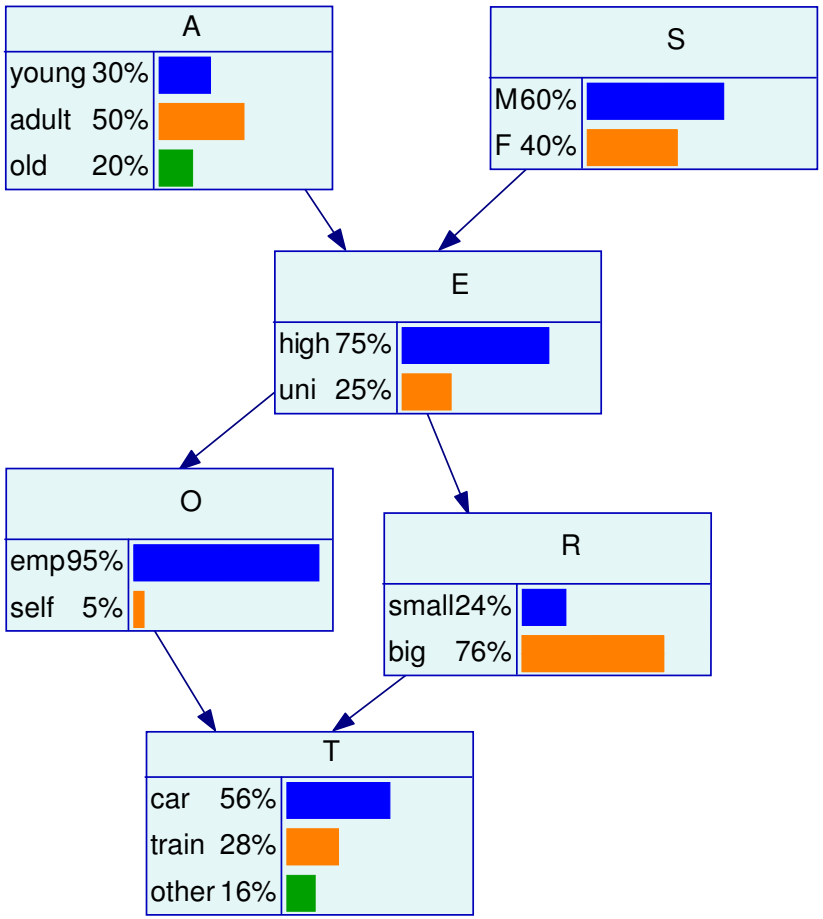
The Effects of Conditioning on Hard Evidence

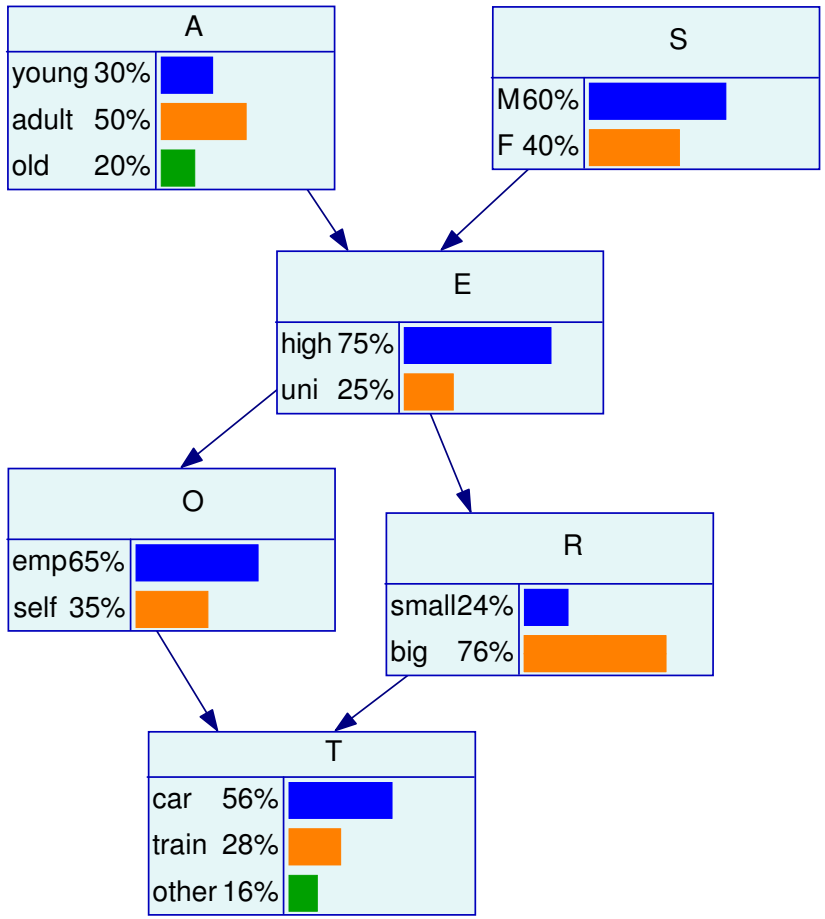




The original survey BN (left), and the posterior BN with hard evidence on Education (right).

The Effects of Conditioning on Soft Evidence





The original survey BN (left), and the posterior BN with soft evidence on Employment (right).

Conditional Probability Queries

Conditional probability queries are concerned with

$$\mathrm{CPQ}(\mathbf{Q} \mid \mathbf{E}, \mathcal{B}) = \mathrm{P}(\mathbf{Q} \mid \mathbf{E}, G, \Theta) = \mathrm{P}(X_{j_1}, \ldots, X_{j_l} \mid \mathbf{E}, G, \Theta),$$

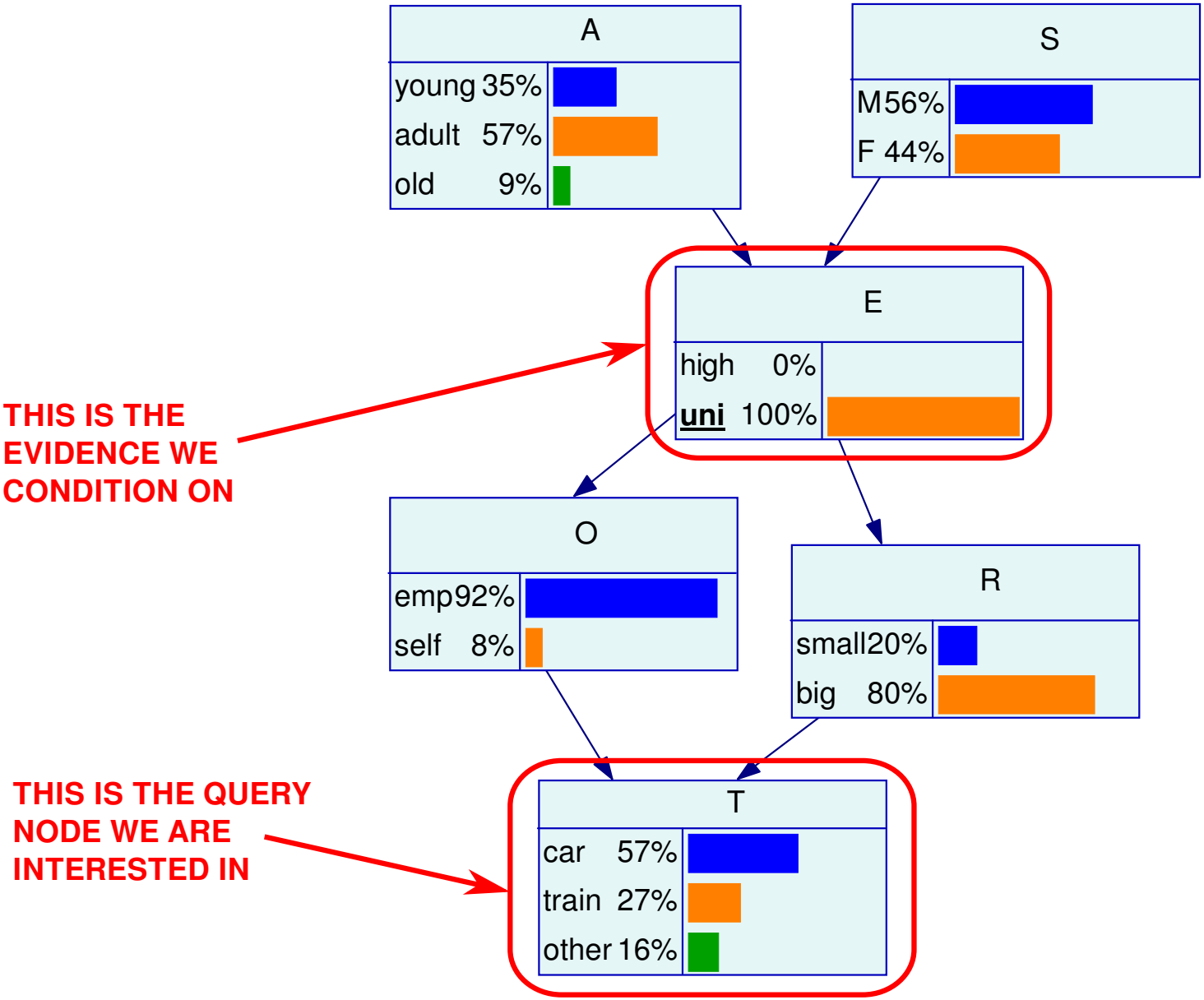
for some query variables $\mathbf{Q}$ given some hard evidence $\mathbf{E}$ on other variables, that is, the marginal posterior probability distribution of $\mathbf{Q}$,

$$\mathrm{P}(\mathbf{Q} \mid \mathbf{E}, G, \Theta) = \int \mathrm{P}(\mathbf{X} \mid \mathbf{E}, G, \Theta)\, d(\mathbf{X} \setminus \mathbf{Q}).$$

This class of queries has many useful applications due to their versatility.

For instance, we can assess the odds of an unfavourable outcome $\mathbf{Q}$ can for different sets of hard evidence $\mathbf{E}_1$, $\mathbf{E}_2$, $\ldots$, $\mathbf{E}_m$ of one or more related variables.

Conditional Probability Queries in Pictures



Maximum a Posteriori Queries

Maximum a posteriori queries are concerned with finding the configuration $\mathbf{q}^*$ of $\mathbf{Q}$ that has the highest posterior probability (for discrete BNs) or the maximum posterior density (for GBNs and CLGBNs),

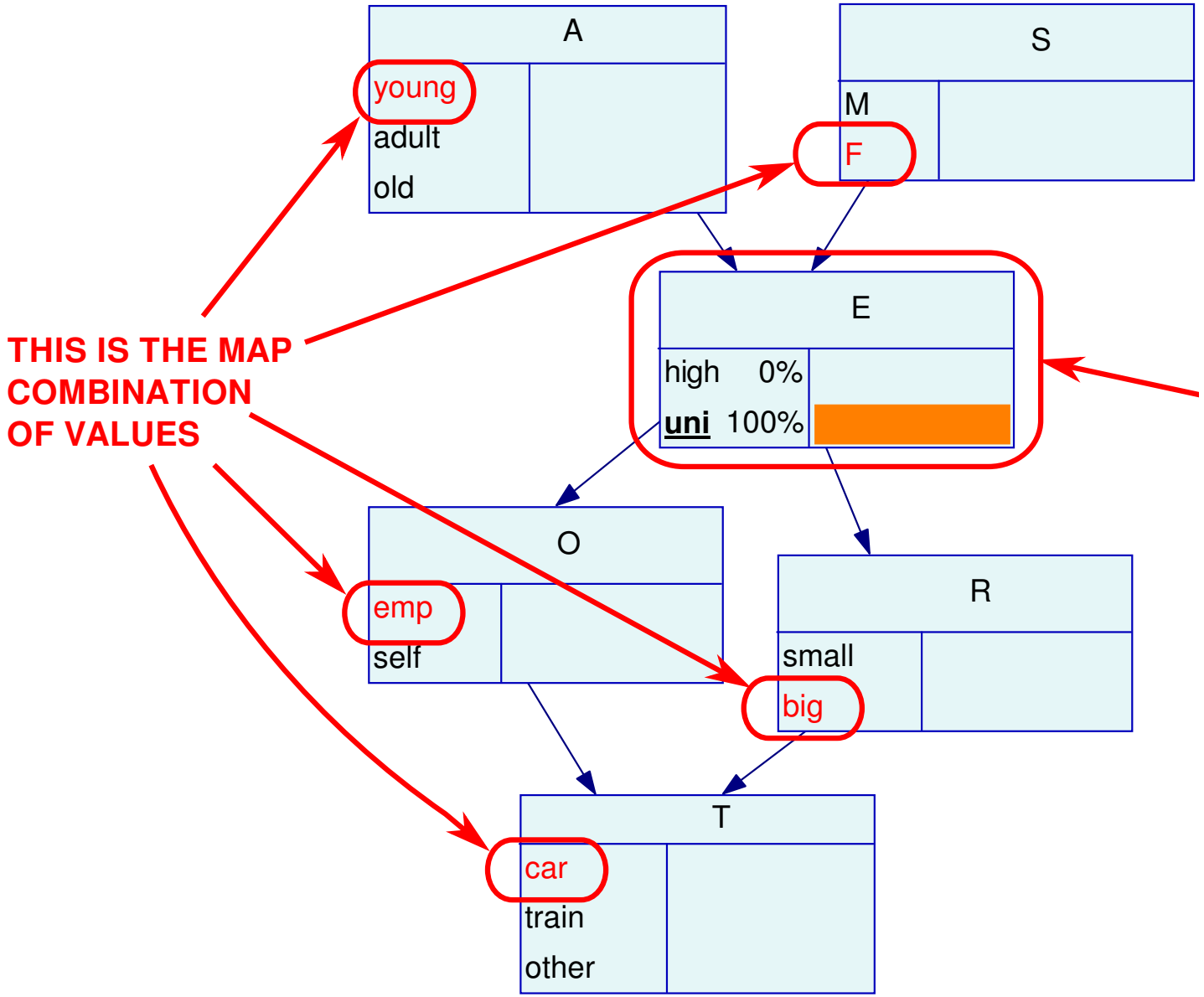
$$\text{MAP}(\mathbf{Q} \mid \mathbf{E}, \mathcal{B}) = \mathbf{q}^* = \underset{\mathbf{q}}{\text{argmax}}\, \text{P}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E}, G, \Theta). \tag{4}$$

Two main applications:

- imputing missing data, where the variables in $\mathbf{Q}$ are not observed and are imputed from those in $\mathbf{E}$;
- comparing $\mathbf{q}^*$ with the observed values for the variables in $\mathbf{Q}$.

NOTE: $\mathbf{q}^*$ is not the collection of the values with the highest posterior in each posterior marginal distribution, those distributions are not independent!

Maximum a Posteriori Queries in Pictures



How do We Update? Belief Propagation

The act of propagating the effects of evidence is called belief updating or belief propagation: our belief on $\mathbf{X}$ as encoded by the BN $\mathcal{B}$ is updated in the face of new evidence $\mathbf{E}$. This task is computationally feasible because we rely on local computations (only using local distributions):

$$\begin{aligned}
\mathrm{P}(\mathbf{Q} \mid \mathbf{E}, G, \Theta) &= \int \mathrm{P}(\mathbf{X} \mid \mathbf{E}, G, \Theta)\, d(\mathbf{X} \setminus \mathbf{Q}) \\
&= \int \left[\prod_{i=1}^{p} \mathrm{P}(X_i \mid \mathbf{E}, \Pi_{X_i}, \Theta_{X_i}) \right] d(\mathbf{X} \setminus \mathbf{Q}) \\
&= \prod_{i:X_i \in \mathbf{Q}} \int \mathrm{P}(X_i \mid \mathbf{E}, \Pi_{X_i}, \Theta_{X_i})\, dX_i.
\end{aligned}$$

The correspondence between d-separation and conditional independence can also be used to further reduce the dimension of the problem (e.g. to the Markov blanket).

Exact and Approximate Inference

Algorithms for belief updating can be classified either as

- Exact: algorithms that combine repeated applications of Bayes theorem with local computations to obtain the exact value of $\mathrm{P}(\mathbf{Q} \mid \mathbf{E}, G, \Theta)$. The two best known are
  - variable elimination; and
  - belief updates based on junction trees.
- Approximate: algorithms that use Monte Carlo simulations to sample from the global distribution and thus estimate $\mathrm{P}(\mathbf{Q} \mid \mathbf{E}, G, \Theta)$. In computer science, these random samples are often called particles, and the algorithms that make use of them are known as particle filters. The two best known are
  - logic sampling; and
  - likelihood weighting.

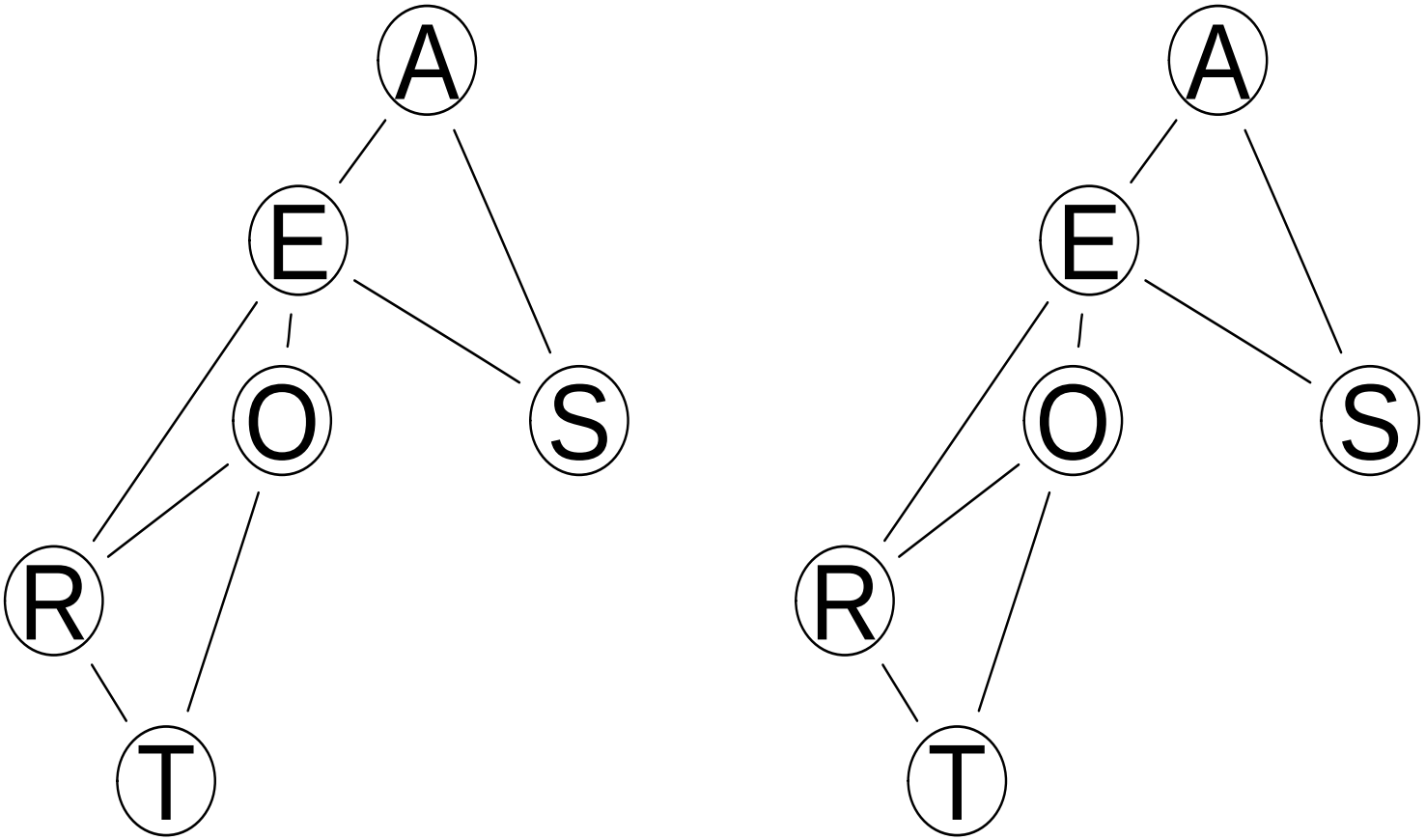
Approximate algorithms tend to scale better to larger number of variables since they are usually embarrassingly parallel; exact algorithms tend to be more sequential and iterative in nature.

The Junction Tree Clustering Algorithm

1. **Moralise:** create the moral graph of the BN $\mathcal{B}$.
2. **Triangulate:** break every cycle spanning 4 or more nodes into sub-cycles of exactly 3 nodes by adding arcs to the moral graph, thus obtaining a triangulated graph.
3. **Cliques:** identify the cliques $C_1, \ldots, C_k$ of the triangulated graph, i.e., maximal subsets of nodes in which each element is adjacent to all the others.
4. **Junction Tree:** create a tree in which each clique is a node, and adjacent cliques are linked by arcs. The tree must satisfy the running intersection property: if a node belongs to two cliques C_i and C_j, it must be also included in all the cliques in the (unique) path that connects C_i and C_j.
5. **Parameters:** use the parameters of the local distributions of $\mathcal{B}$ to compute the parameter sets of the compound nodes of the junction tree.

bnlearn: Moral Graphs

```
survey.dag1 = model2network("[A][S][E|A:S][O|E][R|E][T|O:R]")
survey.dag2 = model2network("[A|E][S|A:E][E|O:R][O|R:T][R|T][T]")
par(mfrow = c(1, 2))
graphviz.plot(moral(survey.dag1))
graphviz.plot(moral(survey.dag2))
```

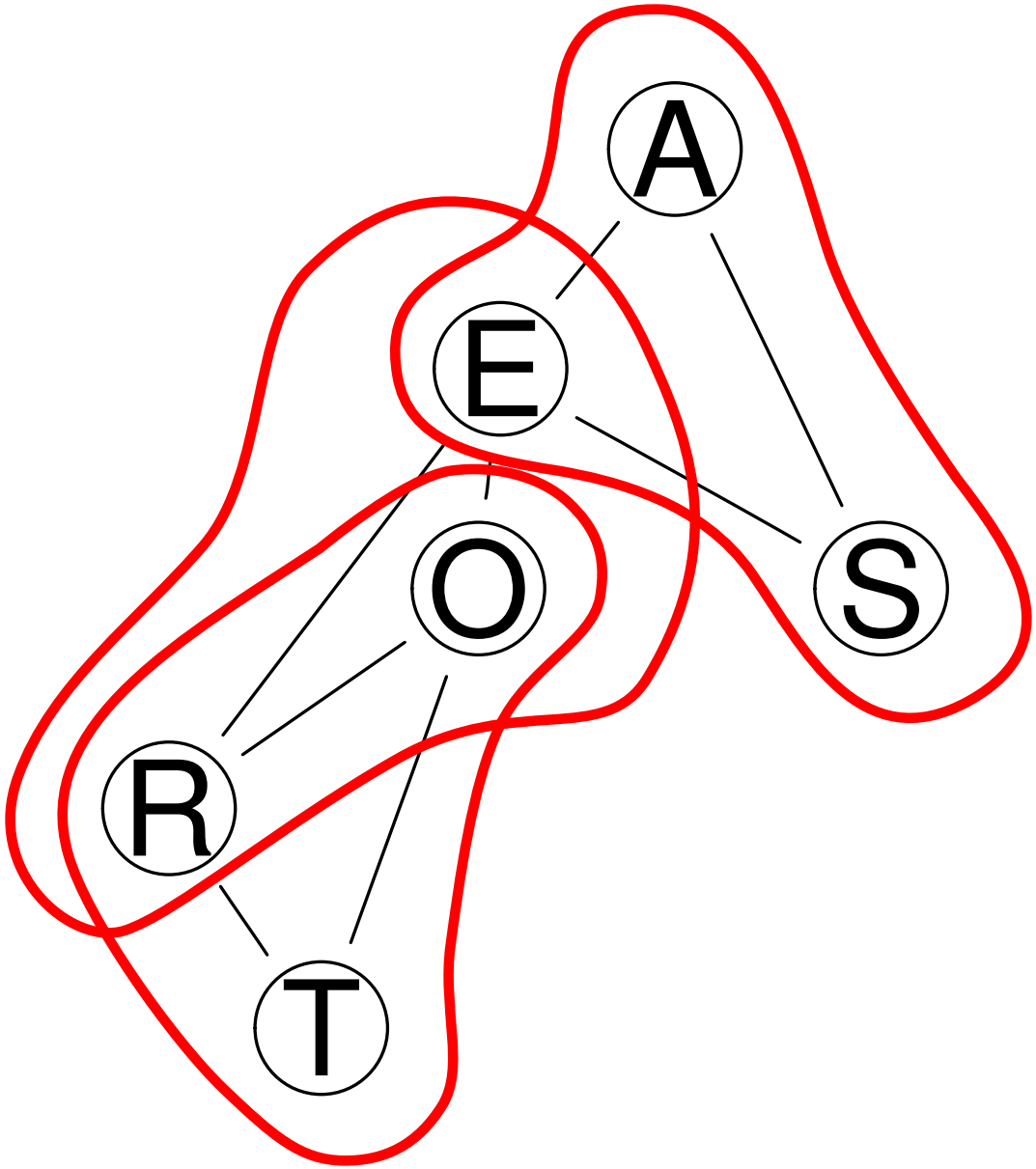


Moral Graphs, Diagnostic Models, Prognostic Models

So we can now see why probabilistic inference give the same results for diagnostic and prognostic models: they express the same set of dependencies, and therefore have the same moral graph, which means exact inference by means of junction trees will return the same results for conditional probability and maximum a posteriori queries. They are probabilistically indistinguishable.

This does not mean that causal inference will be the same, since in that case the direction of the arcs is crucial. The "target" (disease) node is modelled as a child of the other nodes in of the other nodes in prognostic models (risk factors lead to a disease), and as a parent in diagnostic models (the disease causes the symptoms).

Finding the Cliques



The moral graph is already triangulated, and we can see three cliques:

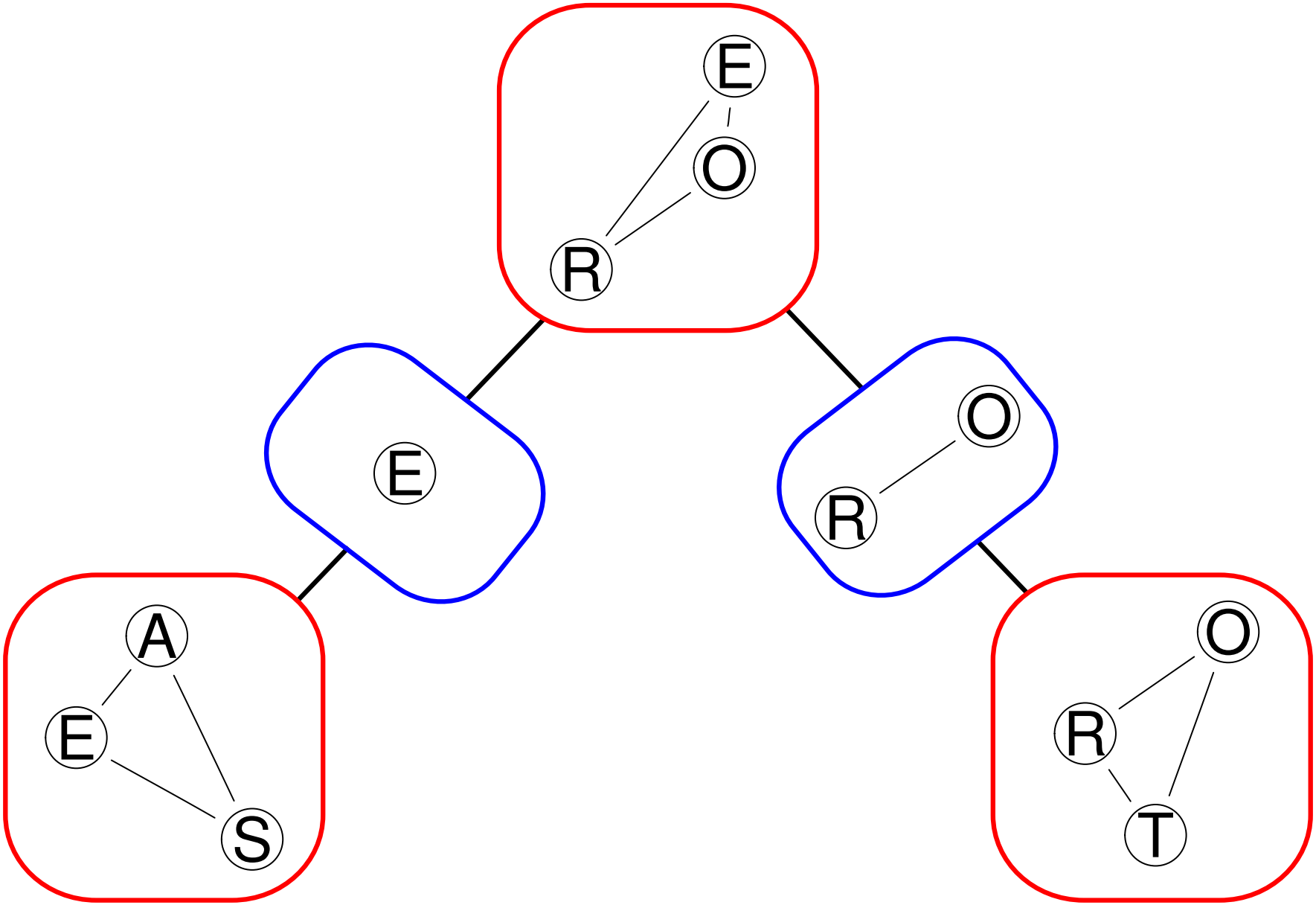
$$C_1 = \{A, E, S\}$$
$$C_2 = \{E, O, R\}$$
$$C_3 = \{O, R, T\}$$

with separators:

$$S_{12} = \{E\}$$
$$S_{23} = \{O, R\}$$

which we can use to build the junction tree.

Building the Junction Tree



Estimating the Parameters

In this example on the `survey` BN, the parameters for the cliques are:

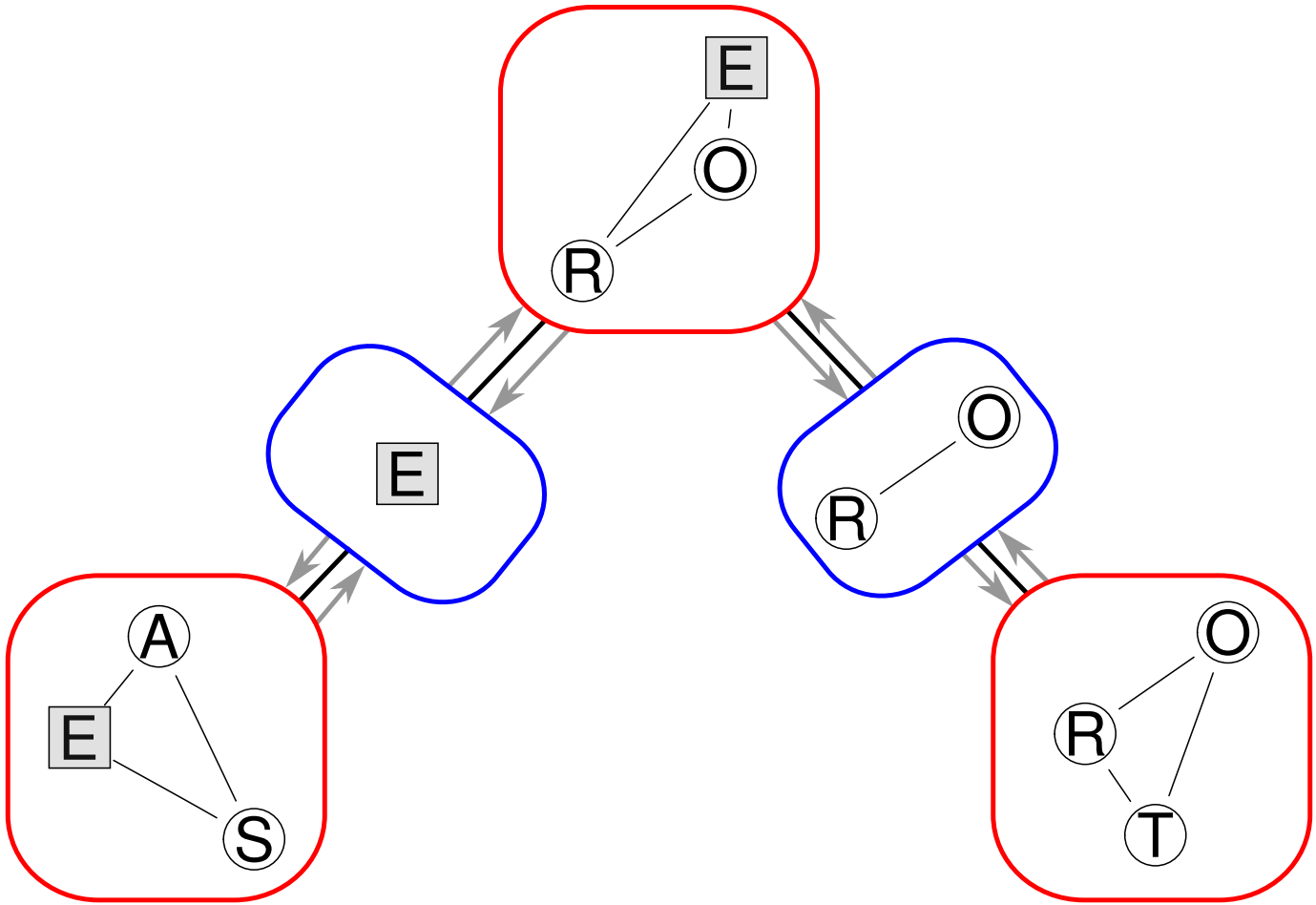
$$\Theta_{C_1} = \mathrm{P}(A, E, S) = \mathrm{P}(A)\,\mathrm{P}(S)\,\mathrm{P}(E \mid A, S)$$
$$\Theta_{C_2} = \mathrm{P}(E, O, R) = \mathrm{P}(O \mid E)\,\mathrm{P}(R \mid E)\,\mathrm{P}(E)$$
$$\Theta_{C_3} = \mathrm{P}(O, R, T) = \mathrm{P}(T \mid O, R)\,\mathrm{P}(O), \mathrm{P}(R)$$

and those for the separators are:

$$\Theta_{S_{12}} = \mathrm{P}(E)$$
$$\Theta_{S_{23}} = \mathrm{P}(O, R)$$

All can be readily computed from the local distributions in the BN.

Belief Propagation and Message Passing



Say we set Education to "high school"; we can change it directly in S_{12}, but then we need to propagate the changes to C_1 and C_2; and from C_2 to S_{23} and to C_3. This procedure is called belief propagation by message passing.

gRain: Exact Inference with Junction Trees

Junction trees and belief propagation as implemented in the **gRain** package. Suppose we would like to investigate the distribution of Sex and Travel given the evidence that Education is "high school".

First, we convert the BN from **bnlearn** to its equivalent in **gRain** with `as.grain()` and we construct the junction tree with `compile()`.

```
library(gRain)
junction = compile(as.grain(bn))
```

Then we set the evidence on the node, fixing it to "high school" with probability 1 with `setEvidence()`.

```
jedu = setEvidence(junction, nodes = "E", states = "high")
```

And after that, we can perform our conditional probability query with `querygrain()`, which also takes care of the belief propagation.

```
SxT.cpt = querygrain(jedu, nodes = c("S", "T"),
                     type = "joint")
```

Joint and Marginal Conditional Probabilities

The result of our query is the joint distribution of Sex and Travel given that Education is "high school".

```
SxT.cpt
```

```
##    T
## S     car train  other
##   M 0.343 0.174 0.0962
##   F 0.217 0.110 0.0609
```

Similarly, we can use `querygrain()` compute the marginal distributions of Sex and Travel conditional on Education.

```
querygrain(jedu, nodes = c("S", "T"), type = "marginal")
```

```
## $S
## S
##     M     F
## 0.613 0.387
##
## $T
## T
##   car train other
## 0.559 0.283 0.157
```

D-Separation and Conditional Independence

Interestingly, we can also compute the conditional distribution of Travel given Sex (still conditioning on Education being "high school"), which turns out to be:

```
querygrain(jedu, nodes = c("S", "T"), type = "conditional")
```

```
##         S
## T            M     F
##   car    0.613 0.387
##   train 0.613 0.387
##   other 0.613 0.387
```

This makes sense in the light of d-separation, which implies conditional independence.

```
dsep(bn, x = "S", y = "T", z = "E")
```

```
## [1] TRUE
```

The Logic Sampling Algorithm

1. Order the variables in $\mathbf{X}$ according to the topological partial ordering implied by G, say $X_{(1)} \prec X_{(2)} \prec \ldots \prec X_{(N)}$.

2. Set $n_{\mathbf{E}} = 0$ and $n_{\mathbf{E},\mathbf{q}} = 0$.

3. For a suitably large number of samples $\mathbf{x} = (x_1, \ldots, x_N)$:

   3.1 generate $x_{(i)}, i = 1, \ldots, N$ from $X_{(i)} \mid \Pi_{X_{(i)}}$ taking advantage of the fact that, thanks to the topological ordering, by the time we are considering X_i we have already generated the values of all its parents $\Pi_{X_{(i)}}$;

   3.2 if $\mathbf{x}$ includes $\mathbf{E}$, set $n_{\mathbf{E}} = n_{\mathbf{E}} + 1$;

   3.3 if $\mathbf{x}$ includes both $\mathbf{Q} = \mathbf{q}$ and $\mathbf{E}$, set $n_{\mathbf{E},\mathbf{q}} = n_{\mathbf{E},\mathbf{q}} + 1$.

4. Estimate $\mathrm{P}(\mathbf{Q} \mid \mathbf{E}, G, \Theta)$ with $n_{\mathbf{E},\mathbf{q}}/n_{\mathbf{E}}$.

bnlearn: Stepping Through Logic Sampling (I)

First, we sample the particles from the BN with `rbn()`, which takes a `bn.fit` object and the number of random samples to generate as arguments.

```
particles = rbn(bn, 10^6)
head(particles, n = 5)
```

```
##       A    E   O   R S     T
## 1   old high emp big M train
## 2   old high emp big M   car
## 3 adult high emp big F   car
## 4   old high emp big M other
## 5 young high emp big M   car
```

The particles are have the correct types and format as derived from the BN, and they are stored in a data frame that has the same structure as that of the data that were used to learn the BN (if any).

bnlearn: Stepping Through Logic Sampling (II)

Then we count how many of those samples that match the evidence $\mathbf{E}$ to estimate $\mathrm{P}(\mathbf{E})$.

```
partE = particles[(particles[, "E"] == "high"), ]
nE = nrow(partE)
```

We also count how many of those samples that match the evidence $\mathbf{E}$ and the query $\mathbf{Q} = \mathbf{q}$ to estimate $\mathrm{P}(\mathbf{Q} = \mathbf{q}, \mathbf{E})$.

```
partEq =
  partE[(partE[, "S"] == "M") & (partE[, "T"] == "car"), ]
nEq = nrow(partEq)
```

Finally, we estimate

$$\mathrm{P}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E}) = \frac{\mathrm{P}(\mathbf{Q} = \mathbf{q}, \mathbf{E})}{\mathrm{P}(\mathbf{E})}.$$

```
nEq/nE
```

```
## [1] 0.343
```

bnlearn: The cpquery() Function

These steps are implemented in `cpquery()`, with the obvious arguments:

- `event` is $\mathbf{Q}$;
- `evidence` is $\mathbf{E}$;
- `method` is `"ls"` for logic sampling (the default);
- `n` is the number of particles.

```
cpquery(bn, event = (S == "M") & (T == "car"),
          evidence = (E == "high"), method = "ls", n = 10^6)
## [1] 0.343
```

Both `event` end `evidence` are expressions that are evaluated on the particles much like `subset()` would, so they must evaluate to a vector of `TRUE` and `FALSE` values (hence `&` and not `&&`).

bnlearn: More Advanced Queries with cpquery()

Specifying the arguments requires some care, but the result is an extremely flexible framework to compute the probability of arbitrary combinations of events.

As an example of a more complex query, we can compute

$$\mathrm{P}(\mathtt{S} = \mathtt{M}, \mathtt{T} = \mathtt{car} \mid \{\mathtt{A} = \mathtt{young}, \mathtt{E} = \mathtt{uni}\} \cup \{\mathtt{A} = \mathtt{adult}\}),$$

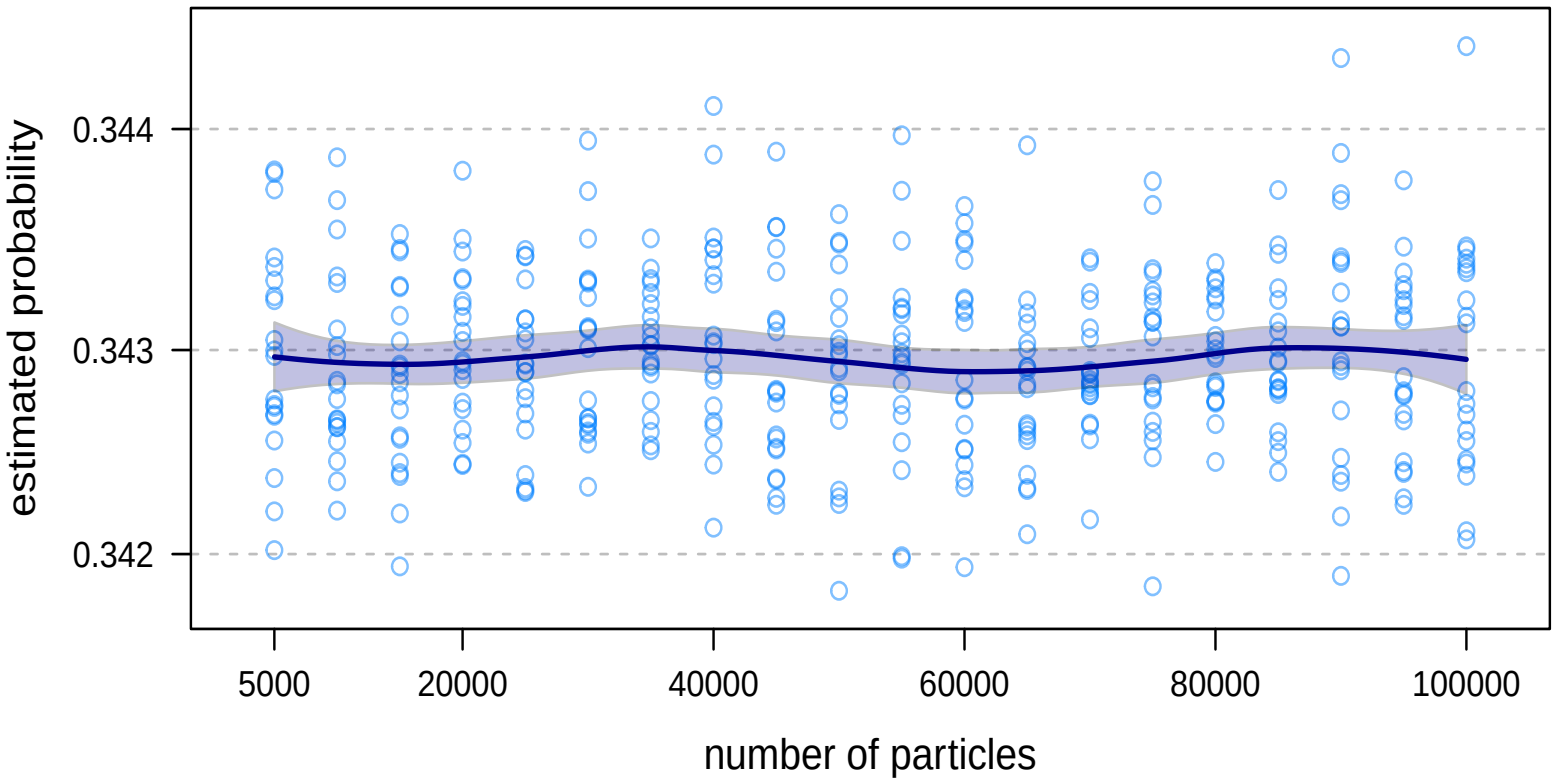
the probability of a man travelling by car given that his Age is young and his Education is uni or that he is an adult, regardless of his Education. That would be:

```
cpquery(bn, event = (S == "M") & (T == "car"),
  evidence = ((A == "young") & (E == "uni")) | (A == "adult"))
```

```
## [1] 0.338
```

bnlearn: Stepping Through Logic Sampling (III)

```
nparticles = seq(from = 5 * 10^3, to = 10^5, by = 5 * 10^3)
prob = matrix(0, nrow = length(nparticles), ncol = 20)
for (i in seq_along(nparticles))
  for (j in 1:20)
    prob[i, j] = cpquery(bn, event = (S == "M") & (T == "car"),
                         evidence = (E == "high"), method = "ls", n = 10^6)
```



The Likelihood Weighting Algorithm

An improvement over logic sampling, designed to solve this problem, is a form of importance sampling called likelihood weighting. Unlike logic sampling, all the particles generated by likelihood weighting include the evidence $\mathbf{E}$ by design.

---

1. Order the variables in $\mathbf{X}$ according to the topological ordering implied by G, say $X_{(1)} \prec X_{(2)} \prec \ldots \prec X_{(N)}$.
2. Set $w_{\mathbf{E}} = 0$ and $w_{\mathbf{E},\mathbf{q}} = 0$.
3. For a suitably large number of samples $\mathbf{x} = (x_1, \ldots, x_N)$:
   - 3.1 generate $x_{(i)}, i = 1, \ldots, N$ from $X_{(i)} \mid \Pi_{X_{(i)}}$ using the values $e_1, \ldots, e_k$ specified by the hard evidence $\mathbf{E}$ for $X_{i_1}, \ldots, X_{i_k}$.
   - 3.2 compute the weight $w_{\mathbf{x}} = \prod \mathrm{P}(X_{i^*} = e_* \mid \Pi_{X_{i^*}})$
   - 3.3 set $w_{\mathbf{E}} = w_{\mathbf{E}} + w_{\mathbf{x}}$;
   - 3.4 if $\mathbf{x}$ includes $\mathbf{Q} = \mathbf{q}$, set $w_{\mathbf{E},\mathbf{q}} = w_{\mathbf{E},\mathbf{q}} + w_{\mathbf{x}}$.
4. Estimate $\mathrm{P}(\mathbf{Q} \mid \mathbf{E}, G, \Theta)$ with $w_{\mathbf{E},\mathbf{q}}/w_{\mathbf{E}}$.

---

bnlearn: Stepping Through Likelihood Weighting (I)

We do not want to sample from the original BN, but from the BN in which all the nodes $X_{i_1}, \ldots, X_{i_k}$ in $\mathbf{E}$ are fixed. This network is called the mutilated network.

```
mutbn = mutilated(bn, list(E = "high"))
coef(mutbn$E)
```

```
## high  uni
##    1    0
```

Simply sampling from `mutbn` is not a valid approach. If we do so, the probability we obtain is $\mathrm{P}(\mathbf{Q}, \mathbf{E} \mid G, \Theta)$, not $\mathrm{P}(\mathbf{Q} \mid \mathbf{E}, G, \Theta)$!

Firstly, we sample particles from the original BN one more time.

```
particles = rbn(mutbn, 10^6)
partQ = particles[(particles[, "S"] == "M") &
                  (particles[, "T"] == "car"), ]
```

bnlearn: Stepping Through Likelihood Weighting (II)

A simple empirical checks tells us that the naive estimate we could draw from `mutbn` is wrong, since it does not match the exact value we got earlier.

```
nrow(partQ) / nrow(particles)
```

```
## [1] 0.336
```

The weights adjust for the fact that we are sampling from the mutilated BN instead of original BN. The weights are just the likelihood components for the particles associated with the nodes we are conditioning on (`E` in this case).

```
w = logLik(bn, particles, nodes = "E", by.sample = TRUE)
wEq = sum(exp(w[(particles[, "S"] == "M") &
                (particles[, "T"] == "car")]))
wE = sum(exp(w))
wEq/wE
```

```
## [1] 0.343
```

bnlearn: Stepping Through Likelihood Weighting (III)

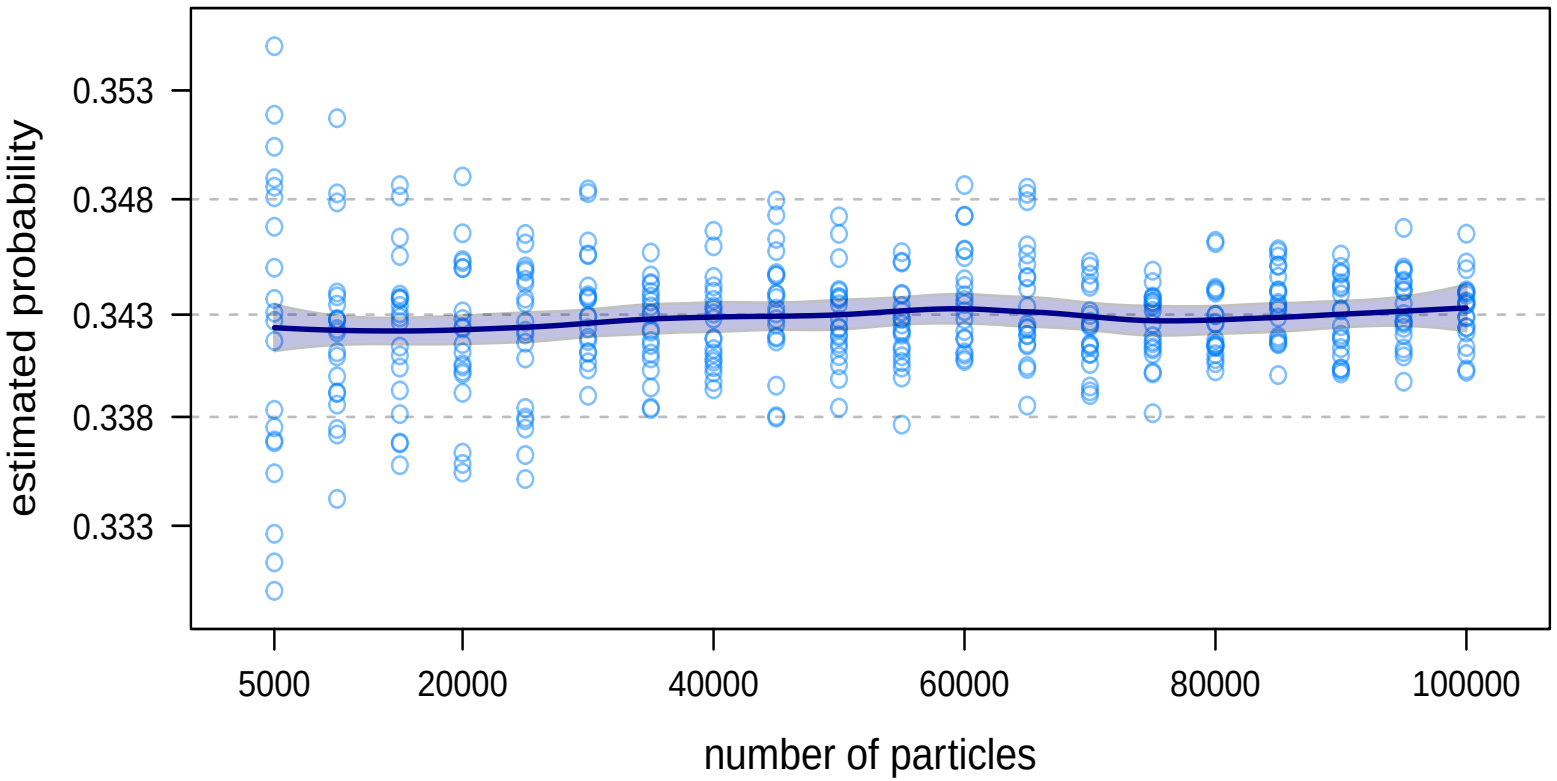
More conveniently, we can perform likelihood weighting with `cpquery` by setting `method = "lw"` and specifying the evidence as a named list with one element for each node we are conditioning on.

```
cpquery(bn, event = (S == "M") & (T == "car"),
          evidence = list(E = "high"), method = "lw", n = 5 * 10^4)
## [1] 0.343
```

The estimate we obtain is still very precise with small numbers of particles, as was the case for logic sampling, but the variability of the estimated probabilities is actually larger. There is no guarantee that likelihood weighting will always have lower variance than logic sampling.

bnlearn: Stepping Through Likelihood Weighting (IV)

```
nparticles = seq(from = 5 * 10^3, to = 10^5, by = 5 * 10^3)
prob = matrix(0, nrow = length(nparticles), ncol = 20)
for (i in seq_along(nparticles))
  for (j in 1:20)
    prob[i, j] = cpquery(bn, event = (S == "M") & (T == "car"),
                         evidence = list(E = "high"), method = "lw",
                         n = nparticles[i])
```



Then Why Use Likelihood Weighting?

Logic sampling will be computationally inefficient and very inaccurate if $\mathrm{P}(\mathbf{E})$ is small because most particles will be discarded without contributing to the estimation of $\mathrm{P}(\mathbf{Q} \mid \mathbf{E})$.

```
extreme.dag = model2network("[A][B|A]")
A.prob = array(c(0.999999, 0.000001), dim = 2,
           dimnames = list(A = c("a1", "a2")))
B.prob = array(c(0.5, 0.5, 0.75, 0.25), dim = c(2, 2),
           dimnames = list(B = c("b1", "b2"), A = c("a1", "a2")))
extreme.bn = custom.fit(extreme.dag, list(A = A.prob, B = B.prob))
cpquery(extreme.bn, event = (B == "b2"), evidence = (A == "a2"),
  method = "ls", n = 10^6)
```

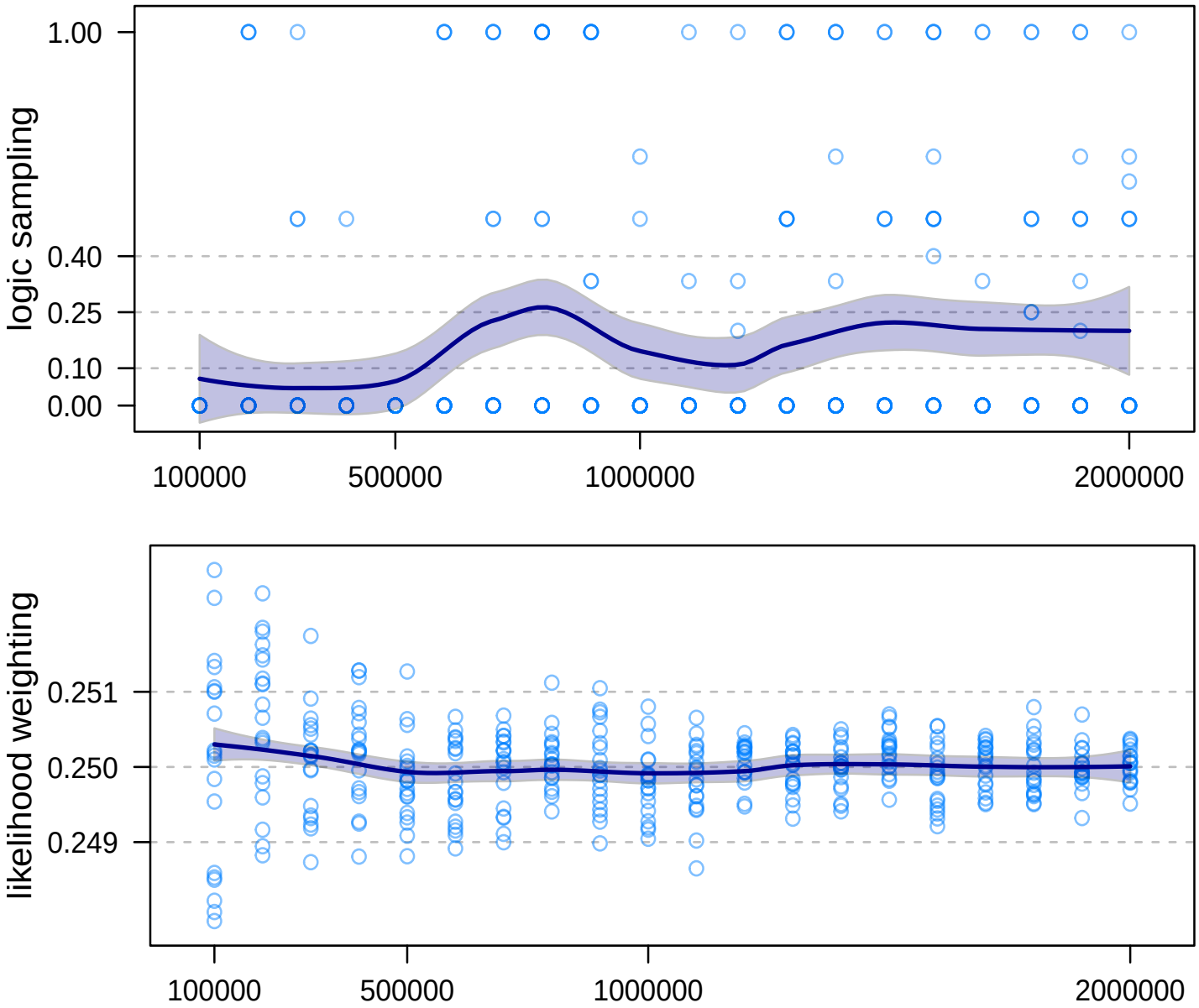
```
## [1] 0
```

This simply does not happen with likelihood weighting.

```
cpquery(extreme.bn, event = (B == "b2"), evidence = list(A = "a2"),
  method = "lw", n = 5 * 10^3)
```

```
## [1] 0.243
```

A Comparison for Different Numbers of Particles



bnlearn: Extensions of Likelihood Weighting

The `event` is still a general expression, which means it is possible to describe complex events. However, likelihood weighting relies on the fact that the `evidence` is fixed to a single value to compute the weights. In **bnlearn** this assumption is relaxed: the `event` can take more than one value for each variable. All combinations of values are given the same probability so as not to alter the weights:

$$\mathrm{P}(\mathbf{Q} \mid \mathbf{E} = \bigcup_i \mathbf{E}_i) = \sum_i \mathrm{P}(\mathbf{Q} \mid \mathbf{E}_i)\,\mathrm{P}(\mathbf{E}_i) = \frac{1}{|\mathbf{E}|} \sum_i \mathrm{P}(\mathbf{Q} \mid \mathbf{E}_i)$$

```
cpquery(bn, event = (S == "M") & (T == "car"),
  evidence = list(A = c("young", "adult")), method = "lw", n = 10^6)
```

```
## [1] 0.337
```

```
cpquery(bn, event = (S == "M") & (T == "car"),
  evidence = list(A = "young"), method = "lw", n = 10^6) * 0.5 +
cpquery(bn, event = (S == "M") & (T == "car"),
  evidence = list(A = "adult"), method = "lw", n = 10^6) * 0.5
```

```
## [1] 0.337
```

bnlearn: Sampling and Conditioning

Last but not least, we can also use `cpdist()` to generate particles conditional on some evidence $\mathbf{E}$. Likelihood weighting works best, and attaches the weights to the particles (for use in later analyses).

```
cpdist(bn, nodes = c("S", "T"), evidence = list(A = "adult"),
  method = "lw", n = 5)
```

```
##   S     T
## 1 M other
## 2 M   car
## 3 M   car
## 4 F   car
## 5 F train
```

Logic sampling works less well because, being a form of rejection sampling, often returns far fewer observations than requested.

```
cpdist(bn, nodes = c("S", "T"), evidence = (A == "young"),
  method = "ls", n = 5)
```

```
##   S   T
## 1 M car
## 2 F car
```

Bayesian Network Classifiers

BNs can also be used as classifiers, to predict which of several classes each observation belongs to. Assuming class labels are observed (so we can train the BN classifier in what is called supervised learning).

The focus in this case is predictive accuracy for new observations instead of representing faithfully the dependence structure of $\mathbf{X}$. There is no implication that an "interpretable" BN will provide good predictive accuracy; on the contrary we introduce bias in the form of an artificially simple DAG to improve the predictive performance of the BN (a la bias-variance trade-off).

Here we will see the two most common BN classifiers:

- the Naive Bayes classifier; and
- the Tree-Augmented Naive Bayes (TAN) classifier.

Naive Bayes Classifier

Let X_C be the training variable and $\mathbf{X} \setminus X_C$ be the explanatory variables which will be used for prediction. Then we can use Bayes theorem to write the posterior probabilities $\mathrm{P}(X_C = c_i \mid \mathbf{X} \setminus X_C)$ as

$$\mathrm{P}(X_C \mid \mathbf{X} \setminus X_C) = \frac{\mathrm{P}(X_C, \mathbf{X} \setminus X_C)}{\mathrm{P}(\mathbf{X} \setminus X_C)} = \frac{\mathrm{P}(\mathbf{X} \setminus X_C \mid X_C)\,\mathrm{P}(X_C)}{\mathrm{P}(\mathbf{X} \setminus X_C)}.$$

If we assume that explanatory variables are independent then

$$\mathrm{P}(\mathbf{X} \setminus X_C \mid X_C) = \prod_{X_i \in \mathbf{X} \setminus X_C} \mathrm{P}(X_i \mid X_c)$$

and the $\mathrm{P}(X_C)$ works as prior probabilities.

The DAG corresponding to that dependence structure has arcs $X_C \to X_i$, so that all X_i depend on X_C but are independent from each other.

Predicting from a Naive Bayes Classifier

The class labels of new observations is predicted as that that maximises

$$\mathrm{P}(X_C \mid \mathbf{X} \setminus X_C) \propto \mathrm{P}(X_C) \prod_{X_i \in \mathbf{X} \setminus X_C} \mathrm{P}(X_i \mid X_c),$$

that is, by maximum a posteriori probability.

The simplicity of this model has several advantages:

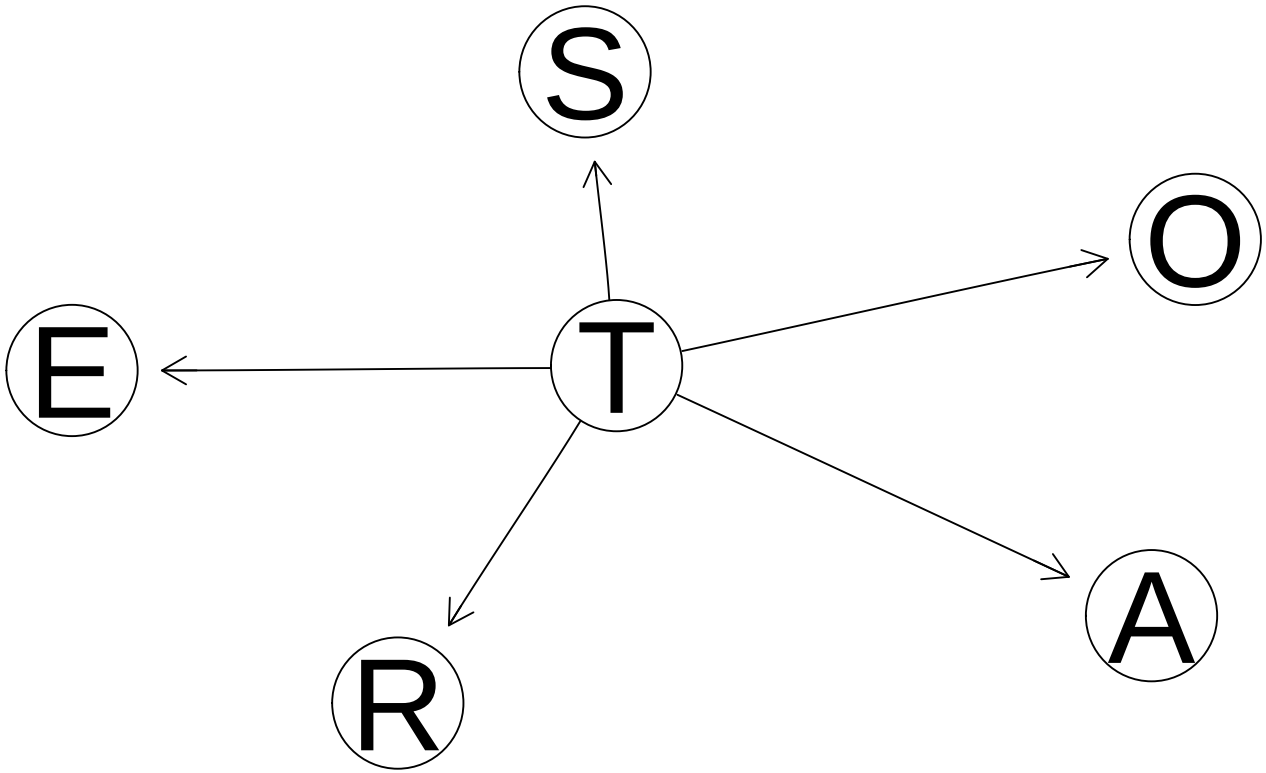
- There are comparatively few parameters to estimate.
- It is easy to include variables following different distributions as explanatory variables, and model them as mixtures.
- The DAG underlying the BN is not estimated from the data, so it is not affected by noise and often outperforms more complex models.

Several R implementations: **bnlearn**, **e1071**, etc.

bnlearn: Naive Bayes Classifier

We can create the star-shaped structure of the BN with `naive.bayes()`, specifying the data and the training variable with the class labels.

```
survey = read.table("../data/survey.txt", header = TRUE)
nbcl = naive.bayes(survey, training = "T")
graphviz.plot(nbcl, layout = "fdp")
```



bnlearn: Training the Classifier

Training the classifier means learning its parameters from the data (since the structure is fixed), which we can do with `bn.fit()`.

```
nbcl.trained = bn.fit(nbcl, survey)
```

This gives us the conditional probabilities tables for the explanatory variables and the class probabilities.

```
coef(nbcl.trained$T)
```

```
##   car other train
##  0.58  0.17  0.25
```

```
coef(nbcl.trained$O)
```

```
##       T
## O          car  other  train
##   emp  0.9586 0.9647 0.9840
##   self 0.0414 0.0353 0.0160
```

bnlearn: Evaluating the Classifier with Cross-Validation

We then check the predictive accuracy of the classifier using cross-validation to obtain an estimate of the predictive classification error. The golden standard is 10 runs of 10-fold cross-validation, using `bn.cv()` with `method = "k-fold"`.

```
cv.nb = bn.cv(nbcl, data = survey, runs = 10, method = "k-fold", folds = 10)
cv.nb
```

```
##
##   k-fold cross-validation for Bayesian networks
##
##   target network structure:
##    [Naive Bayes Classifier]
##   number of folds:                         10
##   loss function:                           Classification Error
##   training node:                           T
##   number of runs:                          10
##   average loss over the runs:              0.421
##   standard deviation of the loss:          0.00267
```

Clearly, the classifier is not very good since it gets predictions right only $\approx 60\%$ of the time.

bnlearn: A Comparison with the Original Network

The original network does not do any worse (or any better)...

```
cv.orig = bn.cv(survey.dag, data = survey, runs = 10, method = "k-fold",
           folds = 10, loss = "pred", loss.args = list(target = "T"))
cv.orig
```

```
##
##   k-fold cross-validation for Bayesian networks
##
##   target network structure:
##    [A][S][E|A:S][O|E][R|E][T|O:R]
##   number of folds:                         10
##   loss function:                           Classification Error
##   training node:                           T
##   number of runs:                          10
##   average loss over the runs:              0.421
##   standard deviation of the loss:          0.0017
```

Here we need to specify a few extra arguments do match what we did for the naive Bayes classifier: the `loss` function and the `target` variable to predict.

Tree-Augmented Naive Bayes Classifier (TAN)

Assuming that explanatory variables are independent is a very strong assumption. One way to relax it while keeping the DAG simple is to assume that each explanatory variable depends from one other explanatory variable:

$$\mathrm{P}(X_C \mid \mathbf{X} \setminus X_C) \propto \mathrm{P}(X_C) \prod_{X_i \in \mathbf{X} \setminus X_C} \mathrm{P}(X_i \mid X_{j \neq i}, X_c),$$

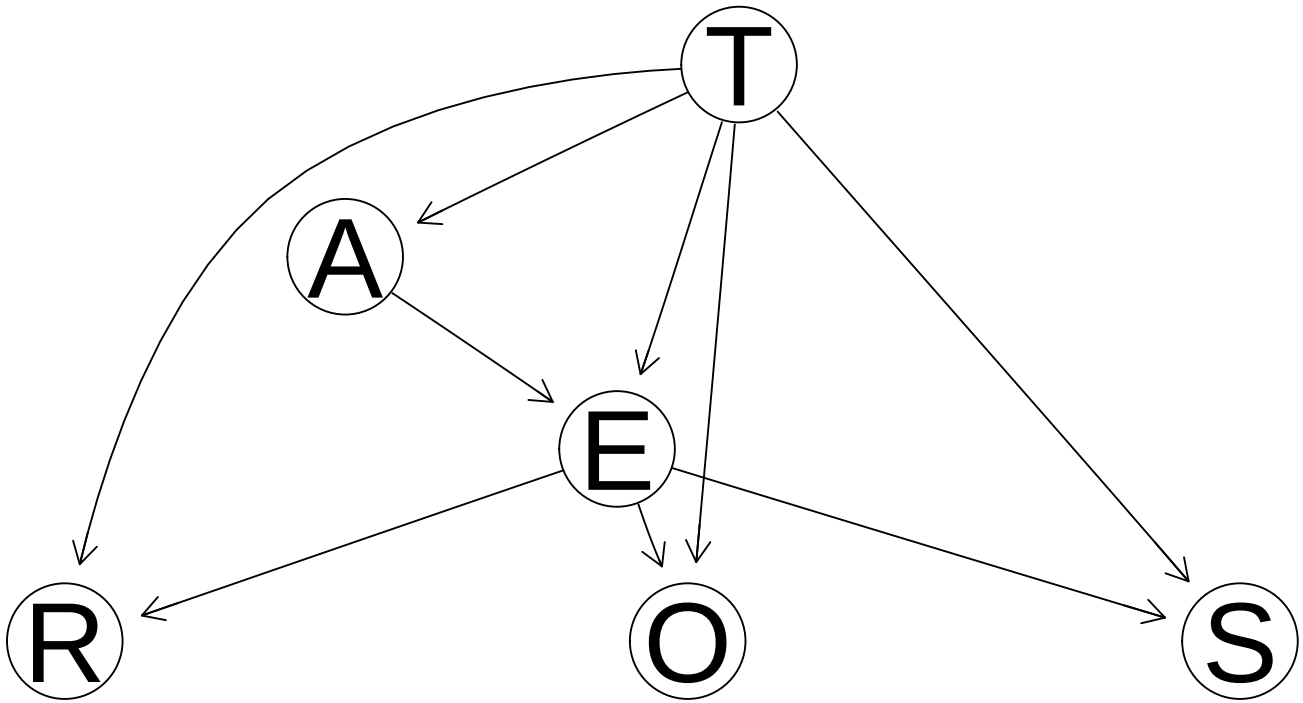
This determines a tree dependence structure among the explanatory variables, which is estimated from the data using Chow-Liu minimum weight spanning trees and picking the arcs $X_j \to X_i$ that have the highest

$$\frac{\mathrm{P}(X_i \mid X_{j \neq i}, X_c)}{\mathrm{P}(X_i \mid X_c)}.$$

bnlearn: Tree-Augmented Naive Bayes Classifier

The `tree.bayes()` function learns the structure of the BN from the data. The root node for the tree is picked at random, unless specified with the `root` argument.

```
tancl = tree.bayes(survey, training = "T")
graphviz.plot(tancl)
```



bnlearn: Training the Classifier

Training the classifier is as before...

```
tancl.trained = bn.fit(tancl, survey)
```

... and we can see that each explanatory variable has one parent besides the training variable.

```
coef(tancl.trained$O)
```

```
## , , E = high
##
##       T
## O         car  other  train
##   emp  0.9815 0.9825 0.9783
##   self 0.0185 0.0175 0.0217
##
## , , E = uni
##
##       T
## O         car  other  train
##   emp  0.8919 0.9286 1.0000
##   self 0.1081 0.0714 0.0000
```

bnlearn: Evaluating the Classifier with Cross-Validation

The predictive accuracy of the TAN is similar to that of naive Bayes and the original network.

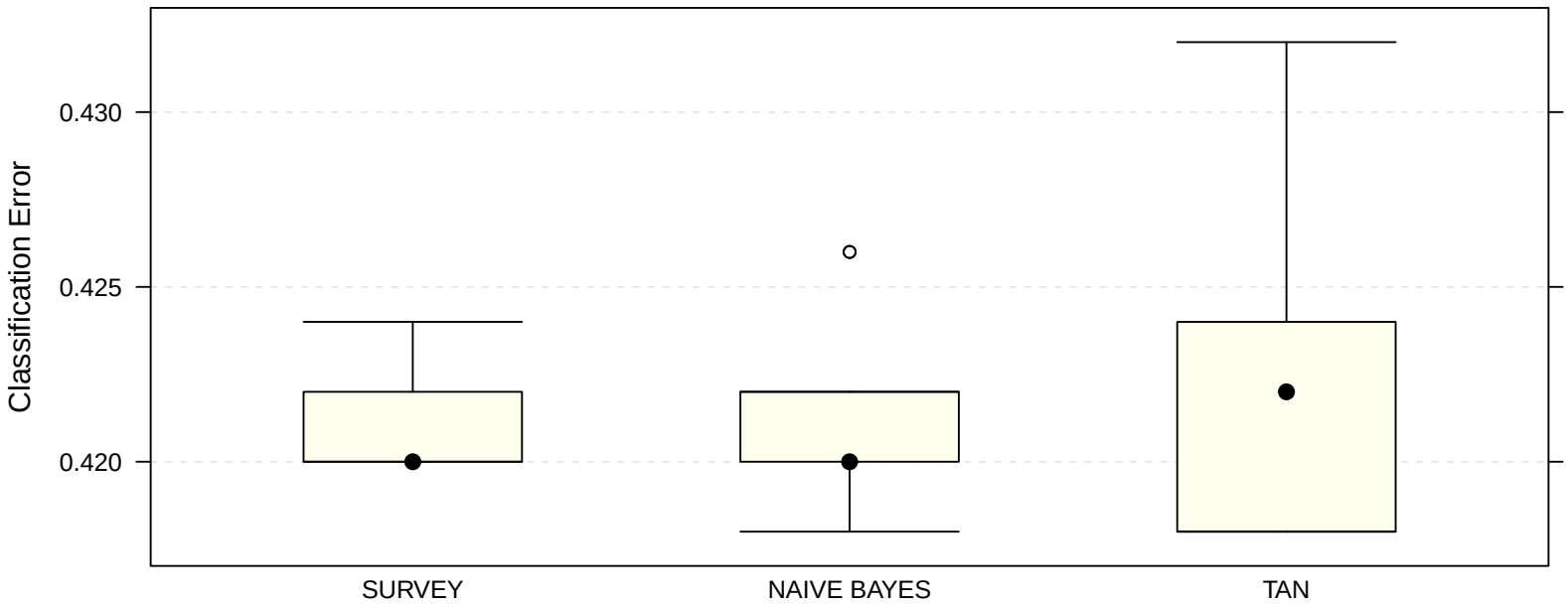
```
cv.tan = bn.cv("tree.bayes", data = survey, runs = 10, method = "k-fold",
           folds = 10, algorithm.args = list(training = "T"))
cv.tan
```

```
##
##   k-fold cross-validation for Bayesian networks
##
##   target learning algorithm:              TAN Bayes Classifier
##   number of folds:                        10
##   loss function:                          Classification Error
##   training node:                          T
##   number of runs:                         10
##   average loss over the runs:             0.422
##   standard deviation of the loss:         0.0042
```

The slightly higher variability is expected, since the DAG is estimated from the data instead of being completely fixed.

bnlearn: Graphical Comparison

```
plot(cv.orig, cv.nb, cv.tan, xlab = c("SURVEY", "NAIVE BAYES", "TAN"))
```



A plot of the average classification errors for the various BNs suggests that naive Bayes performs the same as the original DAG, and TAN is worse. However, the magnitude of the differences is so small as not to be practically significant.

Summary

- There are two kinds of questions: conditional probability queries and maximum a posteriori queries. The latter can be answered from the former.
- There are two kinds of way of answering such questions: exact and approximate inference. One uses Bayes theorem and is more accurate, the other Monte Carlo sampling and is more scalable.
- Now we know why diagnostic and prognostic models are interchangeable for inference: they have the same moral graph.
- BNs can also be used for classification, by using maximum a posteriori queries for prediction. The DAG is simpler in order to improve predictive accuracy by introducing bias in a bias-variance trade-off.

Advanced Inference

Bayesian Networks are not Necessarily Causal

In the previous lecture, we have defined BNs in terms of conditional independence relationships and probabilistic properties, without any implication that arcs should represent cause-and-effect relationships.

The existence of equivalence classes of networks that are indistinguishable from a probabilistic point of view provides a simple proof that arc directions are not indicative of causal effects. The fact that are prognostic and diagnostic formulations of the same BN are identical in terms of inference is another strong hint.

Therefore, while it is appealing to interpret the direction of arcs in causal terms, please do not do it lightly, especially with observational data.

Probabilistic and Causal Bayesian Networks

However, from an intuitive point of view it can be argued that a "good" BN should represent the causal structure of the data it is describing. Such BN are usually fairly sparse, and their interpretation is at the same time clear and meaningful, as explained by Judea Pearl in his book on causality:

> *It seems that if conditional independence judgments are byproducts of stored causal relationships, then tapping and representing those relationships directly would be a more natural and more reliable way of expressing what we know or believe about the world. This is indeed the philosophy behind causal BNs.*

This is the reason why building a BN from expert knowledge in practice codifies known and expected causal relationships for a given phenomenon.

What Additional Assumptions Do We Need For Causality?

We need three additional assumptions:

- Each variable X_i is conditionally independent of its non-effects, both direct and indirect, given its direct causes (the causal Markov assumption, much like the original but causal);
- There must exist a DAG which is faithful to the probability distribution $\mathbf{P}$ of $\mathbf{X}$, so that the only dependencies in $\mathbf{P}$ are those arising from d-separation in the DAG.
- There must be no latent variables (unobserved variables influencing the variables in the network) acting as confounding factors. Such variables may induce spurious correlations between the observed variables, thus introducing bias in the causal network.

What Additional Assumptions Do We Need For Causality?

The third assumption descends from the first two:

- the presence of unobserved variables violates the faithfulness assumption, because the network structure does not include them;
- and possibly the causal Markov property, because an arc may be wrongly added between two observed variables due to the influence of the latent one.

These assumptions are difficult to verify in real-world settings, as the set of the potential confounding factors is not usually known. At best, we can address this issue, along with selection bias, by implementing a carefully planned experimental design in which we use blocking to screen out confounding.

Causality and Equivalence Classes

Even when dealing with interventional data collected from a scientific experiment (where we can control at least some variables and observe the resulting changes), there are usually multiple equivalent BNs that represent reasonable causal models. Many arcs may not have a definite direction, resulting in substantially different DAG. When the sample size is small there may also be several non-equivalent BN fitting the data equally well.

Therefore, in general we are not able to identify a single, "best", causal BN but rather a small set of likely causal BN that fit our knowledge of the data.

The MARKS Example, Revisited

An example of the bias introduced by the presence of a latent variable was illustrated by Edwards ("*Introduction to Graphical Modelling*") using the `marks` data. This data set was originally investigated by Mardia ("*Multivariate Analysis*") and subsequently in Whittaker ("*Graphical Models in Applied Multivariate Statistics*").

`marks` contains the exam scores between 0 and 100 for 88 students across 5 different topics, namely: mechanics (`MECH`), vectors (`VECT`), algebra (`ALG`), analysis (`ANL`) and statistics (`STAT`).

```
library(bnlearn)
head(marks)
```

```
##   MECH VECT ALG ANL STAT
## 1   77   82  67  67   81
## 2   63   78  80  70   81
## 3   75   73  71  66   81
## 4   55   72  63  70   68
## 5   63   63  65  70   63
## 6   53   61  72  64   73
```

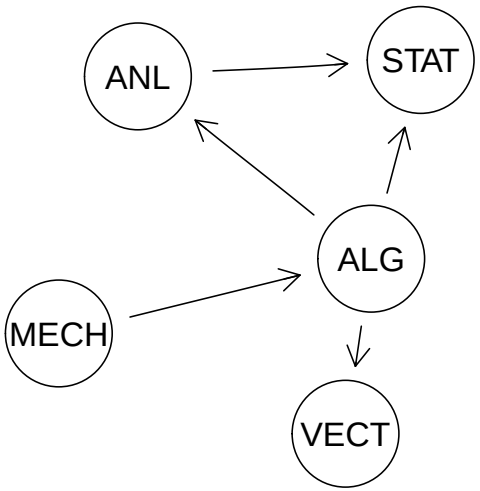
Add Latent Grouping...

Edwards noted that the students apparently belonged to two groups (which we will call `A` and `B`) with substantially different academic profiles. He then assigned each student to one of those two groups using the EM algorithm to impute group membership as a latent variable (say, `LAT`). The EM algorithm assigned the first 52 students (with the exception of number 45) to group `A`, and the rest to group `B`.

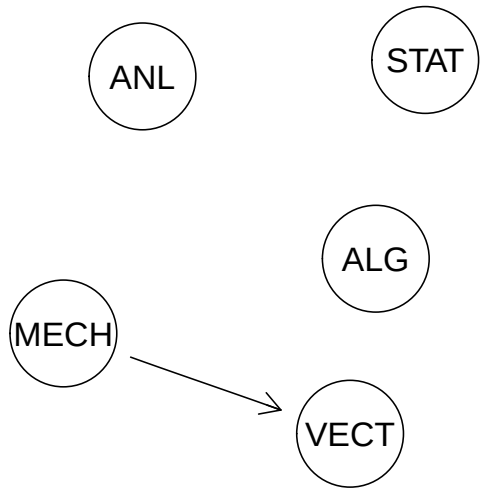
```
latent = factor(c(rep("A", 44), "B", rep("A", 7), rep("B", 36)))
modelstring(hc(marks[latent == "A", ]))
## [1] "[MECH][ALG|MECH][VECT|ALG][ANL|ALG][STAT|ALG:ANL]"
modelstring(hc(marks[latent == "B", ]))
## [1] "[MECH][ALG][ANL][STAT][VECT|MECH]"
modelstring(hc(marks))
## [1] "[MECH][VECT|MECH][ALG|MECH:VECT][ANL|ALG][STAT|ALG:ANL]"
```

... And the Models Look Nothing Alike

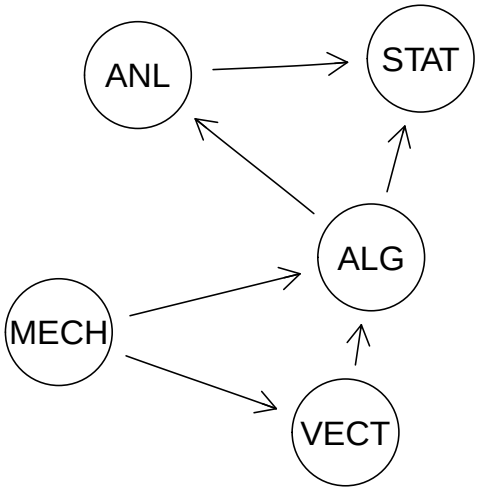
Group A



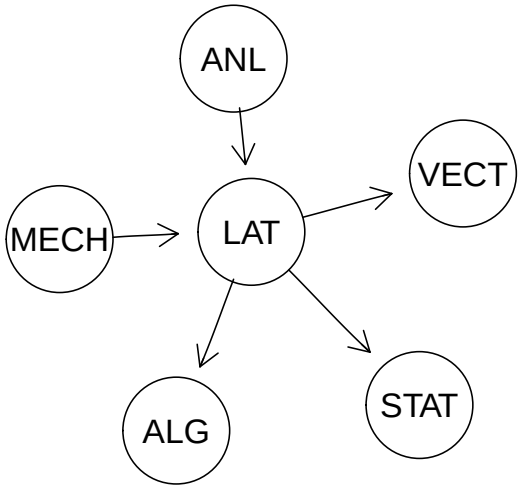
Group B



BN without Latent Grouping



BN with Latent Grouping



The BNs learned from group `A` and group `B` are completely different.

Furthermore, they are both different from the BN learned from the whole data set.

And finally, learning the BN including `LAT` gives a completely different DAG again.

Distributional Assumptions also Matter

We can choose to discretise the `marks` data and include `LAT` when learning the structure of the discrete BN. Again, we obtain a BN whose DAG is completely different from the rest.

```
dmarks = discretize(marks, breaks = 2, method = "interval")
modelstring(hc(data.frame(dmarks, LAT = latent)))
```

```
## [1] "[MECH][ANL][LAT|MECH:ANL][VECT|LAT][ALG|LAT][STAT|LAT]"
```

This BN seems to provide a simple interpretation of the relationships between the topics: the grades in mechanics and analysis can be used to infer which group a student belongs to, and that in turn influences the grades in the remaining topics.

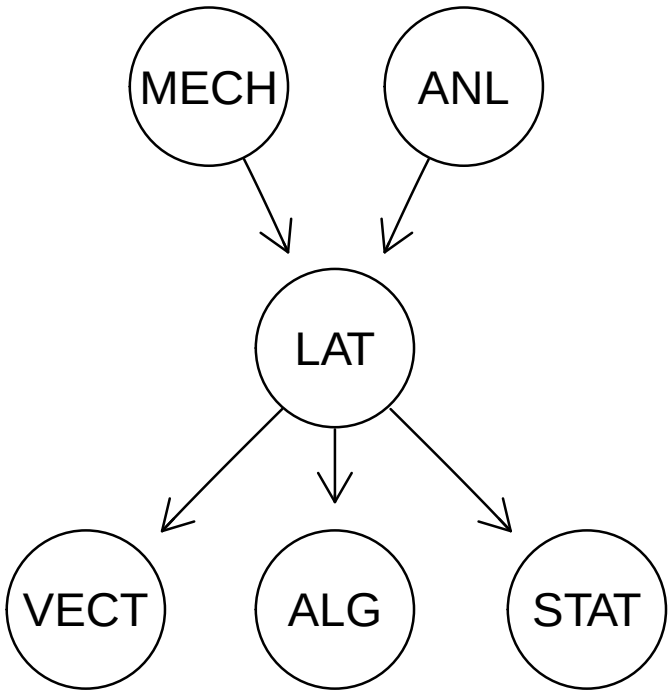
However, if we choose not to discretise:

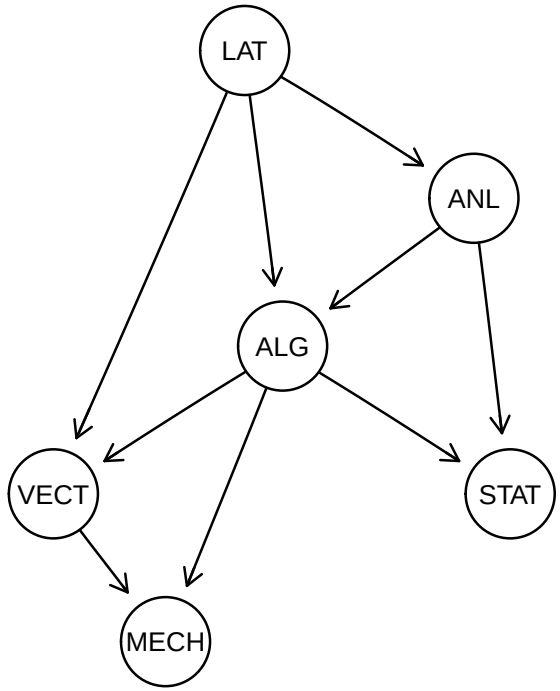
```
modelstring(hc(data.frame(marks, LAT = latent)))
```

```
## [1] "[LAT][ANL|LAT][ALG|ANL:LAT][VECT|ALG:LAT][STAT|ALG:ANL][MECH|VECT:ALG]"
```

With Discretisation, Without Discretisation

```
par(mfrow = c(1, 2))
graphviz.plot(hc(cbind(dmarks, LAT = latent)))
graphviz.plot(hc(cbind(marks, LAT = latent)))
```





We can clearly see that any causal relationship we would have inferred from a DAG learned without taking `LAT` into account would be potentially spurious. And even after including `LAT` the situation is not necessarily clear.

Where Things Go Wrong (I)

Suppose that we have a simple GBN of the form $\mathrm{B} \leftarrow \mathrm{A} \rightarrow \mathrm{C}$:

```
complete.bn = custom.fit(model2network("[A][B|A][C|A]"),
  list(A = list(coef = c("(Intercept)" = 0), sd = 1),
       B = list(coef = c("(Intercept)" = 0, A = 3), sd = 0.5),
       C = list(coef = c("(Intercept)" = 0, A = 2), sd = 0.5))
)
```

In this model we have that `B` is not adjacent to `C` but $\mathrm{B} \not\perp\!\!\!\perp_G \mathrm{C}$ since they are both children of `A`:

```
dsep(complete.bn, "B", "C")
## [1] FALSE
```

However, `B` and `C` are d-separated by `A`, and this implies $\mathrm{B} \perp\!\!\!\perp_P \mathrm{C} \mid \mathrm{A}$.

```
dsep(complete.bn, "B", "C", "A")
## [1] TRUE
```

Where Things Go Wrong (II)

If we generate 100 observations from the complete data we can learn the correct DAG from the data.

```
complete.data = rbn(complete.bn, 100)
modelstring(hc(complete.data))
```

```
## [1] "[A][B|A][C|A]"
```

Now, assume we do not observe `A`; that is, `A` is a latent variable. As a result, `B` and `C` are adjacent in the DAG we learn from the incomplete data.

```
modelstring(hc(complete.data[, c("B", "C")]))
```

```
## [1] "[B][C|B]"
```

If we do not include `A` in the model, there is no way to d-separate `B` and `C`! As a result they end up being linked in this second DAG, as that is the closest we can get to the set of conditional independencies expressed by the true DAG.

Sometimes Things Do Not Go Wrong (I)

However, consider now a GBN of the form $\mathrm{A} \to \mathrm{B} \to \mathrm{C}$:

```
complete.bn = custom.fit(model2network("[A][B|A][C|B]"),
  list(A = list(coef = c("(Intercept)" = 0), sd = 1),
       B = list(coef = c("(Intercept)" = 0, A = 3), sd = 0.5),
       C = list(coef = c("(Intercept)" = 0, B = 2), sd = 0.5))
)
```

Now, `B` depends on `A` and `C` depends on `B`, so by transitivity $\mathrm{A} \not\perp_G \mathrm{C}$ unless we use `B` to d-separate them.

```
dsep(complete.bn, "B", "A")
## [1] FALSE
dsep(complete.bn, "C", "A")
## [1] FALSE
dsep(complete.bn, "C", "A", "B")
## [1] TRUE
```

Sometimes Things Do Not Go Wrong (II)

Again, if we generate 100 observations from the complete data we can learn the correct DAG from the data.

```
complete.data = rbn(complete.bn, 100)
modelstring(hc(complete.data))
```

```
## [1] "[A][B|A][C|B]"
```

The DAG we learn from the incomplete data (omitting `B`) is still consistent with the true DAG as there is still a path leading from `A` to `C`.

```
modelstring(hc(complete.data[, c("A", "C")]))
```

```
## [1] "[A][C|A]"
```

The fact that we do not observe the intermediate node `B` in the causal chain of nodes means that it is now impossible to d-separate `A` and `C` and that `A` appear to be a direct cause of `C`. The DAG simple glosses over the unobserved `B`.

Sometimes Things Do Not Go Wrong (III)

Another situation in which latent variables can have a smaller impact when learning the DAG from the data is for v-structures.

```
complete.bn = custom.fit(model2network("[A][B][C|B:A]"),
  list(A = list(coef = c("(Intercept)" = 0), sd = 1),
       B = list(coef = c("(Intercept)" = 0), sd = 0.5),
       C = list(coef = c("(Intercept)" = 0, A = 3, B = 2), sd = 0.5))
)
complete.data = rbn(complete.bn, 100)
modelstring(hc(complete.data[, c("A", "C")]))
## [1] "[A][C|A]"
modelstring(hc(complete.data[, c("A", "B")]))
## [1] "[A][B]"
```

In this case:

- if one of the parents is a latent variable, we still learn the arc from the other parent correctly;
- if the common child is the latent variable, the parents are not linked by a (spurious) arc.

In Conclusion

- The robustness of causal networks rests on the assumptions that there are no latent variables.
- Learning a DAG from data in the presence of latent variables is likely to result in a DAG that is causally wrong, especially when the DAG includes more than 2-3 nodes or encodes a large set of (in)dependence statements.
- Some patterns of latent variables are more problematic than others: a latent variable that is a common cause for two or more observed nodes represents a confounders and as such always leads to wrong causal networks. Other patterns may be less problematic.
- Latent variables and wrong parametric assumptions interact in determining how wrong the learned DAG is, and it is impossible in practice to determine which is causing a missing/spurious arc.

Causal Inference

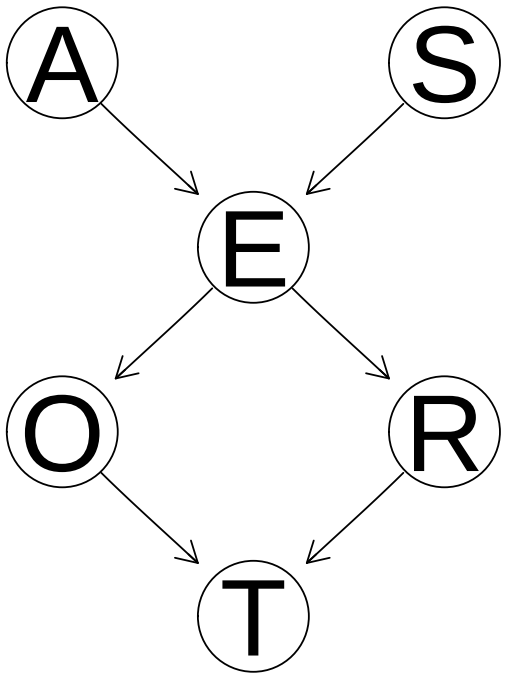
Once we have a causal BN we are happy with, we can again focus on using it to answer relevant questions. In the context of causal networks, we call this causal inference. Compared to the posterior inference we have seen in the previous lecture:

- in probabilistic inference we compute posterior probabilities for events of interest for the observed network;
- in causal inference we compute the effects of interventions for events of interest on a modified network that reflects the interventions.

So in probabilistic inference we are working in an observational setting (look but do not touch), in causal inference we are working in an experimental setting (tweak and see what happens). As a result, causal and probabilistic inference answer different questions; and they will give different probabilities for the same event given the same evidence in general.

The Train Use Survey Revisited

Say that in the original train survey example we collect the data by handing out forms to people chosen at random from the general population; this gives us an observational data set which we can use to learn the BN (from the next lecture).



Say that we are interested in the effect that the residence (R) has on occupation (O), in particular how occupation changes for people living in big cities. The conditional distribution that describes this is:

$$\mathrm{P}(\mathrm{O} \mid \mathrm{R} = \mathrm{big} \mid \mathcal{G}, \Theta).$$

The Train Use Survey Revisited (Posterior)

We can compute the posterior distribution of `O` given `R` = `"big"`.

```
prop.table(table(cpdist(survey.bn, "O", evidence = (R == "big"))))
```

```
##
##   emp  self
## 0.954 0.046
```

This gives us the conditional distribution of the occupation in the part of the general population that lives in a big city. If we compare this with the marginal distribution of `O`

```
prop.table(table(cpdist(survey.bn, "O", evidence = TRUE)))
```

```
##
##    emp   self
## 0.9476 0.0524
```

we see a $\approx 0.07\%$ increase in employees, so the difference from the overall general population is not very big from a practical perspective.

The Train Use Survey Revisited (Causal, I)

Now, we can wonder: if we allow everybody to live and work in a big city (say, by starting a public housing program) how will that affect the occupation status? Note that if we do this we alter the characteristics of the population so the BN will be a valid tool to investigate this. The effects of the intervention (the public housing program) will change

```
coef(survey.bn$R)
```

```
##         E
## R        high  uni
##   small 0.25 0.20
##   big   0.75 0.80
```

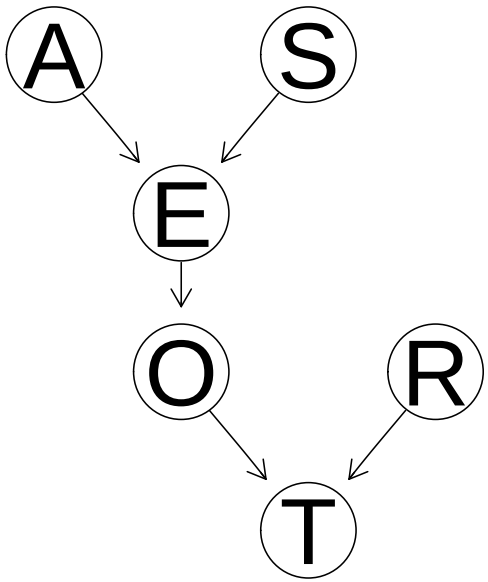
to

```
mut.bn = mutilated(survey.bn, evidence = list(R = "big"))
coef(mut.bn$R)
```

```
## small   big
##     0     1
```

because we give everybody a house in a big city, regardless of their education `E`.

The Train Use Survey Revisited (Causal, II)



We can then compute the effect of this policy on the occupation by calling `cpquery` again but on the mutilated network that incorporates the intervention.

```
prop.table(table(cpdist(mut.bn, "O",
  evidence = TRUE)))
```

```
##
##     emp    self
## 0.9492 0.0508
```

The difference from the general population before the intervention is minimal: this suggests that providing public housing is not a sound policy if the goal is to alter the composition of the workforce.

This approach is called the do-calculus: it rests on the idea that we take complete control of the nodes that are subject to intervention and therefore we remove all their parents from the DAG.

The Train Use Survey Revisited (Causal, III)

It is important to note that interventions need not to be hard interventions (e.g. like hard evidence) but can also be soft interventions (e.g. like soft evidence). For instance, we can consider an alternative housing policy that makes the population spread out to small cities with probability 0.5.

```
mut.bn$R = array(c(0.50, 0.50), dim = 2,
              dimnames = list(R = c("small", "big")))
prop.table(table(cpdist(mut.bn, "O",
  evidence = TRUE)))
```

```
##
##     emp    self
## 0.9486 0.0514
```

Again, not much effect on `O`. Which should not be a surprise since `O` is d-separated from `R` in the mutilated network.

```
dsep(mut.bn, "O", "R")
```

```
## [1] TRUE
```

Causal Inference and Experimental Design

There are three key benefit in this approach to causal inference:

- We can simulate the effect of interventions without the need to carry out a real-world experiment, which is expensive and/or impossible in many cases.
- We can use d-separation to identify which variables produce a change in a target variable if we intervene on them.
- We can re-purpose posterior inference to quantify the effects of (possibly complex) causal interventions.

In situation in which designed experiments are possible, causal inference provides a more intuitive representations of classic experimental design:

- We take control of experimental and blocking factors, which then have no parents in the DAG.
- Randomisation is equivalent to a soft causal intervention.
- Since randomised variables have no parents, causality necessarily flows from them to the target variables

Missing Data

Latent variables are just on kind of missing data:

- A latent variable is a variable which we know nothing about, either its position in the BN or its distribution.
- An unobserved variable is a variable we do not observe, but which we know the position and the distribution of.
- A partially observed variable is a variable for which we observe some but not all the samples (the rest are denoted as `NA`).

The main problems that arise with missing data are:

- How do we learn the structure of BN from the data?
- Given a DAG, how do we estimate the parameters of the local distributions?

The answers to both questions are the Expectation-Maximisation (EM) and Data Augmentation (DA) algorithms.

Classes of Missing Data

There are three classes of missing data:

- Missing completely at random (MCAR): there is no relationship between the missingness of the data and any values, observed or missing. Those missing data points are a random subset of the data.
- Missing at Random (MAR): there is a systematic relationship between the propensity of missing values and the observed data, but not the missing data.
- Missing Not at Random (MNAR): there is a relationship between the propensity of a value to be missing and its values.

MNAR is non-ignorable because the missing data mechanism itself has to be modelled (why the data are missing and what the likely values are). MCAR and MAR are both considered ignorable because we don't have to include any information about the missing data itself when we deal with the missing data.

Representing the Missingness Mechanism

In the context of BNs, each variable has a local distribution $X_i \sim \mathrm{P}(X_i \mid \Pi_{X_i})$ if the data are complete. If X_i has missing data, in the MCAR case

$$X_i \sim \begin{cases} \mathrm{P}(X_i \mid \Pi_{X_i}) & \text{for observed data } X_i^{(O)} \\ \mathrm{P}(X_i \mid \Pi_{X_i}) & \text{for missing data } X_i^{(M)}. \end{cases}$$

The same happens in the MAR case, since the missingness depends on Π_{X_i}. On the other hand, in the MNAR case

$$X_i \sim \begin{cases} \mathrm{P}(X_i^{(O)} \mid \Pi_{X_i}, M) & \text{for observed data } X_i^{(O)} \\ \mathrm{P}(X_i^{(M)} \mid \Pi_{X_i}, M) & \text{for missing data } X_i^{(M)} \end{cases}$$

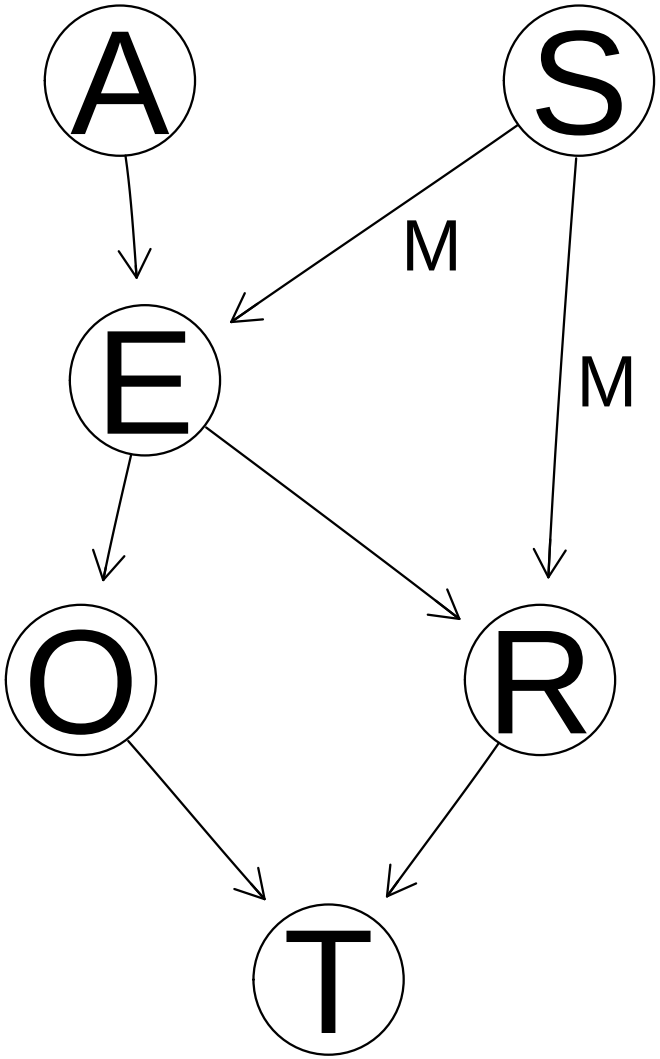
where M is the missingness mechanism. M is non-ignorable because we cannot estimate the local distribution of X_i properly without knowing the missing values in the first place.

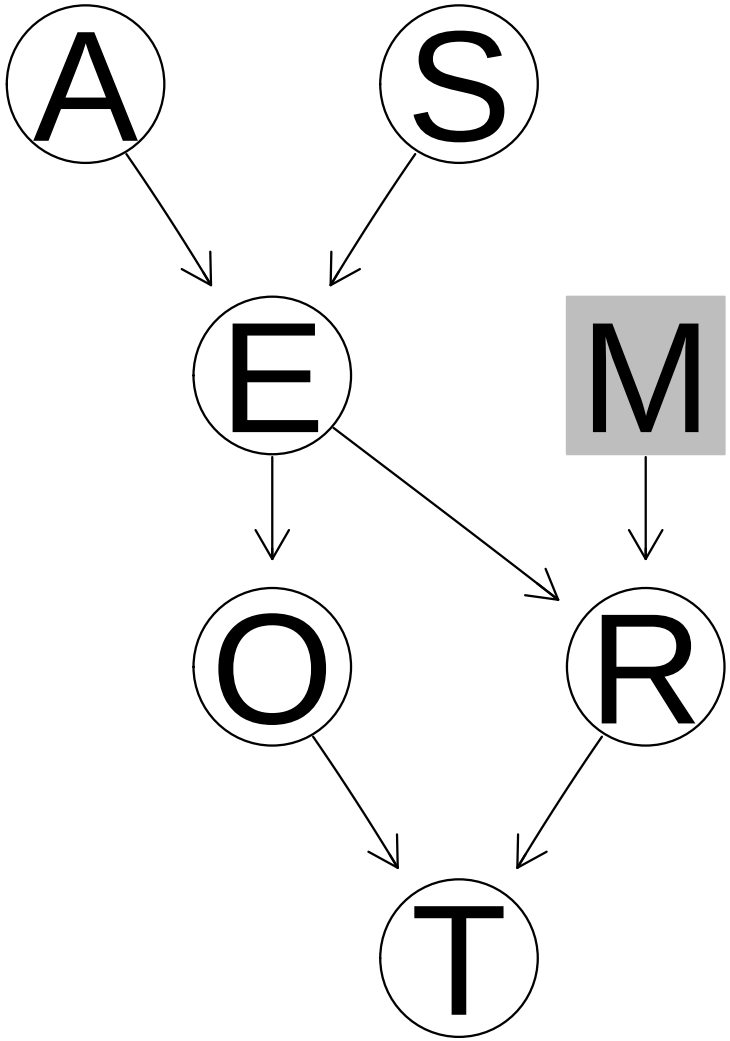
Examples with the Train Use Survey (I)

Since the survey data are collected through a questionnaire, there will be a positive non-response rate for various questions and for the whole questionnaire.

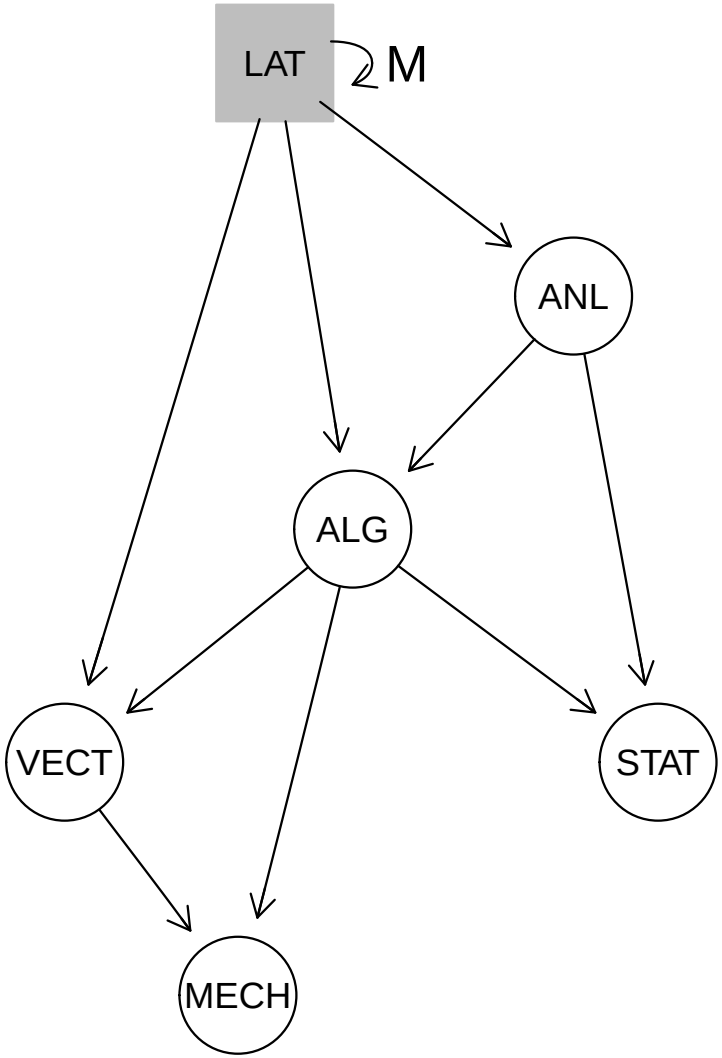
- A MCAR situation may arise when questionnaires are lost in the post – the missingness does not depend on the characteristics of the individual.
- A MAR situation may arise if women refuse to answer some questions in the questionnaire in rates significant higher than men – that is fine since S is observed.
- A MNAR situation may arise if all people in a specific big city do not answer or people of certain social groups do not answer all or part of the questionnaire – we need to introduce M to identify the non-responders.

Examples with the Train Use Survey (II)





The MARKS Example, Revisited



The latent variable in the MARKS example is MCAR, since all the data are missing the missingness mechanism is simply $\mathrm{P}(M \mid \mathrm{LAT}) = 1$.

Which shows that MCAR missingness is not necessarily any less problematic than MAR or MNAR, especially for causal inference!

The Expectation-Maximisation (EM) Algorithm

For a generic statistical quantity θ:

---

1. Choose an initial value $\hat{\theta}_0$ for θ.
2. While $|\hat{\theta}_{j-1} - \hat{\theta}_j| < \varepsilon$, increasing j:
   - 2.1 $\hat{\theta}_j = \hat{\theta}_{j-1}$
   - 2.2 Expectation step: compute the probability distribution over the missing values,

$$\mathrm{P}(X_i^{(M)} \mid X_i^{(O)}, \hat{\theta}_j) = \frac{\mathrm{P}(X_i^{(O)} \mid X_i^{(M)}, \hat{\theta}_j)\,\mathrm{P}(X_i^{(M)} \mid \hat{\theta}_j)}{\int_{X_i^{(M)}} \mathrm{P}(X_i^{(O)} \mid X_i^{(M)}, \hat{\theta}_j)\,\mathrm{P}(X_i^{(M)} \mid \hat{\theta}_j)}$$

   - 2.3 Maximisation step: Compute the new estimate $\hat{\theta}_j$ given $\mathrm{P}(X_i^{(M)} \mid X_i^{(O)}, \hat{\theta}_j)$.
3. Estimate θ with the last $\hat{\theta}_j$.

Properties of the EM Algorithm

- There are both Bayesian and frequentist implementations of EM; the former estimates by maximum posterior and the latter by maximum likelihood.
- EM is guaranteed to converge but
  - it may converge to a local maximum and
  - the convergence can be arbitrarily slow.
- For BNs, convergence is guaranteed only if all steps are carried out with exact inference; the additional variability introduced by approximate inference can derail convergence.

An Example: EM Algorithm, Fixed Structure (I)

Consider a simple BN with two nodes A and B linked by a single arc $A \rightarrow B$, and the following incomplete data

case	1	2	3	4	5	6	7	8	9	10
A	0	0	0	NA	NA	NA	1	1	1	1
B	0	1	1	1	0	0	0	0	1	NA

The parameters of the local distribution of A are

$$\pi_{A,0} = \mathrm{P}(A = 0) \qquad \pi_{A,1} = \mathrm{P}(A = 1)$$

and those of the local distribution of B are

$$\pi_{B,0|A,0} = \mathrm{P}(B = 0 \mid A = 0) \qquad \pi_{B,1|A,0} = \mathrm{P}(B = 1 \mid A = 0)$$
$$\pi_{B,0|A,1} = \mathrm{P}(B = 0 \mid A = 1) \qquad \pi_{B,1|A,1} = \mathrm{P}(B = 1 \mid A = 1).$$

An Example: EM Algorithm, Fixed Structure (II)

1^{st} Maximisation Step: we initialise the parameters of A and B using the complete observations.

$$\pi_{A,0} = 0.5 \qquad \pi_{A,1} = 0.5$$
$$\pi_{B,0|A,0} = 0.333 \qquad \pi_{B,1|A,0} = 0.667$$
$$\pi_{B,0|A,1} = 0.667 \qquad \pi_{B,1|A,1} = 0.333$$

Note that this produces biased estimates if data are MNAR!

1^{st} Expectation Step: we estimate the distributions of the missing data, that is, the (posterior) probabilities of their possible values (with `cpquery()` or `cpdist()` in **bnlearn**).

case	B	$\pi_{A,0\|B}$	$\pi_{A,1\|B}$
4	1	0.667	0.333
5	0	0.333	0.667
6	0	0.333	0.667

case	A	$\pi_{B,0\|A}$	$\pi_{B,1\|A}$
10	1	0.667	0.333

An Example: EM Algorithm, Fixed Structure (III)

2^{nd} Maximisation Step: we can then update the parameter estimates for A and B by summing up the observation indicators and the probabilities of the completions (say, $\pi_{x_i^M}$):

$$\pi = \frac{1}{n} \sum_{x_i} \mathbb{1}_O + \mathbb{1}_M \pi_{x_i^M}$$

The updated parameter estimates are:

$$\pi_{A,0} = 0.433 \qquad \pi_{A,1} = 0.567$$
$$\pi_{B,0|A,0} = 0.385 \qquad \pi_{B,1|A,0} = 0.615$$
$$\pi_{B,0|A,1} = 0.706 \qquad \pi_{B,1|A,1} = 0.294.$$

An Example: EM Algorithm, Fixed Structure (IV)

2^{nd} Expectation Step: using these updated parameter values, we can recompute the distributions of the missing values.

case	B	$\pi_{A,0\|B}$	$\pi_{A,1\|B}$
4	1	0.615	0.385
5	0	0.294	0.706
6	0	0.294	0.706

case	A	$\pi_{B,0\|A}$	$\pi_{B,1\|A}$
10	1	0.706	0.294

And so on, so forth ...

As the number of iterations increases, the parameter updates gradually become smaller and smaller until (after ≈ 4 iterations in this simple example) we can decide EM has converged and stop. We can set a threshold, for instance, by computing the Kullback-Leibler distance between the local distributions at two consecutive iterations.

The EM Algorithm, Unknown Graph Structure

Learning the (CP)DAG of a BN in the presence of missing data (in addition to the parameters) is a problem that is challenging from both a statistical and a computational point of view. Friedman extended the EM algorithm to work for this task, and called the resulting algorithm Structural EM:

---

1. Start with a BN $\mathcal{B}_0$ with an empty DAG $\mathcal{G}_0$ (with no arcs).
2. As long as $\mathcal{B}_i$ is different from B_{i-1}:
   - 2.1 Expectation step: impute the missing data with their posterior expectations or their maximum likelihood estimates using the current BN.
   - 2.2 Maximisation step: learn an updated BN from the completed data.

The MARKS Example, Revisited (I)

```
ldmarks = data.frame(dmarks, LAT = factor(rep(NA, nrow(dmarks)),
            levels = c("A", "B")))

# initialise an empty BN that includes LAT.
imputed = ldmarks
imputed$LAT = sample(factor(c("A", "B")), nrow(dmarks), replace = TRUE)
bn = bn.fit(empty.graph(names(ldmarks)), imputed)
bn$LAT = array(c(0.5, 0.5), dim = 2, dimnames = list(c("A", "B")))

# three iterations of structural EM.
for (i in 1:3) {

  # expectation step.
  imputed = impute(bn, ldmarks, method = "bayes-lw")
  # maximisation step (forcing LAT to be connected to the other nodes).
  dag = hc(imputed, whitelist = data.frame(from = "LAT", to = names(dmarks)))
  bn = bn.fit(dag, imputed, method = "bayes")

}#FOR

modelstring(bn)
## [1] "[LAT][MECH|LAT][VECT|LAT][ALG|LAT][STAT|LAT][ANL|ALG:LAT]"
```

The MARKS Example, Revisited (II)

From Structural EM we get putative class assignments for the students,

```
table(imputed$LAT)
```

```
##
##  A  B
## 70 18
```

and parameters for the CPTs conditional on class.

```
coef(bn$ANL)
```

```
## , , LAT = A
##
##               ALG
## ANL            [14.9,47.5] (47.5,80.1]
##   [8.94,39.5]       0.597       0.105
##   (39.5,70.1]       0.403       0.895
##
## , , LAT = B
##
##               ALG
## ANL            [14.9,47.5] (47.5,80.1]
##   [8.94,39.5]       0.646       0.500
##   (39.5,70.1]       0.354       0.500
```

Imputing Missing Data

Imputing missing values in an incomplete data set implies:

- replacing them with their posterior expectations or maximum a posteriori estimates in a Bayesian setting;
- replacing them with their maximum likelihood estimates, possibly using their parents, in a frequentist setting.

In both cases:

- we need a fully specified BN to do it;
- it is preferable to learn the BN in a Bayesian/frequentist way to perform imputation in a Bayesian/frequentist way;
- all the information needed to make inference on each node is included in its Markov blanket, so we do not need the rest of the BN to impute missing values for that node.

The Data Augmentation (DA) Algorithm

Data augmentation is similar in spirit to EM, but it is a stochastic MCMC algorithm that uses sampling instead of expectation.

---

1. Choose an initial value $\hat{\theta}_0$ for θ.
2. Until convergence, increasing j:
   - 2.1 Imputation step: Sample θ_j from $\mathrm{P}(\theta_{j-1} \mid X_i^{(O)})$, and then sample $X_i^{(M)}$ from $\mathrm{P}(X_i^{(M)} \mid \theta_{j-1}, X_i^{(O)})$.
   - 2.2 Posterior step: Update the posterior

$$\mathrm{P}(\theta_j \mid X_i^{(O)}) = \int_{X_i^{(M)}} \mathrm{P}(\theta_j \mid X_i^{(O)}, X_i^{(M)}).$$

$\mathrm{P}(\theta_j \mid X_i^{(O)}) = \int_{X_i^{(M)}}$ is posterior distribution of the parameters given the observed data averaged over the missing data.

Predicting New Observations

One of the tasks statistical models are commonly used for is prediction: we have new samples that are only partially observed (or for which we assume we know the values they take for some variables), and we would like to have principled estimates of their values for the variables we do not observe. Much like missing data imputation:

- we need a fully specified BN to do it;
- it is preferable to learn the BN in a Bayesian/frequentist way to perform imputation in a Bayesian/frequentist way;
- all the information needed to make inference on each node is included in its Markov blanket, so we do not need the rest of the BN to impute missing values for that node.

The crucial difference is that we use the partially observed data to learn the BN, whereas the new data which we would like to predict are independent of the BN we use for prediction.

bnlearn: `predict()` New Observations

bnlearn implements a `predict()` method for fitted BNs.

```
pred.maxlik = predict(marks.bn, node = "ALG", new.students, method = "parents")
```

It takes the following arguments:

- the fitted BN;
- the `node` to predict values for;
- the observed data for the new observations;
- the prediction method, either `parents` for frequentist predictions or `bayes-lw` for Bayesian predictions.

The frequentist prediction above predicts the most likely mark in `ALG` given its parents for 30 new students; that is, the prediction uses only the local distribution of `ALG`.

bnlearn: Frequentist and Bayesian Predictions

However, this does not work very well because `ALG` has no parents: every prediction is just the mean mark for `ALG`.

```
cor(new.students$ALG, pred.maxlik)
```

```
## [1] NA
```

Bayesian posterior predictions perform better because they use all the nodes that are provided in `new students`: the mean difference between observed and predicted `ALG` marks is ≈ 4 marks.

```
pred.bayes = predict(marks.bn, "ALG", new.students, method = "bayes-lw")
mean(abs(new.students$ALG - pred.bayes))
```

```
## [1] 4.12
```

Predicting using just the nodes in the Markov blanket of `ALG` provides predictions identical (up to simulation noise) to those above, as expected.

```
pred.mb = predict(marks.bn, "ALG", new.students, method = "bayes-lw",
                  from = mb(marks.bn, "ALG"))
mean(abs(pred.bayes - pred.mb))
```

```
## [1] 0.372
```

Predictive Accuracy Decreases with Graph Distance

Computing predictions from nodes outside of the Markov blanket is certainly possible; Bayesian posterior predictions can predict any node from any other node(s). However, predictions become less and less accurate the farther the nodes we predict from are from the target node.

```
modelstring(marks.dag)
```

```
## [1] "[ALG][ANL|ALG][VECT|ALG][MECH|ALG:VECT][STAT|ALG:ANL]"
```

```
pred.mb = predict(marks.bn, "STAT", new.students, method = "bayes-lw",
                  from = mb(marks.bn, "STAT"))
mean(abs(new.students$STAT - pred.mb))
```

```
## [1] 11.4
```

Predictive accuracy for STAT is not good when using the nodes in the Markov blanket (ALG and ANL); it get worse with nodes outside of the Markov blanket.

```
pred.far = predict(marks.bn, "STAT", new.students, method = "bayes-lw",
                   from = c("VECT", "MECH"))
mean(abs(new.students$STAT - pred.far))
```

```
## [1] 13.3
```

Predicting from Multiple Models: Ensembles

A tried-and-tested way to improve predictive accuracy is to predict from an ensemble of multiple models instead of just a single model.
Intuitively, enough models will provide accurate predictions for each new observations to make the consensus prediction accurate. Consider three models each with classification accuracy 0.70 will classify correctly if at least two are correct, which happens with probability

$$0.7^3 + 3 \times (0.7^2 * 0.3) \approx 0.784.$$

Assuming that models are independent of each other, the more models the better: with five models the probability above increases to ≈ 0.837.

The problem is, how to produce models that are independent from each other? And how do we combine predictions?

bnlearn: Ensembles and Cross-Validation (I)

As long as we use BNs (or any kind of model, really) learned from data, those models will never be independent. A common way to obtain an ensemble of models that are at least moderately different is to learn them on multiple resampled data sets to introduce perturbations in the estimation process.

In a way, this naturally happens when we evaluate predictive accuracy with cross-validation. For instance, if we take the first 40 students in MARKS to be the `new.students` and we learn a BN from the rest, we reach a mean difference between observed and predicted `STAT` marks of ≈ 16.

```
new.students = marks[1:40, ]
old.students = marks[-(1:40), ]
single = bn.fit(hc(old.students), old.students)
pred.single = predict(single, "STAT", new.students, method = "bayes-lw")
mean(abs(new.students$STAT - pred.single))
```

```
## [1] 16.1
```

bnlearn: Ensembles and Cross-Validation (II)

If we perform cross-validation with `bn.cv()`, we can:

1. extract the BNs that were fitted withdrawing each fold;

```
kfold = bn.cv(old.students, "hc", k = 10)
ensemble = lapply(kfold, `[[`, "fitted")
```

2. predict each new student from each model;

```
pred.ensemble = sapply(ensemble, predict, node = "STAT",
                       data = new.students, method = "bayes-lw")
```

3. average the predictions;

```
pred.ensemble = rowMeans(pred.ensemble)
```

4. compute the predictive accuracy.

```
mean(abs(new.students$STAT - pred.ensemble))
```

```
## [1] 10.6
```

The result is much more precise, with a mean difference of ≈ 10.5; and that even though BNs from cross-validation are fairly similar and even though we use just 10 BNs.

Bootstrap Aggregation: Bagging

A second approach to resample data in order to produce a set of diverse models is bootstrap aggregation or bagging.

---

1. For $b = 1, 2, \ldots, B$:
   - 1.1 sample a new data set $\mathcal{D}_b^*$ from the original data $\mathcal{D}$ using nonparametric bootstrap;
   - 1.2 learn the the BN $\mathcal{G}_b = (\mathbf{V}, A_b)$ from $\mathcal{D}_b^*$;
   - 1.3 predict the values $\tilde{T}_b$ of the target variable T in the new observations using $\mathcal{G}_b$.
2. Compute the consensus prediction $\tilde{T}$ from the $\tilde{T}_b$.

The literature provides many options for computing the consensus predictions, mainly involving introducing weights for the $\mathcal{G}_b$ and more advanced schemes than mean or majority vote to aggregate the $\tilde{T}_b$.

bnlearn: Ensembles and Bagging

A simple implementation of the first step in **bnlearn** is a follows.

```
bagging.iteration = function(old, new, target) {

  # step 1.1: resampling.
  Db = old[sample(nrow(old), replace = TRUE), ]
  # step 1.2: learn the BN.
  Gb = bn.fit(hc(Db), Db)
  # step 1.3: predict.
  predict(Gb, node = target, data = new, method = "bayes-lw")

}#BAGGING.ITERATION
```

Then we can compute the average predictions as we did before for `bn.cv()`.

```
# step 2: average the predictions.
Tb = replicate(100, bagging.iteration(old = old.students,
       new = new.students, target = "STAT"))
mean(abs(new.students$STAT - rowMeans(Tb)))
```

```
## [1] 15.9
```

Summary

- BNs are defined as probabilistic models, but it is possible to use them as causal models with great care. Additional assumptions are required and latent variables are a constant source of difficult-to-debug problems.
- Inference is different for causal BNs: it focuses on simulating interventions and measuring their effects as opposed to compute conditional probabilities of events for the original BN.
- A related problem in learning BNs and performing inference is dealing with missing data by applying algorithms such as EM to these tasks.
- BNs provide a nice way to represent and reason about different patterns of missingness.
- BNs can also be used to impute missing values or predict values for new observations in a variety of ways; as usual using an ensemble of multiple, diverse BNs provides better accuracy than using a single BN.

Fundamentals of Structure Learning

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 $\mathcal{D}$ and if we denote the parameters of the global distribution as $\mathbf{X}$ with Θ, 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 practise 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

Most tasks related to both learning and inference are NP-hard (they cannot be solved in polynomial time in the number of variables). 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.
In Bayesian networks, for example, 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 specify the parameters of local distribution, or to constrain 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 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

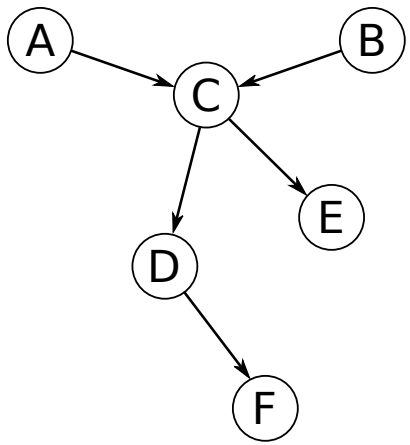
- There must be a one-to-one correspondence between the nodes in the DAG and the random variables in $\mathbf{X}$; there must not be multiple nodes which are deterministic functions of a single variable.
- All the relationships between the variables in $\mathbf{X}$ must be conditional independencies, 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 have uniquely determined Markov blankets and, therefore, 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 Bayesian networks.

Classes of Structure Learning Algorithms from Data

Despite the (sometimes confusing) variety of theoretical backgrounds and terminology they 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: each candidate DAG is assigned a score reflecting its goodness of fit, which is then taken as an objective function to maximise.
- 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



CPDAG $\Longleftarrow$ Graphical separation $\Longleftarrow$ Conditional independence tests

$$
\begin{aligned}
&A \perp\!\!\!\perp_G B \\
&A \perp\!\!\!\perp_G D \mid C \\
&B \perp\!\!\!\perp_G D \mid C \\
&A \perp\!\!\!\perp_G E \mid C \\
&B \perp\!\!\!\perp_G E \mid C \\
&D \perp\!\!\!\perp_G E \mid C \\
&C \perp\!\!\!\perp_G F \mid D \\
&\dots
\end{aligned}
$$

$$
\begin{aligned}
&p_{A \perp\!\!\!\perp B} < \alpha \\
&p_{A \perp\!\!\!\perp E} < \alpha \\
&\vdots \\
&p_{A \perp\!\!\!\perp E|C} < \alpha \\
&p_{B \perp\!\!\!\perp E|D} < \alpha \\
&p_{A \perp\!\!\!\perp B|C} > \alpha \\
&p_{E \perp\!\!\!\perp F|D} < \alpha \\
&\vdots \\
&p_{F \perp\!\!\!\perp E|CD} < \alpha
\end{aligned}
$$

The mapping between edges and conditional independence relationships lies at the core of BNs; therefore, one way to learn the structure of a BN is to check which such relationships hold using a suitable conditional independence test. Such an approach results in a set of conditional independence constraints that identify a single equivalence class.

Assuming a Perfect Map

BNs are defined as I-maps so

$$\mathbf{A} \perp\!\!\!\perp_G \mathbf{B} \mid \mathbf{C} \Longrightarrow \mathbf{A} \perp\!\!\!\perp_P \mathbf{B} \mid \mathbf{C}.$$

However, constraint-based algorithms treat them as perfect maps since they do

$$\mathbf{A} \perp\!\!\!\perp_P \mathbf{B} \mid \mathbf{C} \Longleftrightarrow \mathbf{A} \perp\!\!\!\perp_G \mathbf{B} \mid \mathbf{C}.$$

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

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

The 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.

Other Constraint-based algorithms

- Peter & Clark (PC): a true-to-form implementation of the Inductive Causation algorithm, specifying only the order of the conditional independence tests. Starts from a saturated network and performs tests gradually increasing the number of conditioning nodes.
- Grow-Shrink (GS) and Incremental Association (IAMB) variants: these algorithms learn the Markov blanket of each node to reduce the number of tests required by the Inductive Causation algorithm. Markov blankets are learned using different forward and step-wise approaches; the initial network is assumed to be empty (i.e. not to have any edge).
- Max-Min Parents & Children (MMPC): uses a minimax approach to avoid conditional independence tests known *a priori* to accept the null hypothesis of independence.
- Hiton-PC (HITON-PC): currently the most scalable choice, it uses a first pass based on marginal tests followed by a backward selection.

Conditional Independence Tests: Discrete Variables

Conditional independence tests used to learn DBN are functions of the observed frequencies $\{n_{ijk}, i = 1, \ldots, R, j = 1, \ldots, C, k = 1, \ldots, L\}$ for the random variables X and Y and all the configurations of the conditioning variables $\mathbf{Z}$. Classic choices are:

- mutual information/log-likelihood ratio

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

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

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

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

Conditional Independence Tests: Gaussian Variables

Conditional independence tests used to learn GBNs are functions of the partial correlations $\rho_{XY|\mathbf{Z}}$ that are used as proxies for the cells of $\Omega = \Sigma^{-1}$. Classic choices are:

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

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

  and distributed as a Student's t with $n - |\mathbf{Z}| - 2$ degrees of freedom;

- Fisher's Z test, a transformation of $\rho_{XY|\mathbf{Z}}$ with an asymptotic normal distribution and defined as

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

  where n is the number of observations and $|\mathbf{Z}|$ is the number of nodes belonging to $\mathbf{Z}$.

Conditional Independence Tests: Conditional Gaussian (I)

It is more complicated to specify tests for CLGBNs, because not all triplets $(X, Y, \mathbf{Z})$ can be directly represented as a single local distribution. 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

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

  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 $\mathbf{Z} = \{Z_{c_1}, \ldots, Z_{c_l}, Z_{d_1}, \ldots, Z_{d_m}\}$ 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 an unrolled chain of log-likelihood ratios that can be treated as a mutual information test.

Conditional Independence Tests: Permutations

Asymptotic tests require a sample size large enough for the null distribution to converge to its asymptotic behaviour. We can use permutation tests instead:

1. Compute the test statistic $\hat{t}$ on the original $(X, Y, \mathbf{Z})$.
2. For $b = 1, \ldots, B$:
   - 2.1 permute Y while keeping X and $\mathbf{Z}$ fixed, to obtain a new sample $(X, Y_b^*, \mathbf{Z})$ from the null distribution in which $X \perp\!\!\!\perp_P Y_b^* \mid \mathbf{Z}$.
   - 2.2 Compute the test statistic $\hat{t}_b$ on $(X, Y_b^*, \mathbf{Z})$.
3. The p-value of the test as

$$\frac{1}{B} \sum_{b=1}^{B} \mathbb{1}\{\hat{t} > t_b\}$$

for one-tailed tests and

$$\frac{1}{B} \sum_{b=1}^{B} \mathbb{1}\{|\hat{t}| > |t_b|\}$$

for two-tailed tests.

Conditional Independence Tests: Shrinkage

An alternative is to regularise the test statistic by shrinking it towards a regular target distribution. For instance, in the case of a covariance matrix we estimate $\tilde{\Sigma}$ as a linear combination of the maximum likelihood estimator $\hat{\Sigma}$ and a target distribution with a diagonal covariance matrix T:

$$\tilde{\Sigma} = \lambda T + (1 - \lambda)\hat{\Sigma}, \qquad \lambda \in [0, 1].$$

λ can be estimated in closed form as

$$\lambda^* = \frac{\sum_{i=1}^{k} \sum_{j=1}^{k} \mathrm{VAR}(\hat{\sigma}_{ij}) - \mathrm{COV}(\hat{\sigma}_{ij}, t_{ij})}{\sum_{i=1}^{k} \sum_{j=1}^{k} (t_{ij} - \hat{\sigma}_{ij})^2}.$$

The modified $\tilde{\Sigma}$ can then be used to compute the (partial) correlations used in the conditional independence tests.

A similar approach can be used for categorical data and mutual information.

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:                     1
##     directed arcs:                       4
##   average markov blanket size:           1.75
##   average neighbourhood size:            1.25
##   average branching factor:              0.50
##
##   learning algorithm:                    Semi-Interleaved HITON-PC
##   conditional independence test:         Mutual Information (disc.)
##   alpha threshold:                       0.05
##   tests used in the learning procedure:  55
##   optimized:                             TRUE
```

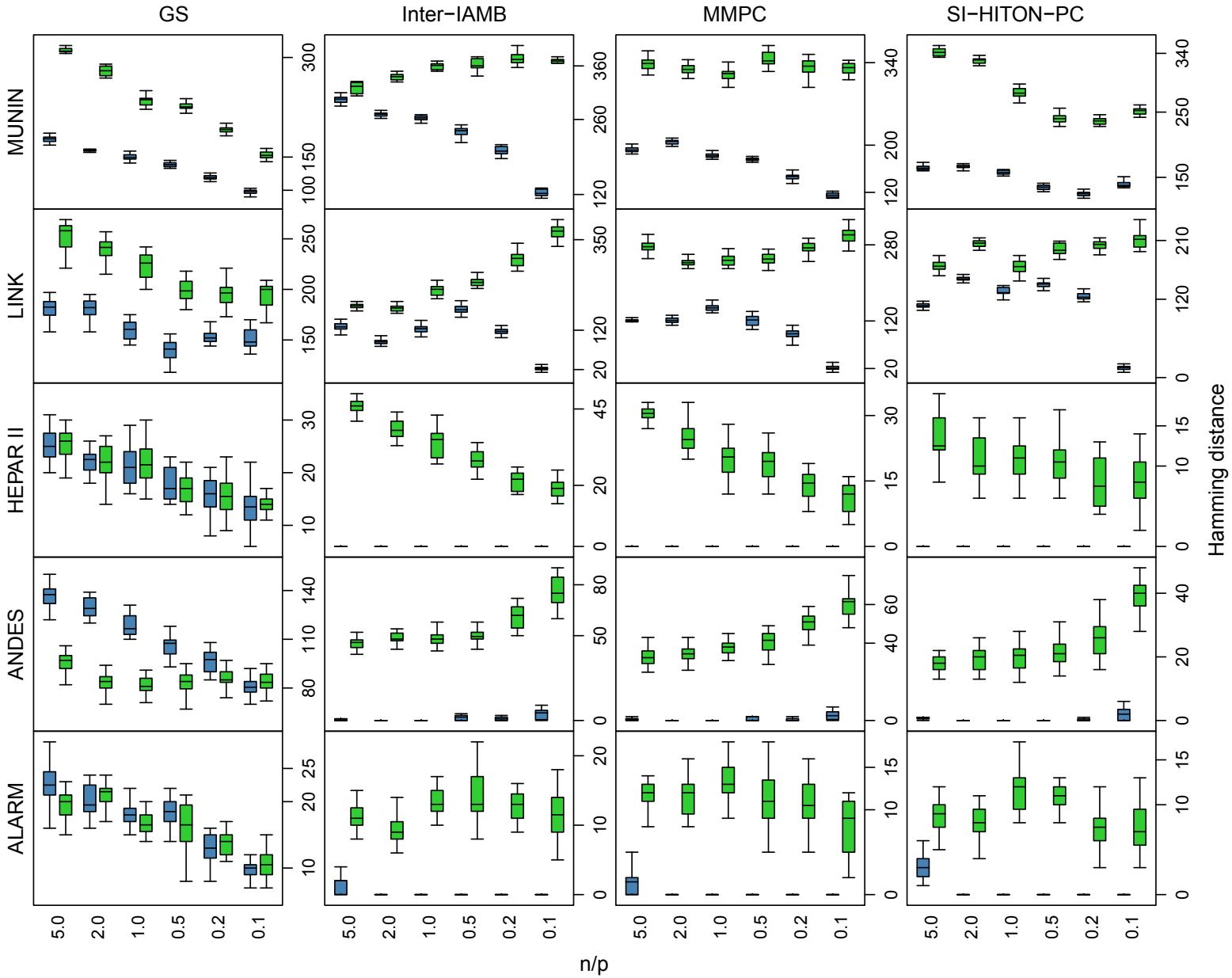
bnlearn: Parameters and Tuning Arguments

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

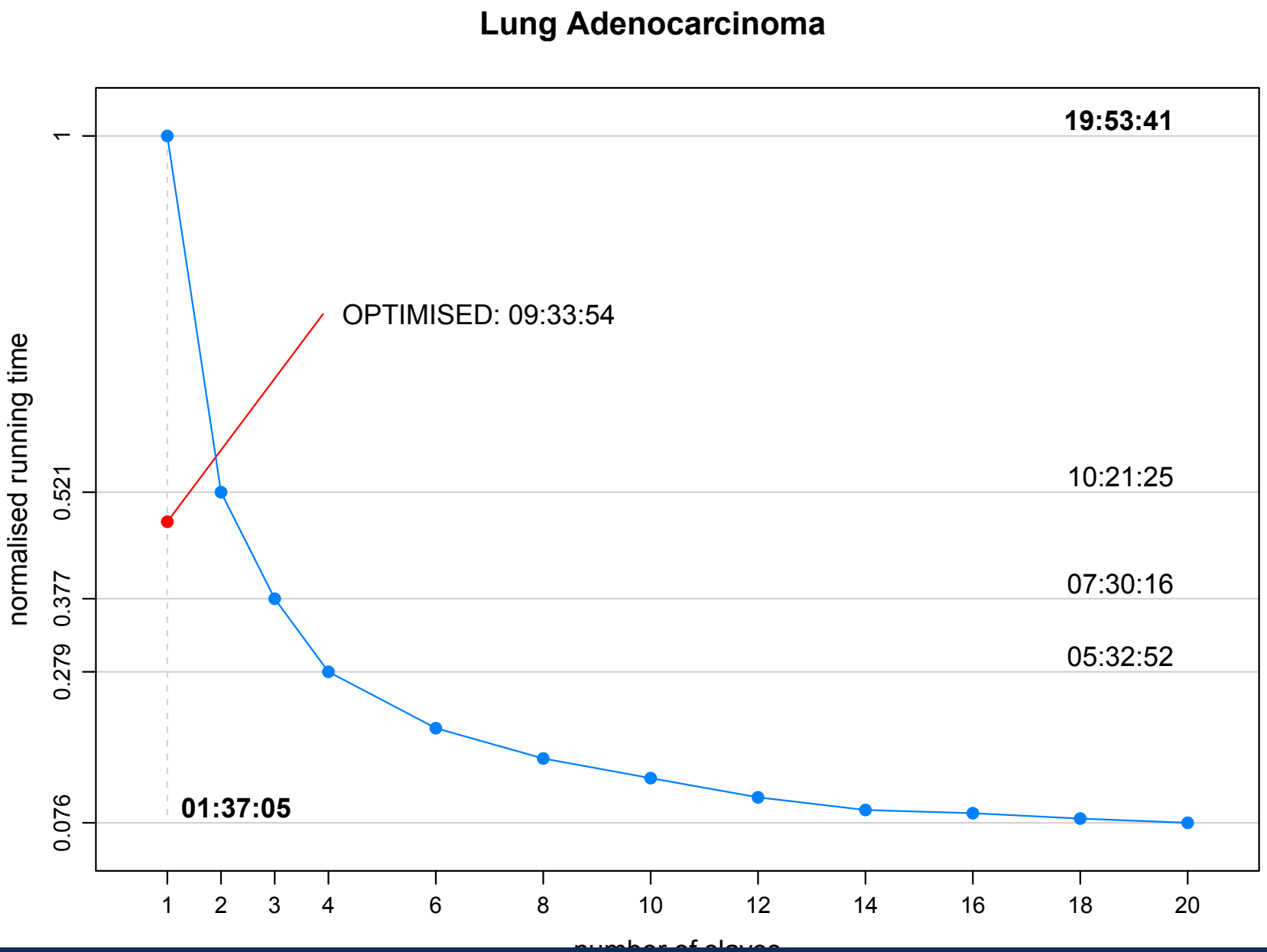
- the first argument is the data.
- `cluster`: a cluster object from the **parallel** package to perform steps in parallel for different nodes.
- `test`: the label of the test statistic.
- `alpha`: the type-I error threshold for the individual conditional independence tests (i.e. without any multiplicity adjustment).
- `B`: number of permutations to use in permutation tests.
- `optimized`: use (or not) backtracking to roughly halve the number of tests by using the symmetry of Markov blankets and neighbours.
- `skeleton`: whether to learn just the skeleton instead of the CPDAG.
- `debug`: whether to print out the steps performed by the algorithm.

UNIVERSITY OF OXFORD

Using Backtracking Is Not Such A Good Idea...

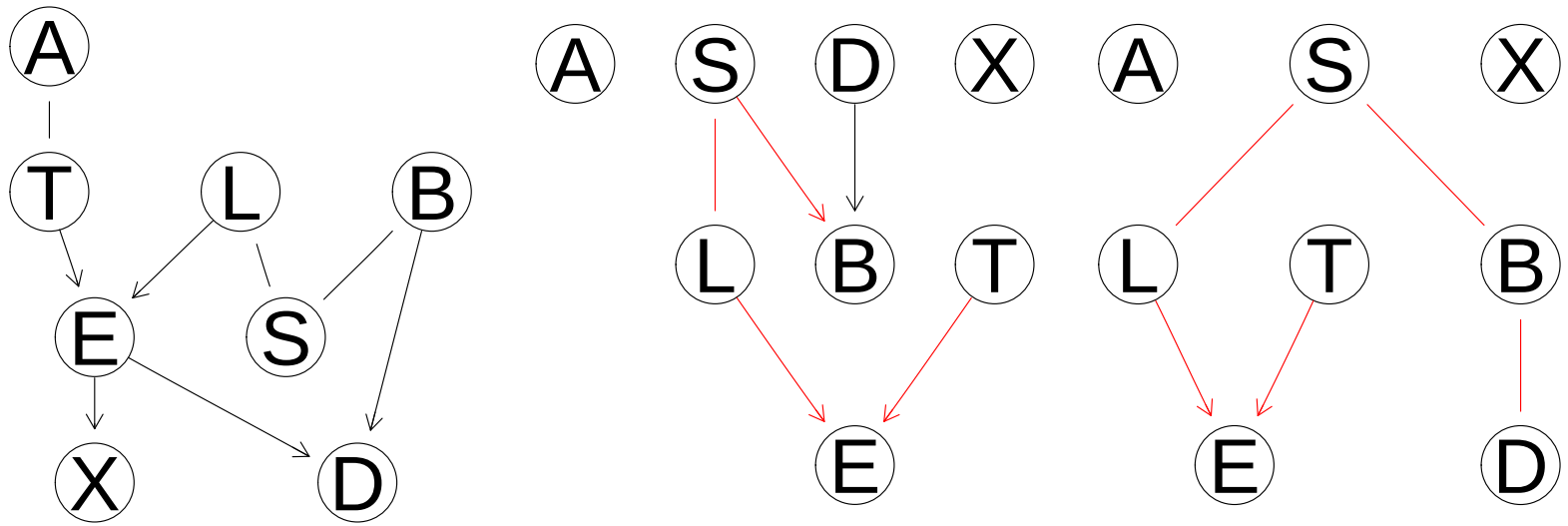


... Because Parallel Computing is Safer and Faster



bnlearn: With and Without Backtracking

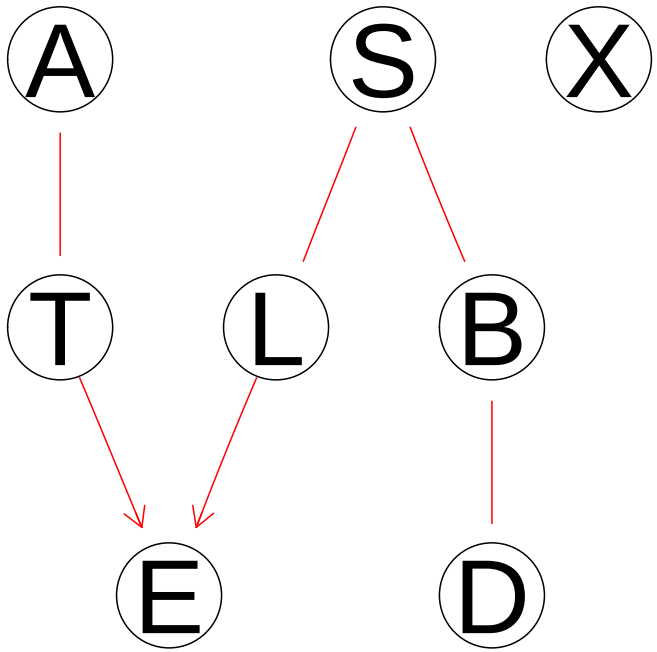
```
par(mfrow = c(1, 3))
true.dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
graphviz.plot(cpdag(true.dag))
graphviz.plot(cpdag, highlight = list(arcs = arcs(cpdag(true.dag))), )
cpdag2 = si.hiton.pc(asia, undirected = FALSE, optimized = FALSE)
graphviz.plot(cpdag2, highlight = list(arcs = arcs(cpdag(true.dag))))
```



The reason why `si.hiton.pc()` cannot learn the CPDAG is that there are many nodes with 0s and 1s in the CPTs, which breaks the convergence of the mutual information to the χ^2 distribution.

bnlearn: Permutation Tests Do A Little Better

```
cpdag2 = si.hiton.pc(asia, test = "mc-mi", undirected = FALSE,
           optimized = FALSE)
graphviz.plot(cpdag2, highlight = list(arcs = arcs(cpdag(true.dag))))
```



There is only one arc missing; all the reference DBNs are impossible to learn perfectly at any reasonable sample size, so this is a pretty good result.

bnlearn: The Debugging Output (I)

```
debugging.output = capture.output(
  si.hiton.pc(asia, test = "mc-mi", undirected = FALSE, optimized = 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 0.0046 ."
##  [7] "  * nodes that will be disregarded from now on."
##  [8] "    > S has p-value 0.131 ."
##  [9] "    > L has p-value 0.368 ."
## [10] "    > B has p-value 0.0616 ."
## [11] "    > E has p-value 0.0758 ."
## [12] "    > X has p-value 0.182 ."
## [13] "    > D has p-value 0.0858 ."
## [14] "  @ T accepted as a parent/children candidate ( p-value: 0.0046 )."
## [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("^\\*", 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] "* backward phase for candidate node A ."
## [11] "* forward phase for node L ."
## [12] "* backward phase for candidate node B ."
## [13] "* backward phase for candidate node E ."
## [14] "* backward phase for candidate node X ."
## [15] "* backward phase for candidate node D ."
```

bnlearn: The Debugging Output (III)

```
head(grep("^\\*|\\s*@", debugging.output, value = TRUE), n = 20)
```

```
##  [1] "* forward phase for node A ."
##  [2] "  @ T accepted as a parent/children candidate ( p-value: 0.0046 )."
##  [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] "* backward phase for candidate node A ."
## [15] "  @ A accepted as a parent/children candidate ( p-value: 0.0056 )."
## [16] "* forward phase for node L ."
## [17] "  @ S accepted as a parent/children candidate ( p-value: 0 )."
## [18] "* backward phase for candidate node B ."
## [19] "* backward phase for candidate node E ."
## [20] "  @ E accepted as a parent/children candidate ( p-value: 0 )."
```

bnlearn: Learning Markov Blankets and Neighbourhoods

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.

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

```
##
##  Mutual Information (disc., MC)
##
## data:  S ~ E | L
## mc-mi = 4e-06, Monte Carlo samples = 5000, p-value = 0.9
## 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 $\texttt{x} \perp\!\!\!\perp_P \texttt{y} \mid \texttt{z}$ where `z` can be either absent (for marginal tests) or a vector of labels (to condition on one or more variables).

Pros & 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 underperform.
  - Permutation tests on the other hand 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 is slower than score-based and hybrid algorithms.
- 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.

Score-based Structure Learning Algorithms

The dimensionality of the space of graph structures makes 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, i.e.

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

such as

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

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

if each comparison involves structures differing 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 $Score_{\mathcal{G}}$.
5. Return the directed acyclic graph $\mathcal{G}$.

DBNs: The Bayesian Dirichlet Marginal Likelihood

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

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

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

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

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

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 implementation of BD assumes $\alpha_{ijk} = \alpha/(r_i q_i)$, $\alpha_i = \alpha$ and is known as the Bayesian Dirichlet equivalent uniform (BDeu) marginal likelihood. 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 $\mathcal{D}$.
- There are formal proofs of all this.

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 $\mathcal{D}$.

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

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

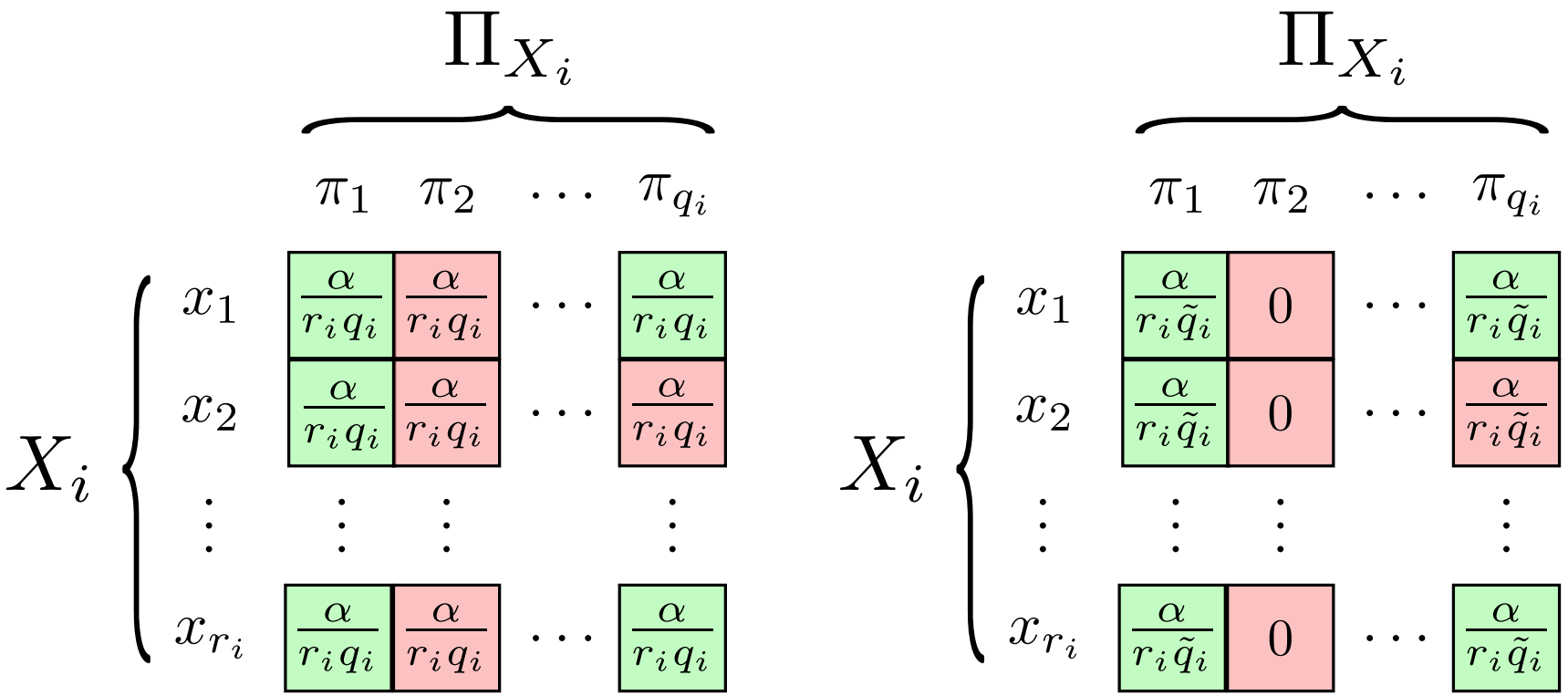
So the effective imaginary sample size decreases as the number of unobserved parents configurations increases, and the MAP estimates of π_{ijk} gradually converge to the ML and favour overfitting.

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

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

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 also make very popular scores for DBNs, GBNs and CLGBNs. AIC tends to overfit a lot, while BIC tends to underfit a bit but it often used an approximation to $\mathrm{P}(\mathcal{D} \mid \mathcal{G})$. For DBNs, the log-likelihood and the number of parameters associated with a local distribution are:

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

for GBNs:

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

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))
```

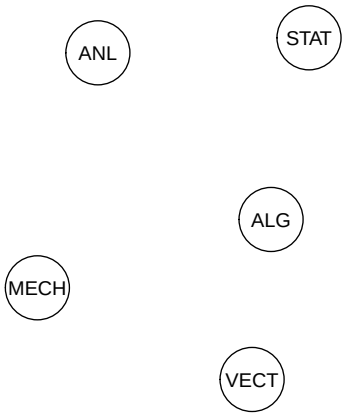
```
## tp fp fn
##  3  3  3
```

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

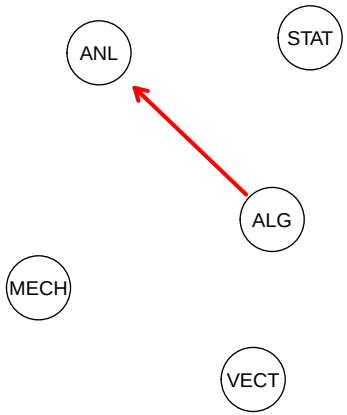
```
## tp fp fn
##  6  0  0
```

The Hill-Climbing Algorithm (MARKS)

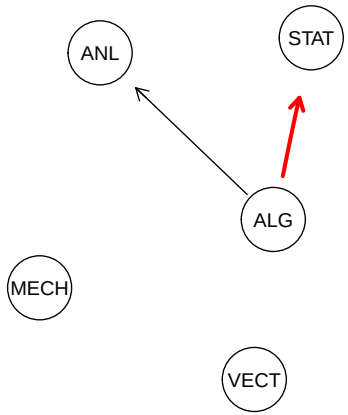
Initial BIC score: −1807.528



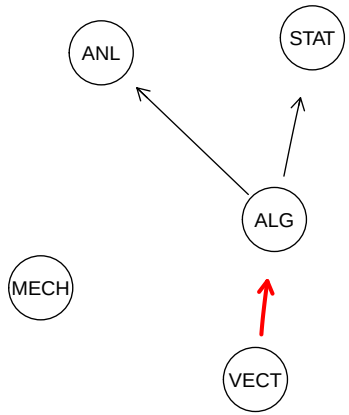
Current BIC score: −1778.804



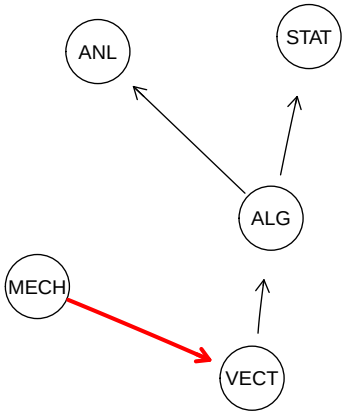
Current BIC score: −1755.383



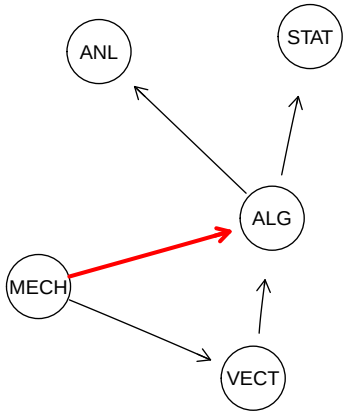
Current BIC score: −1737.176



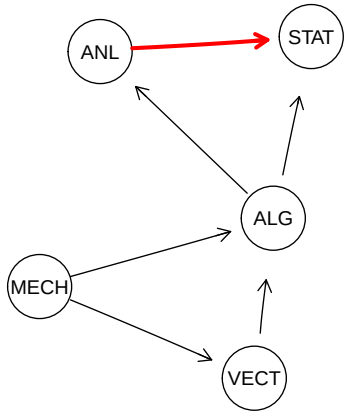
Current BIC score: −1723.325



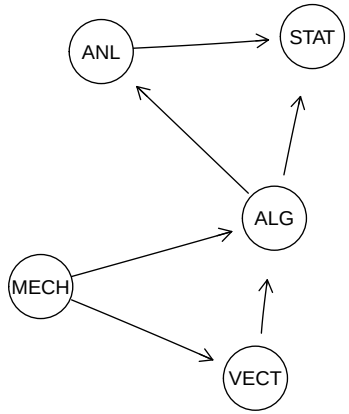
Current BIC score: −1720.901



Current BIC score: −1720.150



Final BIC score: −1720.150



bnlearn: Comparing Networks

- `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|doing)", 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] "* doing a random restart, 9 of 10 left."
##  [9] "* best operation was: adding E -> X ."
## [10] "* best operation was: adding E -> D ."
```

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, changing a number of its arcs to created 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: -11237 "
## [11] "* current score: -11106 "
## [12] "* current score: -11101 "
## [13] "* current score: -11096 "
## [14] "* doing a random restart, 8 of 10 left."
```

bnlearn: Hill-Climbing With Preseeded Networks

Another way of 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|A][E|A][D|S][L|T][B|S:L][X|S:B] "
```

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

Other Score-based Algorithms

- Greedy Equivalent Search: hill-climbing over equivalence classes rather than graph structures; the search space is much smaller.
- Tabu Search: a modified hill-climbing that keeps a list of the last k structures visited (the *tabu list*), and returns only if they are all worse than the current one.
- Genetic Algorithms: they perturb (*mutation*) and combine (*crossover*) features through several generations of structures, and keep the ones leading to better scores. Inspired by Darwinian evolution.
- Simulated Annealing: again similar to hill-climbing, but not looking at the maximum score improvement at each step. Very difficult to use in practice because of its tuning parameters.

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 dec
##  [9] "* best operation was: reversing S -> L ."
## [10] "* network score did not increase (for 2 times), looking for a minimal dec
```

Pros & Cons of Score-based Algorithms

- Convergence to the global maximum (i.e. 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 nobody can 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.

bnlearn: rsmax2()

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

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

Its main arguments are:

- test: the conditional independence test to use in the restrict phase;
- score: score function to use in the maximise phase;
- 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.

bnlearn: mmhc()

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:                Max-Min Parent Children "
## [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:                Max-Min Parent Children "
## [4] "  score-based method:                     Hill-Climbing "
```

Pros & 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. We 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
##  5  3  1
```

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

```
## [1] 3
```

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.

Advanced Structure Learning, Parameter Learning

The DAGs and the Distributions

BN literature focuses mostly on (the parameters of) local probability distributions. However:

- Comparing models learned with different algorithms is difficult, because they maximise different scores, use different estimators for the parameters, work under different sets of hypotheses, etc.
- Unless the true global probability distribution is known it is difficult to assess the uncertainty of the estimated models.
- The few available measures of structural difference are completely descriptive in nature (e.g. the Structural Hamming distance), and are difficult to interpret.
- When learning causal graphical models often we are looking for particular patterns of arcs in the DAG.

Looking for a Solution

Focusing on the DAGs $\mathcal{G}$ sidesteps some of these problems and is useful in structure learning as well, since

$$\mathrm{P}(\mathcal{G} \mid \mathcal{D}) \propto \mathrm{P}(\mathcal{G})\,\mathrm{P}(\mathcal{D} \mid \mathcal{G}).$$

So:

0. We need to know more about the properties of priors $\mathrm{P}(\mathcal{G})$ and posteriors $\mathrm{P}(\mathcal{G} \mid \mathcal{D})$ over the space of DAGs, preferably as a function of their arc sets, say $\mathrm{P}(\mathcal{G}(\mathcal{E}))$ and $\mathrm{P}(\mathcal{G}(\mathcal{E}) \mid \mathcal{D})$ with $\mathcal{E} = \{(v_i, v_j), i \neq j\}$.

And then:

1. It would be good to have measures of spread for $\mathcal{G}$, to assess the noisiness of $\mathrm{P}(\mathcal{G}(\mathcal{E}) \mid \mathcal{D})$ and the informativeness of $\mathrm{P}(\mathcal{G}(\mathcal{E}))$.
2. It would be interesting to study the convergence speed of structure learning algorithms given their tuning parameters using those measures.

A Simpler Case: Undirected Graphs

Each edge e_{ij} in an undirected graph $\mathcal{G} = (\mathbf{V}, E)$ has only two possible states and therefore can be modelled as a Bernoulli random variable:

$$e_{ij} \sim E_{ij} = \begin{cases} 1 & \text{if } e_i \in E \\ 0 & \text{otherwise} \end{cases}.$$

The natural extension of this approach is to model any set of edges as a multivariate Bernoulli random variable $\mathbf{B} \sim Ber_k(\mathbf{p})$. $\mathbf{B}$ is uniquely identified by the parameter set

$$\mathbf{p} = \{p_I : I \subseteq \{1, \ldots, k\}, i \neq \varnothing\}, \qquad k = \frac{|\mathbf{V}|(|\mathbf{V}| - 1)}{2}$$

which represents the dependence structure among the marginal distributions $B_i \sim Ber(p_i)$, $i = 1, \ldots, k$ of the edges. $\mathbf{p}$ can be estimated using a large number of bootstrap samples or MCMC samples from $\mathrm{P}(\mathcal{G}(\mathcal{E}) \mid \mathcal{D})$.

DAGs as Multivariate Trinomials

Each arc a_{ij} in $\mathcal{G} = (\mathbf{V}, A)$ has three possible states, and therefore it can be modelled as a Trinomial random variable A_{ij}:

$$a_{ij} \sim A_{ij} = \begin{cases} -1 & \text{if } a_{ij} = \overleftarrow{a_{ij}} = \{v_i \leftarrow v_j\} \\ 0 & \text{if } a_{ij} \notin A, \text{ denoted with } \mathring{a_{ij}} \\ 1 & \text{if } a_{ij} = \overrightarrow{a_{ij}} = \{v_i \rightarrow v_j\} \end{cases}.$$

As before, the natural extension to model any set of arcs is to use a multivariate Trinomial random variable $\mathbf{T} \sim Tri_k(\mathbf{p})$. However:

- the acyclicity constraint of Bayesian networks makes deriving exact results very difficult because it cannot be written in closed form;
- the score equivalence of most structure learning strategies makes inference on $Tri_k(\mathbf{p})$ tricky.

Second Order Properties of $Ber_k(\mathbf{p})$ and $Tri_k(\mathbf{p})$

All the elements of the covariance matrix Σ of an edge set $\mathcal{E}$ are bounded,

$$p_i \in [0,1] \Rightarrow \sigma_{ii} = p_i - p_i^2 \in \left[0, \frac{1}{4}\right] \Rightarrow \sigma_{ij} \in \left[0, \frac{1}{4}\right],$$

and similar bounds exist for the eigenvalues $\lambda_1, \ldots, \lambda_k$,

$$0 \leqslant \lambda_i \leqslant \frac{k}{4} \qquad \text{and} \qquad 0 \leqslant \sum_{i=1}^{k} \lambda_i \leqslant \frac{k}{4}.$$

These bounds define a closed convex set in $\mathbb{R}^k$,

$$\mathcal{L} = \left\{ \Delta^{k-1}(c) : c \in \left[0, \frac{k}{4}\right] \right\}$$

where $\Delta^{k-1}(c)$ is the non-standard $k-1$ simplex

$$\Delta^{k-1}(c) = \left\{ (\lambda_1, \ldots, \lambda_k) \in \mathbb{R}^k : \sum_{i=1}^{k} \lambda_i = c, \lambda_i \geqslant 0 \right\}.$$

Similar results hold for arc sets, with $\sigma_{ii} \in [0,1]$ and $\lambda_i \in [0,k]$.

Minimum and Maximum Entropy

These results provide the foundation for characterising three cases corresponding to different configurations of the probability mass in $\mathrm{P}(\mathcal{G}(\mathcal{E}))$ and $\mathrm{P}(\mathcal{G}(\mathcal{E}) \mid \mathcal{D})$:

- minimum entropy: the probability mass is concentrated on a single DAG. This is the best possible configuration for $\mathrm{P}(\mathcal{G}(\mathcal{E}) \mid \mathcal{D})$, because only one arc set A has a non-zero posterior probability.

- intermediate entropy: several DAGs have non-zero probability. This is the case for informative priors $\mathrm{P}(\mathcal{G}(\mathcal{E}))$ and for the posteriors $\mathrm{P}(\mathcal{G}(\mathcal{E}) \mid \mathcal{D})$ resulting from real-world data sets.

- maximum entropy: all DAGs have the same probability. This is the worst possible configuration for $\mathrm{P}(\mathcal{G}(\mathcal{E}) \mid \mathcal{D})$: it corresponds to a non-informative prior. In other words, the data $\mathcal{D}$ do not provide any information useful in identifying a high-posterior $\mathcal{G}$.

Properties of the Multivariate Bernoulli

In the minimum entropy case, only one configuration of edges E has non-zero probability, which means that

$$p_{ij} = \begin{cases} 1 & \text{if } e_{ij} \in E \\ 0 & \text{otherwise} \end{cases} \qquad \text{and} \qquad \Sigma = \mathbf{O}$$

where $\mathbf{O}$ is the zero matrix.

The uniform distribution over $\mathbf{G}$ arising from the maximum entropy case has been studied extensively in random graph theory; its two most relevant properties are that all edges e_{ij} are independent and have $p_{ij} = \frac{1}{2}$. As a result, $\Sigma = \frac{1}{4} I_k$; all edges display their maximum possible variability, which along with the fact that they are independent makes this distribution non-informative for $\mathcal{E}$ as well as $\mathcal{G}(\mathcal{E})$.

Properties of the Multivariate Trinomial

The minimum entropy is the same; in the maximum entropy case:

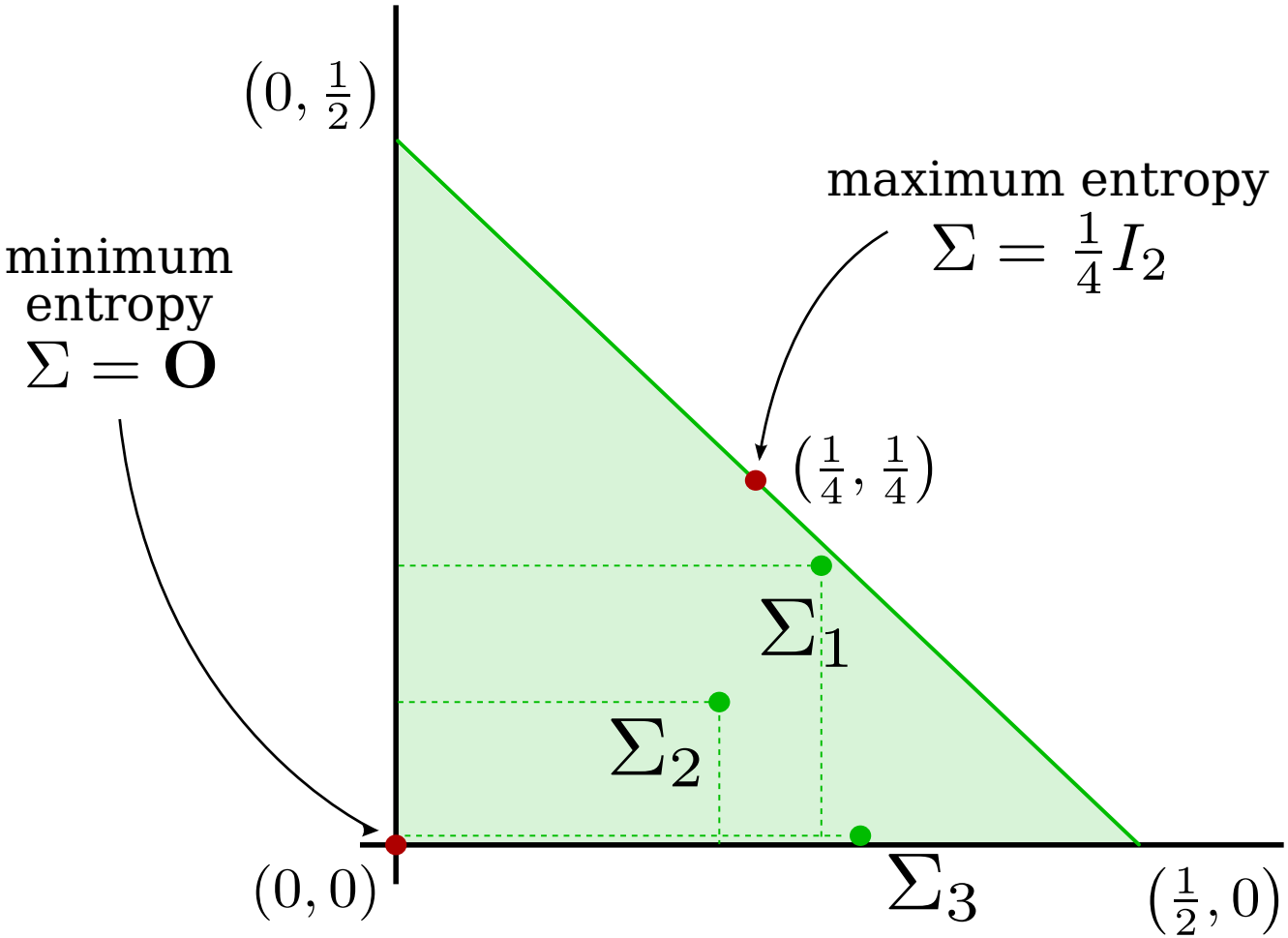
$$\mathrm{P}(\overrightarrow{a_{ij}}) = \mathrm{P}(\overleftarrow{a_{ij}}) \approx \frac{1}{4} + \frac{1}{4(N-1)} \to \frac{1}{4},$$
$$\mathrm{P}(\mathring{a}_{ij}) \approx \frac{1}{2} - \frac{1}{2(N-1)} \to \frac{1}{2} \text{ as } N \to \infty$$

and

$$\begin{aligned}
\mathrm{E}(A_{ij}) &= \mathrm{P}(\overrightarrow{a_{ij}}) - \mathrm{P}(\overleftarrow{a_{ij}}) = 0, \\
\mathrm{VAR}(A_{ij}) &= 2\,\mathrm{P}(\overrightarrow{a_{ij}}) \approx \frac{1}{2} + \frac{1}{2(N-1)} \to \frac{1}{2}, \\
|\mathrm{COV}(A_{ij}, A_{kl})| &= 2\,[\mathrm{P}(\overrightarrow{a_{ij}}, \overrightarrow{a_{kl}}) - \mathrm{P}(\overrightarrow{a_{ij}}, \overleftarrow{a_{kl}})] \\
&\lessapprox 4 \left[\frac{3}{4} - \frac{1}{4(N-1)}\right]^2 \left[\frac{1}{4} + \frac{1}{4(N-1)}\right]^2 \to \frac{9}{64}.
\end{aligned}$$

with $\mathrm{COV}(A_{ij}, A_{jl}) \to {}^{9}/_{64}$ and $\mathrm{COV}(A_{ij}, A_{kl}) = 0$.

A Geometric Representation of Entropy in $\mathcal{L}$



The space of the eigenvalues $\mathcal{L}$ for two edges in an undirected graph.

Univariate Measures of Variability

- The generalised variance, $\mathrm{VAR}_G(\Sigma) = \det(\Sigma) = \prod_{i=1}^{k} \lambda_i \in \left[0, \frac{1}{4^k}\right]$.
- The total variance (or total variability),

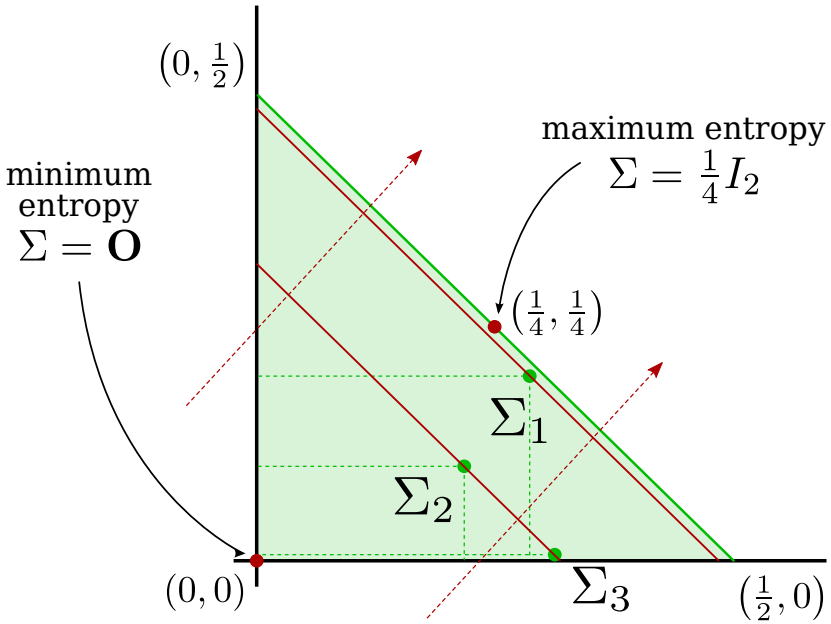
$$\mathrm{VAR}_T(\Sigma) = \mathrm{tr}\,(\Sigma) = \sum_{i=1}^{k} \lambda_i \in \left[0, \frac{k}{4}\right].$$

- The squared Frobenius matrix norm,

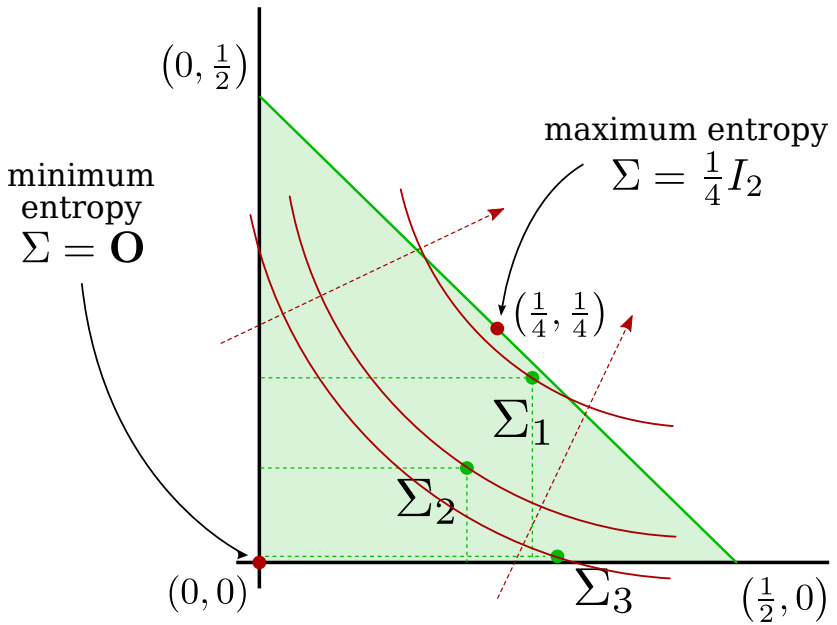
$$\mathrm{VAR}_F(\Sigma) = |||\Sigma - \frac{k}{4} I_k|||_F^2 = \sum_{i=1}^{k} \left(\lambda_i - \frac{k}{4}\right)^2 \in \left[\frac{k(k-1)^2}{16}, \frac{k^3}{16}\right].$$

All of these measures can be rescaled to vary in $[0, 1]$ and to associate high values to networks whose structure displays a high entropy. The equivalent measures of variability for DAGs work in the same way.

Structure Variability: Level Curves



Level curves in $\mathcal{L}$ for $\mathrm{VAR}_T(\Sigma)$.

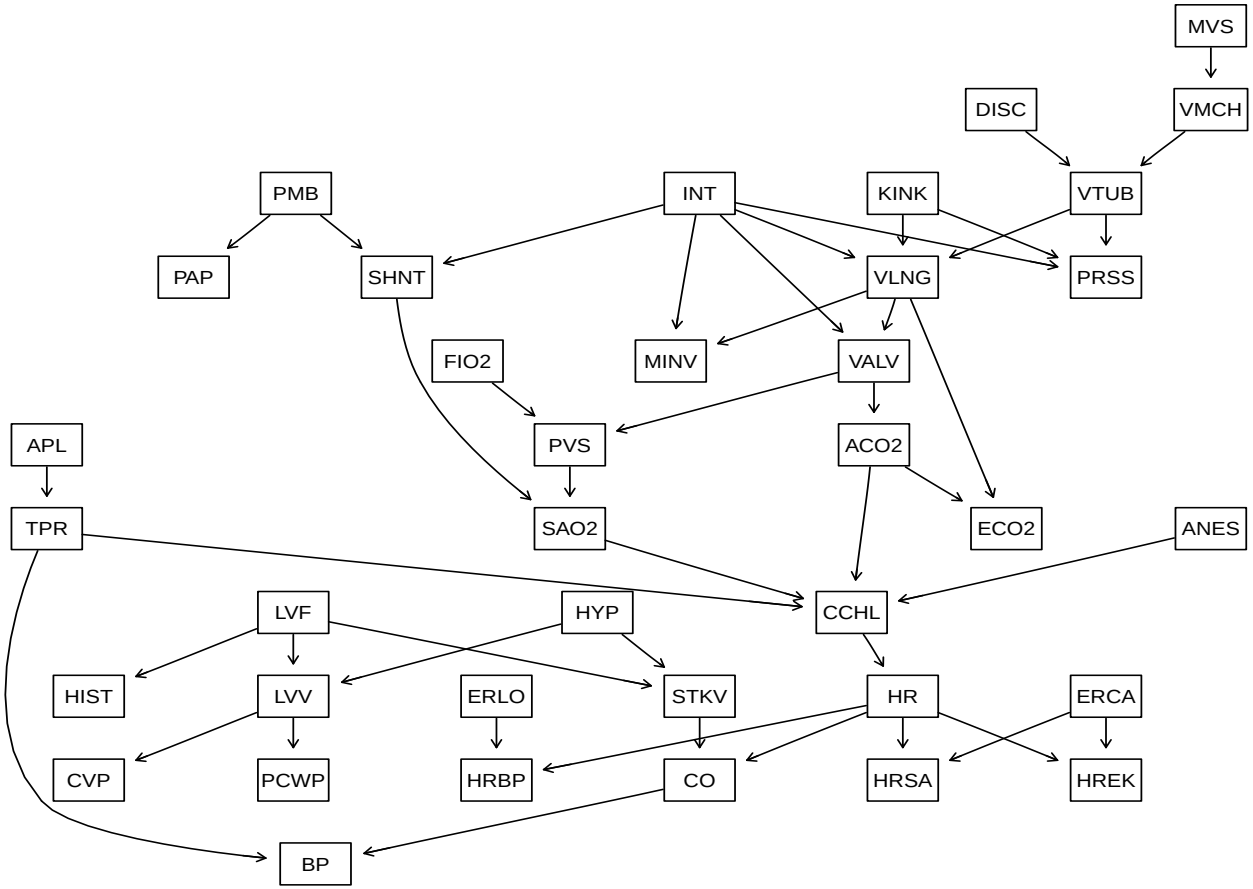


Level curves in $\mathcal{L}$ for $\mathrm{VAR}_F(\Sigma)$.

Pros & Cons About This Approach

- First and second order properties of $\mathrm{P}(\mathcal{G}(\mathcal{E}))$ and $\mathrm{P}(\mathcal{G}(\mathcal{E}) \mid \mathcal{D})$ can be often derived in closed form, and have a geometric interpretation.
- We now have descriptive measures of variability over the space of DAGs; we know that structure learning algorithms are consistent, so we can check how quickly the variability decreases as $n \to \infty$.
- Is there a way of identifying paths using covariance matrix decompositions?
- The covariance matrix $\mathrm{COV}(A_{ij}, A_{kl})$ is very big; so may want to regularise it by shrinking. This affects $\mathrm{P}(a_{ij})$ as well, and it is possible to use it for regularisation purposes. Applications to Bayesian model averaging and to identify significant arcs?

The ALARM Network



ALARM is a network designed to provide an alarm message system for intensive care unit patient monitoring. It has 37 nodes and 46 edges (of 666 possible edges), and its distribution has 509 parameters.

bnlearn: An Aside, Generating Observations from a BN

ALARM is one of several golden standard networks, which we can download from `bnlearn.com` to use in **bnlearn**. The fitted BN provides the true DAG of the network, which we can save as an R objects with `bn.net()`.

```
load("alarm.rda")
true.dag = bn.net(bn)
```

And we can use it to generate random samples from the BN for use in simulations and inference.

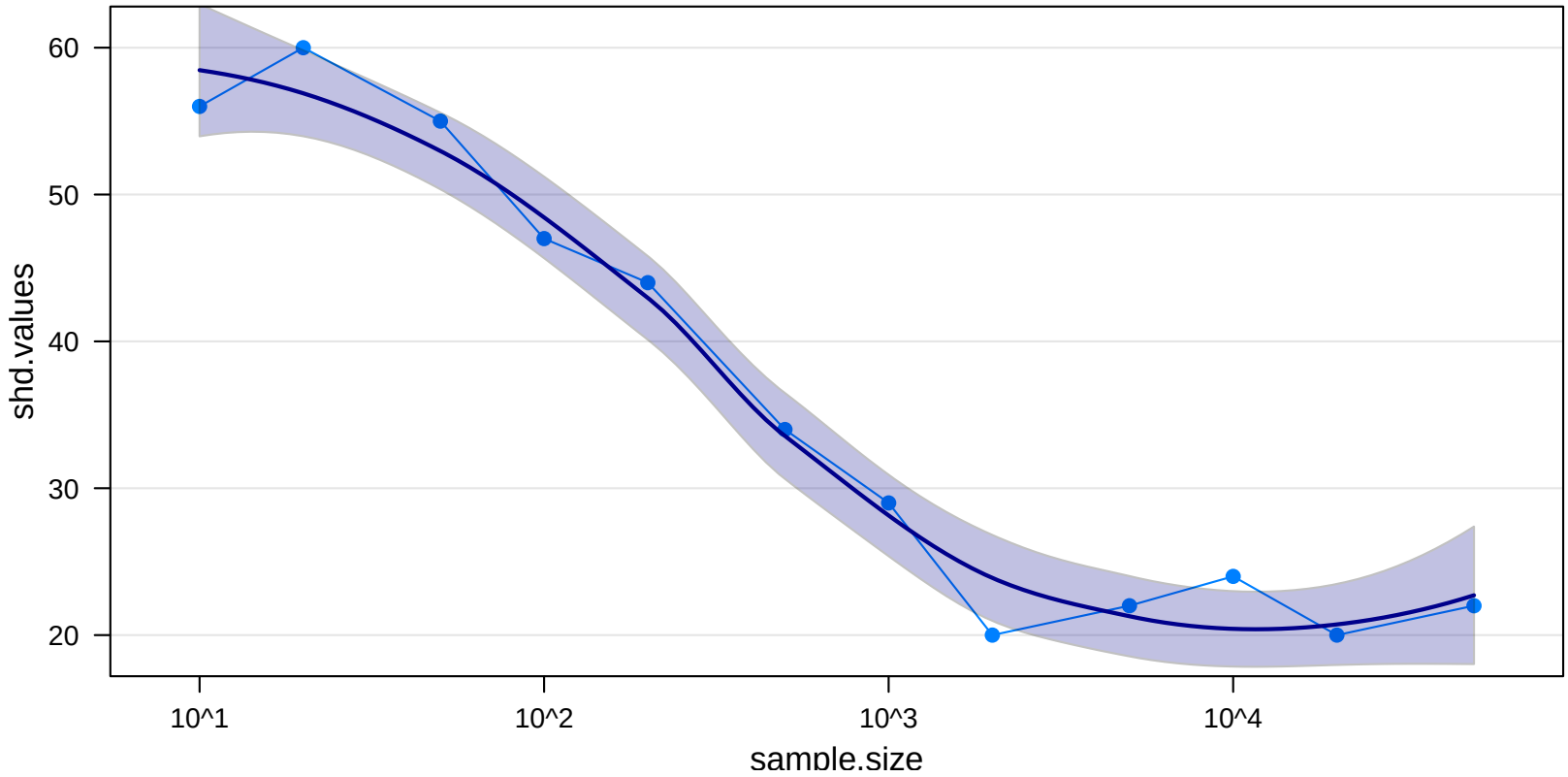
```
sim = rbn(bn, 100)
shd(hc(sim), true.dag)
```

```
## [1] 51
```

So, with these two functions we can now investigate whether structure learning algorithms are consistent.

So, Are Structure Learning Algorithms Consistent?

```
sample.size = outer(c(1, 2, 5), c(10, 10^2, 10^3, 10^4))
shd.values = numeric(length(sample.size))
for (i in seq_along(sample.size)) {
  sim = rbn(bn, sample.size[i])
  shd.values[i] = shd(hc(sim), true.dag)
}#FOR
```



bnlearn: Graph Priors in Structure Learning

The posterior scores BDe and BGe accept `prior` as an additional, optional argument specifying the prior $\mathrm{P}(\mathcal{G}(\mathcal{E}))$. The default is the uniform prior. So

```
unif = hc(alarm, score = "bde", iss = 1)
```

is equivalent to

```
unif = hc(alarm, score = "bde", iss = 1, prior = "uniform")
```

and the uniform graph prior has no tuning arguments.

```
shd(unif, dag)
## [1] 38
```

That is the reason why it was originally chosen as a "default" prior: it does not require prior information on the data and it is computationally very simple.

The Uniform Graph Prior, Revisited

Assuming a uniform prior is problematic because:

- Score-based structure learning algorithms typically generate new candidate DAGs by a single arc addition, deletion or reversal, e.g.

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

  U always simplifies, and that implies $\overrightarrow{p_{ij}} = \overleftarrow{p_{ij}} = \mathring{p}_{ij} = 1/3$ favouring the inclusion of new arcs as $\overrightarrow{p_{ij}} + \overleftarrow{p_{ij}} = 2/3$ for each possible arc a_{ij}.

- Two arcs are correlated if they are incident on a common node ($\mathrm{COV}(A_{ij}, A_{jl}) \to 9/64$) , 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.

The Marginal Uniform (MU) Graph Prior

We showed that

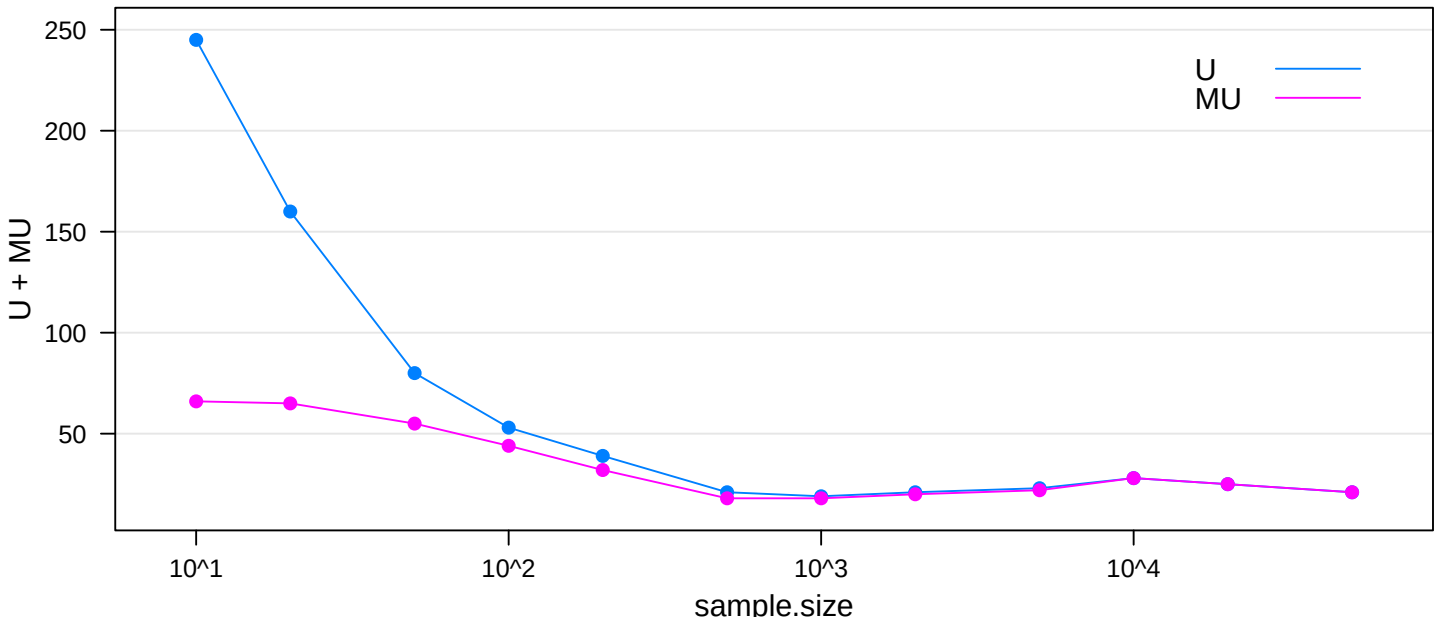
$$\overrightarrow{p_{ij}} = \overleftarrow{p_{ij}} \approx \frac{1}{4} + \frac{1}{4(N-1)} \to \frac{1}{4} \quad \text{and} \quad \mathring{p}_{ij} \approx \frac{1}{2} - \frac{1}{2(N-1)} \to \frac{1}{2},$$

so each possible arc is present in $\mathcal{G}$ with marginal probability $\approx 1/2$ and, when present, it appears in each direction with probability $1/2$. We can use that as a starting point, and assume an independent prior for each arc with the same marginal probabilities (hence the name MU).

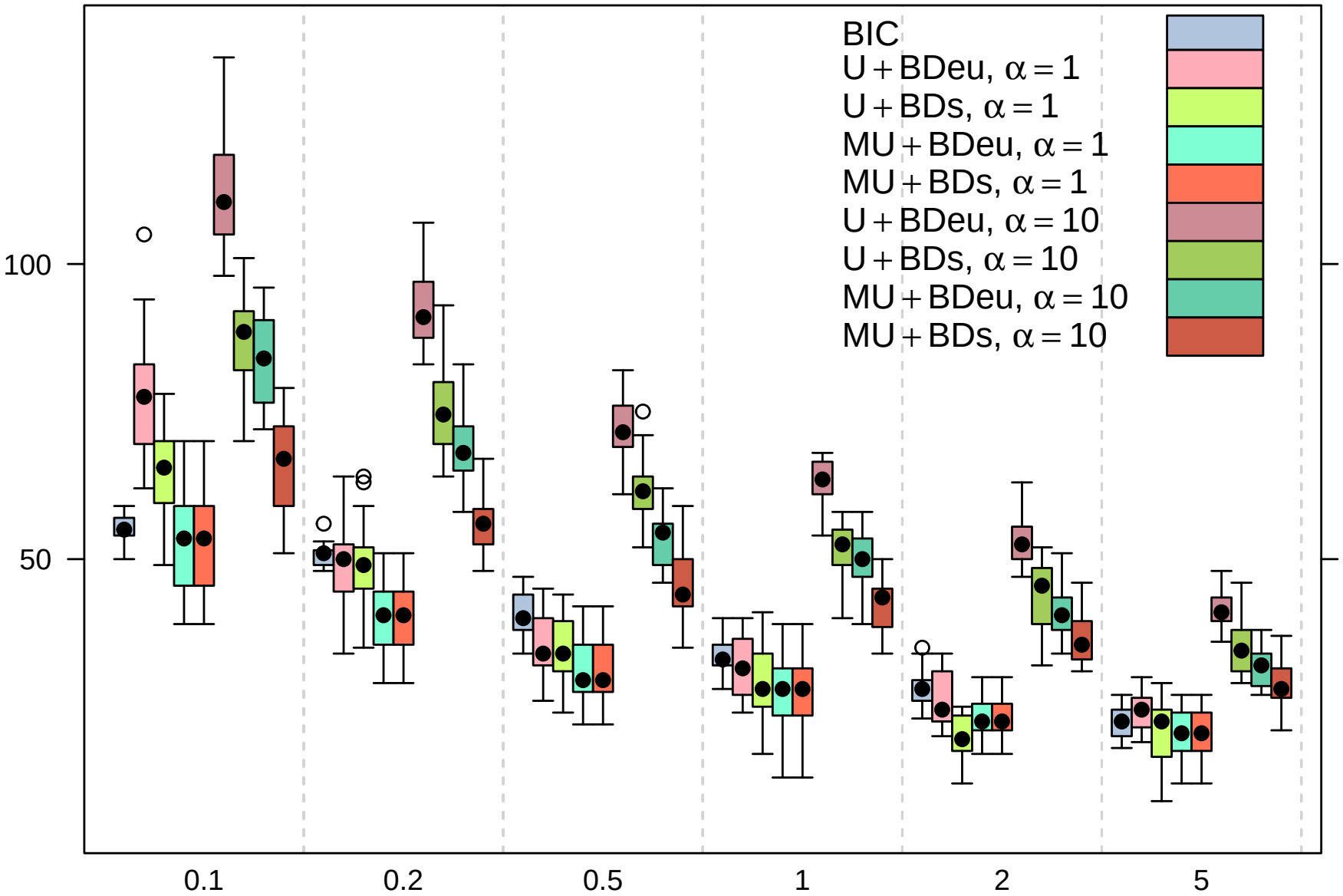
- MU does not favour arc inclusion as $\overrightarrow{p_{ij}} + \overleftarrow{p_{ij}} = 1/2$.
- MU does not favour the propagation of errors in structure learning because arcs are independent from each other.
- MU computationally trivial to use: the ratio of the prior probabilities is $1/2$ for arc addition, 2 for arc deletion and 1 for arc reversal, for all arcs.

bnlearn: A Comparison of Uniform Priors

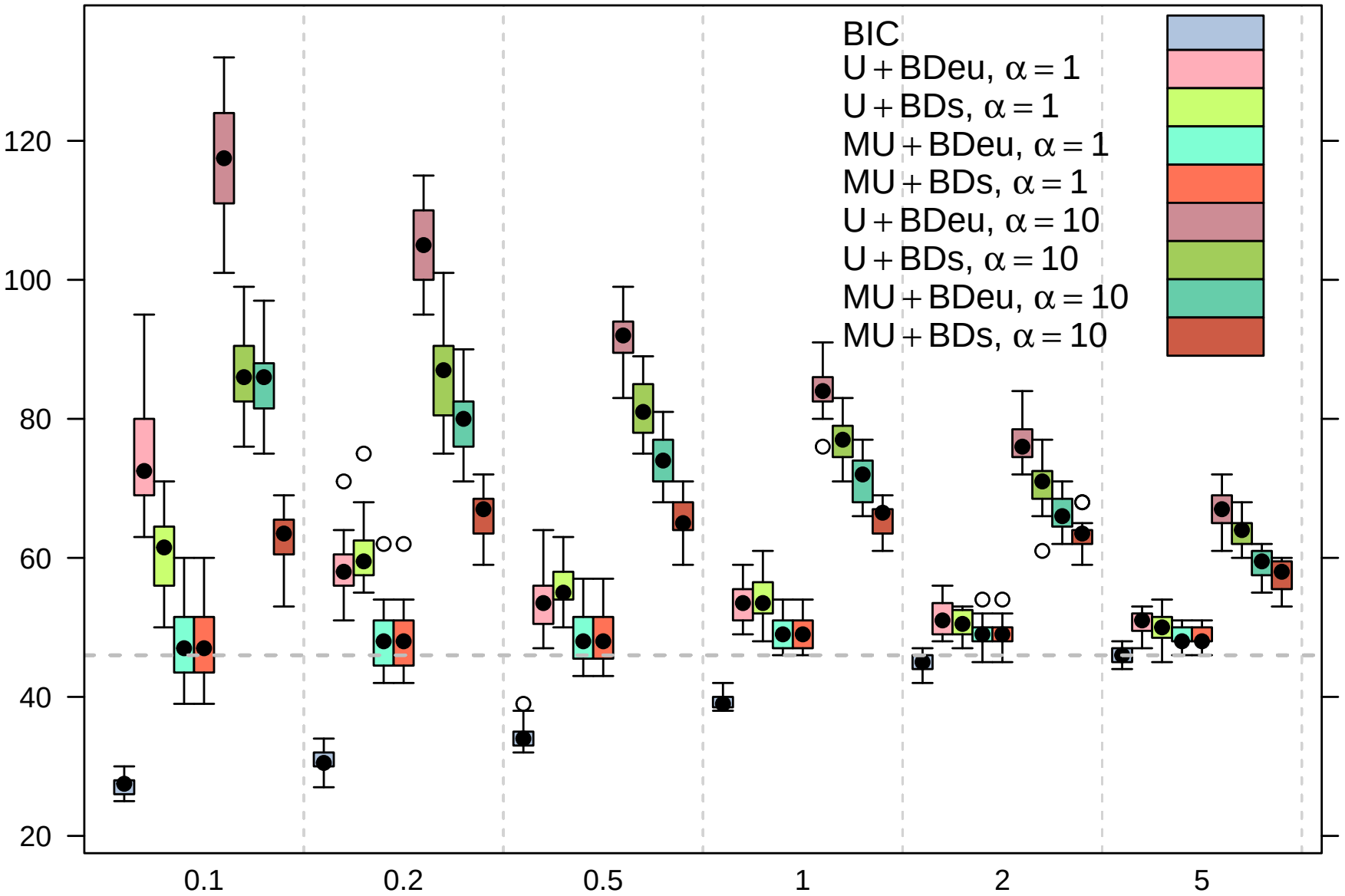
```
shd =
  data.frame(sample.size = outer(c(1, 2, 5), c(10, 10^2, 10^3, 10^4)),
    U = numeric(length(sample.size)), MU = numeric(length(sample.size)))
for (i in seq_along(sample.size)) {
  sim = rbn(bn, sample.size[i])
  dagU = hc(sim, score = "bde", iss = 1, prior = "uniform")
  dagMU = hc(sim, score = "bde", iss = 1, prior = "marginal")
  shd[i, c("U", "MU")] = c(shd(dagU, true.dag), shd(dagMU, true.dag))
}#FOR
```



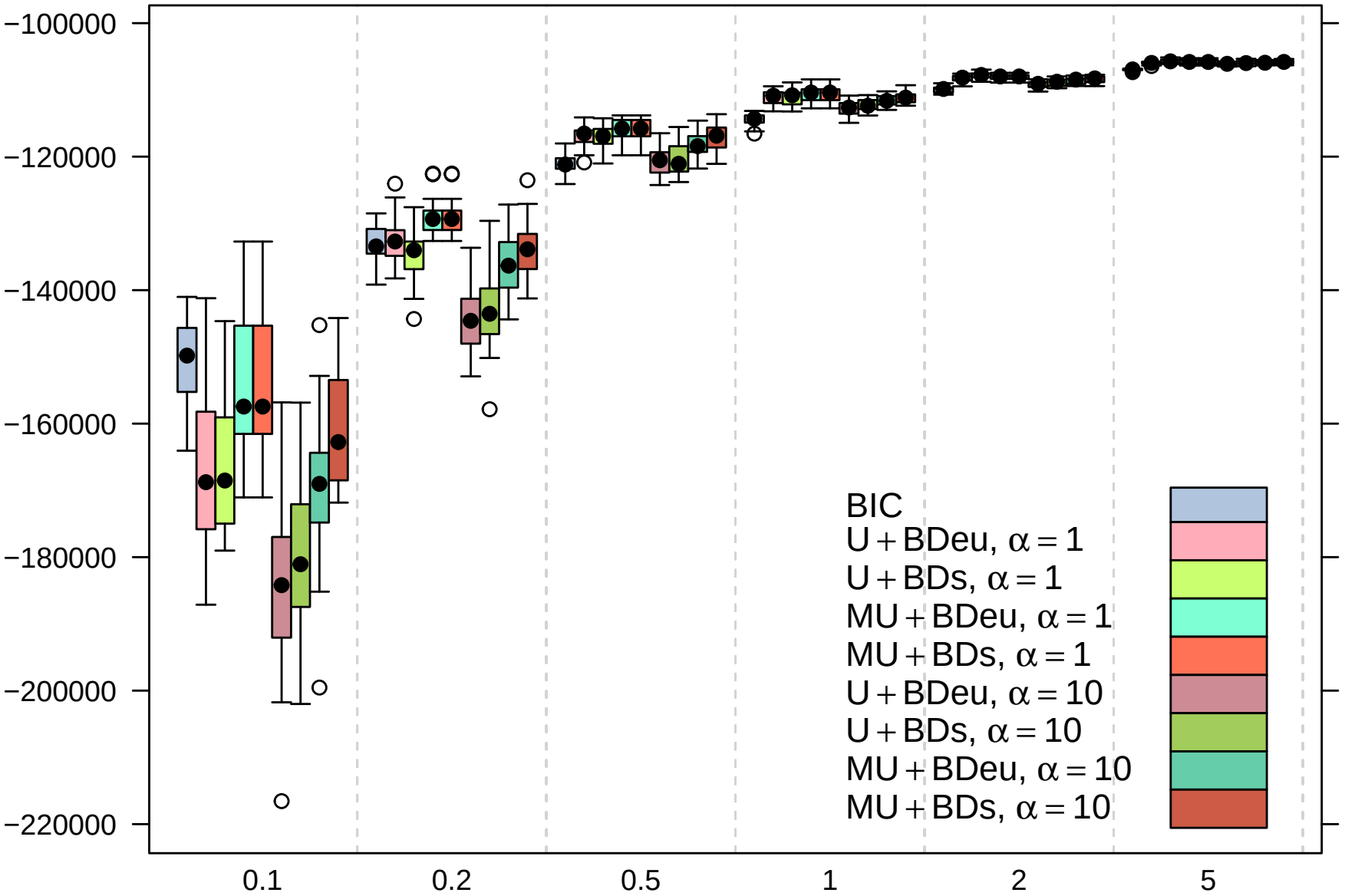
bnlearn: More Simulations (SHD)



bnlearn: More Simulations (Arcs)



bnlearn: More Simulations (Prediction)



The Castelo & Siebes Marginal Prior

In the marginal uniform prior the probabilities are fixed ; in the general case the Castelo & Siebes marginal prior makes it possible to specify different $\overrightarrow{p_{ij}}$, $\overleftarrow{p_{ij}}$, $\mathring{p}_{ij}$ for each arc. We can do this in a number of functions in **bnlearn** by setting `prior = "cs"` and `beta` as follows:

```
beta = data.frame(from = c("LVF", "CCHL"), to = c("LVV", "MVS"),
         prob = c(0.9, 0.1), stringsAsFactors = FALSE)
beta
```

```
##   from  to prob
## 1  LVF LVV  0.9
## 2 CCHL MVS  0.1
```

```
dag.cs = hc(alarm, score = "bde", iss = 1, prior = "cs", beta = beta)
dag.cs$learning$args$beta
```

```
##   from   to aid  fwd bkwd
## 1  MVS CCHL 445 0.45 0.10
## 2  LVF  LVV 482 0.90 0.05
```

Setting values for any number of arcs requires a substantial amount of prior knowledge, and it is easy to get them wrong!

The Variable Selection Prior

We can also borrow the classic variable selection prior from linear regression models, that is,

$$\mathrm{P}(k \text{ parents, } N-k \text{ non-parents}) = \frac{\beta^k}{(1-\beta)^{N-k}}, \qquad \beta \in (0,1);$$

whether or not a new parent is added to a node is controlled by the corresponding odds

$$\frac{\mathrm{P}(k+1 \text{ parents, } N-k-1 \text{ non-parents})}{\mathrm{P}(k \text{ parents, } N-k \text{ non-parents})} = \frac{\beta}{1-\beta}.$$

We can use it by setting `prior = vsp"` and `beta` to β.

```
hc(alarm, score = "bde", iss = 1, prior = "vsp", beta = 0.1)
```

Limiting the Number of Parents

A more drastic measure along the same lines is to put a hard limit on the number of parents of each node, which is implies the prior:

$$\mathrm{P}(\text{adding } (k+1)\text{th parent}) = \begin{cases} 1/2 & \text{if } k+1 \leqslant maxp \\ 0 & \text{otherwise} \end{cases}$$

that sets $\mathrm{P}(\mathcal{G}) = 0$ for any $\mathcal{G}$ that has at least one node with more than $maxp$ parents, while all other graphs have the same $\mathrm{P}(\mathcal{G})$.

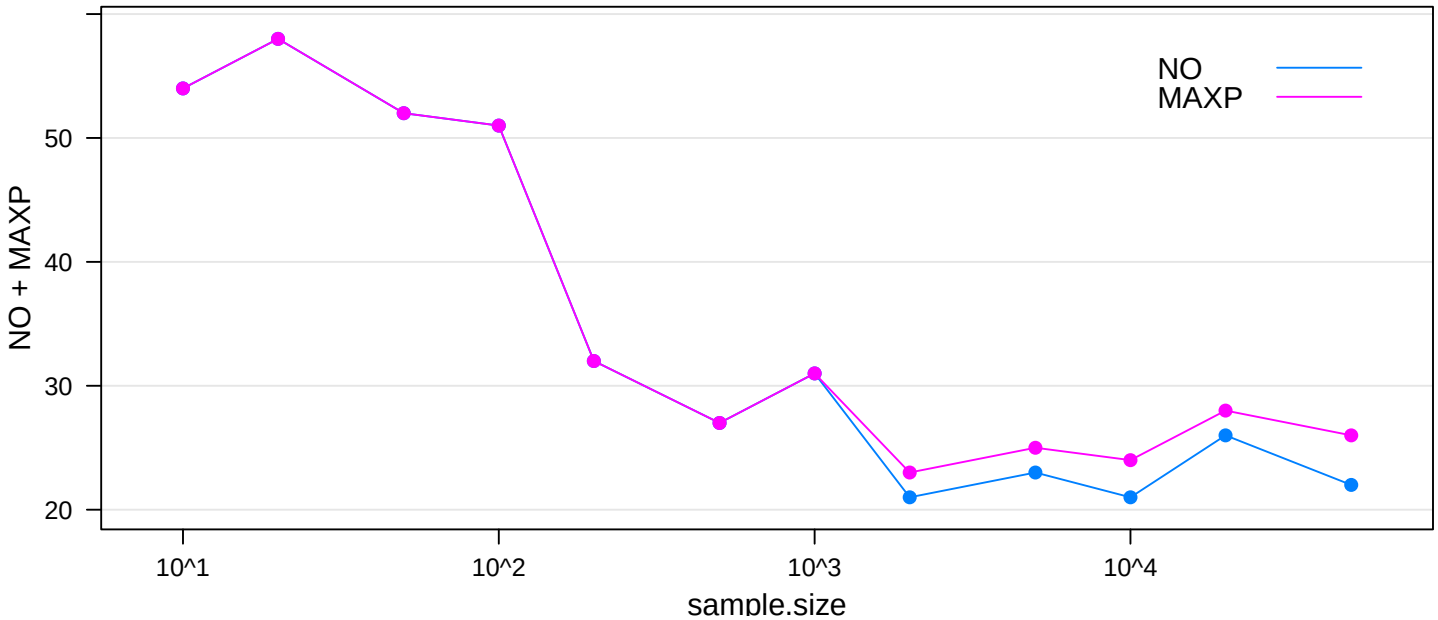
By convention we call sparse a DAG that has $\mathcal{O}(\mathbf{V}) = \mathcal{O}(A)$, so we usually want to set $maxp \in [1, 4]$ (1 forces DAGs to be trees):

```
hc(alarm, score = "bde", iss = 1, maxp = 3)
hc(alarm, score = "bic", maxp = 3)
```

Customarily, this has been used in the literature with all kinds of scores, so the `maxp` argument is available for use with any score in **bnlearn**.

bnlearn: It Can Make Things Worse If You Set It Too Low

```
shd =
  data.frame(sample.size = outer(c(1, 2, 5), c(10, 10^2, 10^3, 10^4)),
    NO = numeric(length(sample.size)), MAXP = numeric(length(sample.size)))
for (i in seq_along(sample.size)) {
  sim = rbn(bn, sample.size[i])
  dagNO = hc(sim, score = "bic")
  dagMAXP = hc(sim, score = "bic", maxp = 2)
  shd[i, c("NO", "MAXP")] = c(shd(dagNO, true.dag), shd(dagMAXP, true.dag))
}#FOR
```



Whitelisting and Blacklisting

A more granular application of this kind of hard prior constraints leads to the use of whitelists and blacklists:

- Arcs blacklisted in one direction only (i.e. $A \to B$ is blacklisted but $B \to A$ is not) are never present in that particular direction, but may be present in the other direction.
- Arcs blacklisted in both directions (i.e. both $A \to B$ and $B \to A$ are blacklisted) are never present in the graph, even as an undirected arc in a CPDAG.
- Arcs whitelisted in one direction only (i.e. $A \to B$ is whitelisted but $B \to A$ is not) have the respective reverse arcs blacklisted, and are always present in the graph.
- Arcs whitelisted in both directions (i.e. both $A \to B$ and $B \to A$ are whitelisted) are present in the graph, but their direction is set by the learning algorithm.

Any arc whitelisted and blacklisted at the same time is assumed to be whitelisted, and is thus removed from the blacklist.

bnlearn: Whitelists and Blacklists (I)

All structure learning algorithms in **bnlearn** have a `whitelist` and a `blacklist` arguments, that are interpreted as appropriate in terms of directed and undirected arcs at various stages of the algorithms.

In score-based algorithms, individual arcs are whitelisted and blacklisted.

```
head(arcs(hc(alarm)), n = 4)
```

```
##      from   to
## [1,] "PCWP" "LVV"
## [2,] "HRBP" "HR"
## [3,] "MINV" "VALV"
## [4,] "HR"   "HREK"
```

```
bl = data.frame(from = c("HRBP", "MINV"), to = c("HR", "VALV"))
head(arcs(hc(alarm, blacklist = bl)), n = 4)
```

```
##      from   to
## [1,] "PCWP" "LVV"
## [2,] "HREK" "HRSA"
## [3,] "HR"   "HRBP"
## [4,] "HREK" "HR"
```

bnlearn: Whitelists and Blacklists (II)

In constraint-based algorithms, arcs must be blacklisted in both directions to prevent them from being included in Markov blankets and neighbour sets; whitelists work normally.

```
head(arcs(si.hiton.pc(alarm)), n = 3)
```

```
##      from   to
## [1,] "CVP"  "LVV"
## [2,] "PCWP" "LVV"
## [3,] "HIST" "LVF"
```

```
bl = data.frame(from = c("PCWP"), to = c("LVV"))
head(arcs(si.hiton.pc(alarm, blacklist = bl)), n = 3)
```

```
##      from   to
## [1,] "CVP"  "LVV"
## [2,] "PCWP" "LVV"
## [3,] "HIST" "LVF"
```

```
bl = data.frame(from = c("PCWP", "LVV"), to = c("LVV", "PCWP"))
head(arcs(si.hiton.pc(alarm, blacklist = bl)), n = 3)
```

```
##      from   to
## [1,] "CVP"  "LVV"
## [2,] "PCWP" "LVF"
## [3,] "HIST" "LVF"
```

Parameter Learning: Likelihood, Bayesian and Shrinkage

Once the structure of the model is known, the problem of estimating the parameters of the global distribution can be solved by estimating the parameters of the local distributions, one at a time.

Common choices are:

- Maximum likelihood estimators: just the usual empirical estimators. Often described as either maximum entropy or minimum divergence estimators in information-theoretic literature.
- Bayesian posterior estimators: posterior estimators, based on conjugate priors to keep computations fast, simple and in closed form.
- Shrinkage estimators: regularised estimators based either on James-Stein or Bayesian shrinkage results.

Maximum Likelihood and Maximum Entropy Estimators

The classic estimators for (conditional) probabilities and (partial) correlations / regression coefficients are a bad choice for almost all real-world problems. They are still around because:

- they are used in benchmark simulations;
- computer scientists do not care much about parameter estimation.

However:

- maximum likelihood estimates are unstable in most multivariate problems, both discrete and continuous;
- for the multivariate Gaussian distribution, James & Stein proved in the 1950s that the maximum likelihood estimator for the mean is not admissible in 3+ dimensions;
- partial correlations are often ill-behaved because of that, even with Moore-Penrose pseudo-inverses;
- maximum likelihood estimates are non-smooth and create problems when using the graphical model for inference.

Maximum a Posteriori Bayesian Estimators

Bayesian posterior estimates are the sensible choice for parameter estimation according to Koller's & Friedman's tome on graphical models. Choices for the priors are limited (for computational reasons) to conjugate distributions, namely:

- the Dirichlet for discrete models, i.e.

$$Dir(\alpha_{k|\Pi_{X_i}=\pi}) \quad \xrightarrow{data} \quad Dir(\alpha_{k|\Pi_{X_i}=\pi} + n_{k|\Pi_{X_i}=\pi})$$

  meaning that $\hat{p}_{k|\Pi_{X_i}=\pi} = \alpha_{k|\Pi_{X_i}=\pi} / \sum_{\pi} \alpha_{k|\Pi_{X_i}=\pi}$.

- the Inverse Wishart for Gaussian models, i.e.

$$IW(\Psi, m) \quad \xrightarrow{data} \quad IW(\Psi + n\Sigma, m + n).$$

In both cases (when a non-informative prior is used) the only free parameter is the equivalent or imaginary sample size, which gives the relative weight of the prior compared to the observed sample.

Bayesian LASSO and Ridge Regression

Gaussian graphical models, being closely related with linear regression, have also used ridge regression (L_2 regularisation) and LASSO (L_1 regularisation) in their Bayesian capacity.

LASSO corresponds to a Laplace prior on the regression coefficients,

$$\beta_k \mid \sigma^2 \sim Laplace(0, \sigma^2).$$

Ridge Regression corresponds to a Gaussian prior,

$$\beta_k \mid \sigma^2 \sim N(0, \sigma^2).$$

In both cases tuning the σ^2 parameter is crucial, as it takes the role of the λ regularisation parameter found in the original frequentist definitions of these methods. Also, excessive regularisation might lead to zero coefficients that would make a node independent of its parents.

Shrinkage, James-Stein Estimation

Shrinkage estimation is based on results from James & Stein on the estimation of the mean of a multivariate Gaussian distribution, and takes the form

$$\tilde{\theta} = \lambda t + (1 - \lambda)\hat{\theta} \qquad \lambda \in [0, 1]$$

where the optimal λ (with respect to squared loss) can be estimated in closed form as

$$\lambda^* = \min \left(\frac{\sum_k \mathrm{VAR}(\hat{\theta}_k) - \mathrm{COV}(\hat{\theta}_k, t_k) + \mathrm{Bias}(\hat{\theta}_k)\,\mathrm{E}(\hat{\theta}_k - t_k)}{\sum_k (\hat{\theta}_k - t_k)^2}, 1 \right)$$

The James-Stein estimator $\tilde{\theta}$ dominates the maximum likelihood estimator $\hat{\theta}$ and converges to the latter as the sample size grows. It can be interpreted as an empirical Bayes estimator.

Shrinkage, James-Stein Estimation

For discrete data, conditional probabilities $p_{k|\pi} = p_{k|\Pi_{X_i}=\pi}$ end up being estimated as

$$\tilde{p}_{k|\pi} = \lambda^* t_{k|\pi} + (1-\lambda^*)\hat{p}_{k|\pi}, \quad \lambda^* = \min\left(\frac{1-\sum_k \hat{p}^2_{k|\pi}}{(n-1)\sum_k (t_{k|\pi}-\hat{p}_{k|\pi})^2}, 1\right),$$

where t is the uniform (discrete) distribution.

For continuous data, correlations end up being estimated from the shrunk covariance matrix $\tilde{\Sigma}$

$$\tilde{\sigma}_{ii} = \hat{\sigma}_{ii}, \quad \tilde{\sigma}_{ij} = (1-\lambda^*)\hat{\sigma}_{ij}, \quad \lambda^* = \min\left(\frac{\sum_{i\neq j}\mathrm{VAR}(\hat{\sigma}_{ij})}{\sum_{i\neq j}\hat{\sigma}^2_{ij}}, 1\right)$$

where t is $\mathrm{diag}(\hat{\Sigma})$. $\tilde{\Sigma}$ is guaranteed to have full rank, so it can be safely inverted to get partial correlations.

bnlearn: Parameter Learning, DBNs

Parameter learning is implemented in `bn.fit()` and defaults to `method = "mle"`; for discrete data we can also use Bayesian posterior estimation with `method = "bayes"` with an imaginary sample size `iss`.

```
fitted = bn.fit(hc(asia), asia, method = "mle")
coef(fitted$X)
```

```
##      E
## X          no     yes
##   no  0.95659 0.00541
##   yes 0.04341 0.99459
```

```
fitted = bn.fit(hc(asia), asia, method = "bayes", iss = 20)
coef(fitted$X)
```

```
##      E
## X          no    yes
##   no  0.9556 0.0184
##   yes 0.0444 0.9816
```

bnlearn: Parameter Learning, GBNs

bnlearn implements only `method = "mle"` directly for GBNs, but we can use `penalized()` to replace parameter estimates with ridge, LASSO, or elastic net estimates.

```
library(penalized)
fitted = bn.fit(hc(marks), marks)
coef(fitted$ALG)
```

```
## (Intercept)        MECH        VECT
##      25.362       0.183       0.358
```

```
fitted$ALG = penalized(response = marks[, "ALG"],
               penalized = marks[, parents(fitted, "ALG")],
               lambda2 = 100, model = "linear", trace = FALSE)
coef(fitted$ALG)
```

```
## (Intercept)        MECH        VECT
##      25.481       0.184       0.355
```

We can also fit the parameters directly using `penalized()` and a DAG, and collect them in a BN with `custom.fit()`.

Model Averaging: Frequentist, Bayesian and Hybrid

The results of both structure learning and parameter learning should be validated before using a BN for inference. Since parameters are learned conditional on the results of structure learning, validating the (CP)DAG learned from the data would be the first step.

- frequentist: generating network structures using bootstrap and model averaging (aka bagging).
- Bayesian: generating network structures from the posterior $\mathrm{P}(\mathcal{G} \mid \mathcal{D})$ using exhaustive enumeration or Markov Chain Mote Carlo approximations.
- hybrid: generating network structures again using bootstrap, but weighting them with their posterior probabilities when performing model averaging.

A Frequentist Approach: Friedman's Confidence

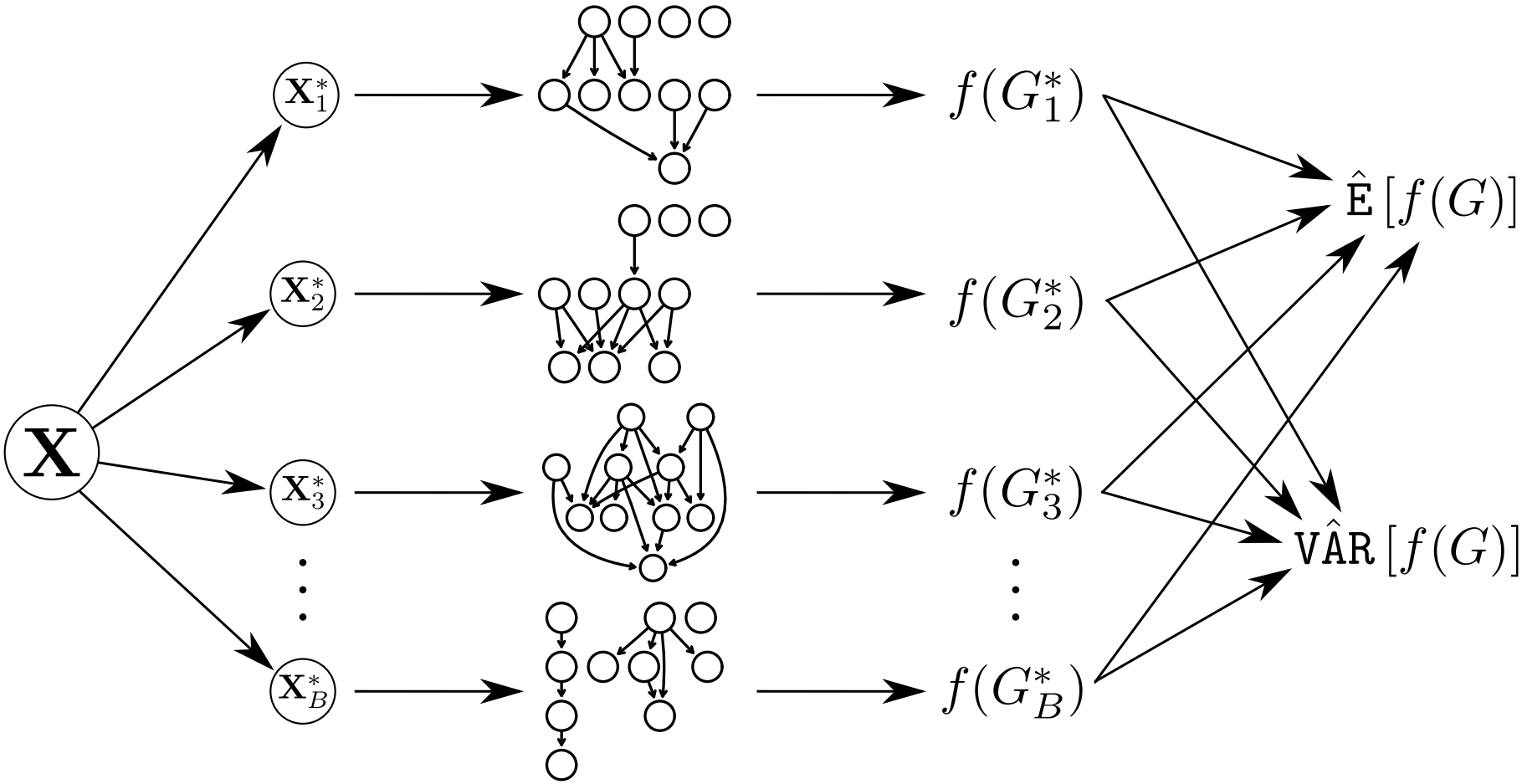
Friedman et al. proposed an approach to model validation based on bootstrap resampling and model averaging:

1. For $b = 1, 2, \ldots, B$:
   1.1 sample a new data set $\mathcal{D}_b^*$ from the original data $\mathcal{D}$ using either parametric or nonparametric bootstrap;
   1.2 learn the structure of the BN $\mathcal{G}_b = (\mathbf{V}, A_b)$ from $\mathcal{D}_b^*$.
2. Estimate the strength or confidence that each possible arc a_i is present in the true DAG $\mathcal{G}_0 = (\mathbf{V}, A_0)$ as

$$\hat{p}_i = \hat{\mathrm{P}}(a_i) = \frac{1}{B} \sum_{b=1}^{B} \mathbb{1}_{\{e_i \in A_b\}},$$

where $\mathbb{1}_{\{e_i \in A_b\}}$ is equal to 1 if $e_i \in E_b$ and 0 otherwise.

A Frequentist Approach: Friedman's Confidence



bnlearn: Arc Strength

This approach is implemented in `boot.strength()`, which takes a data set $\mathcal{D}$, a structure learning `algorithm` and its `algorithm.args`, and performs bootstrap resampling `R` times.

```
str = boot.strength(alarm, algorithm = "hc",
        algorithm.args = list(score = "bde", iss = 1), R = 100)
head(str[str$strength > 0.50, ])
```

```
##     from   to strength direction
## 24   CVP  LVV        1     0.160
## 53  PCWP  LVF        1     0.165
## 60  PCWP  LVV        1     0.510
## 89  HIST  LVF        1     0.755
## 112  TPR   BP        1     1.000
## 118  TPR SAO2        1     0.000
```

The return value has two strength measures, `strength` and `direction`, representing

$$\mathrm{P}(\overrightarrow{p_{ij}} + \overleftarrow{p_{ij}}) \qquad \text{and} \qquad \mathrm{P}(\overrightarrow{p_{ij}} \mid \overrightarrow{p_{ij}} + \overleftarrow{p_{ij}}).$$

A (Full) Bayesian Approach

Performing a full posterior Bayesian analysis on DAGs, that is, working with

$$\hat{p}_i = \mathrm{E}(e_i|\mathcal{D}) = \sum_{\mathcal{G}} \mathbb{1}_{\{e_i \in E_{\mathcal{G}}\}} \mathrm{P}(\mathcal{G} \mid \mathcal{D}),$$

is considered unfeasible for DAGs with more than ≈ 10 nodes because:

- an exhaustive enumeration takes too long, and it's even worse for BNs because of the acyclicity constraint;
- generating DAGs from the posterior distribution is feasible but convergence of the MCMC to the stationary distribution is far from certain (mixing is often too slow).

A Hybrid Approach: the "Bayesian confidence"

Friedman's confidence and Bayesian posterior analysis may be combined as follows:

1. For $b = 1, 2, \ldots, B$:
   - 1.1 sample a new data set $\mathcal{D}_b^*$ from the original data $\mathcal{D}$ using either parametric or nonparametric bootstrap;
   - 1.2 learn the structure of the graphical model $\mathcal{G}_b = (\mathbf{V}, E_b)$ from $\mathcal{D}_b^*$.
2. Estimate the strength confidence for each possible edge e_i as

$$\hat{p}_i = \mathrm{E}(e_i|\mathcal{D}) \approx \frac{1}{B} \sum_{b=1}^{B} \mathbb{1}_{\{e_i \in E_b\}} \mathrm{P}(\mathcal{G}_b \mid \mathcal{D}).$$

The result is a form of approximate Bayesian estimation, whose behaviour depends on how much of the posterior probability mass is concentrated in the subset of DAGs $\mathcal{G}_b$.

bnlearn: Arc Strength and Weights (I)

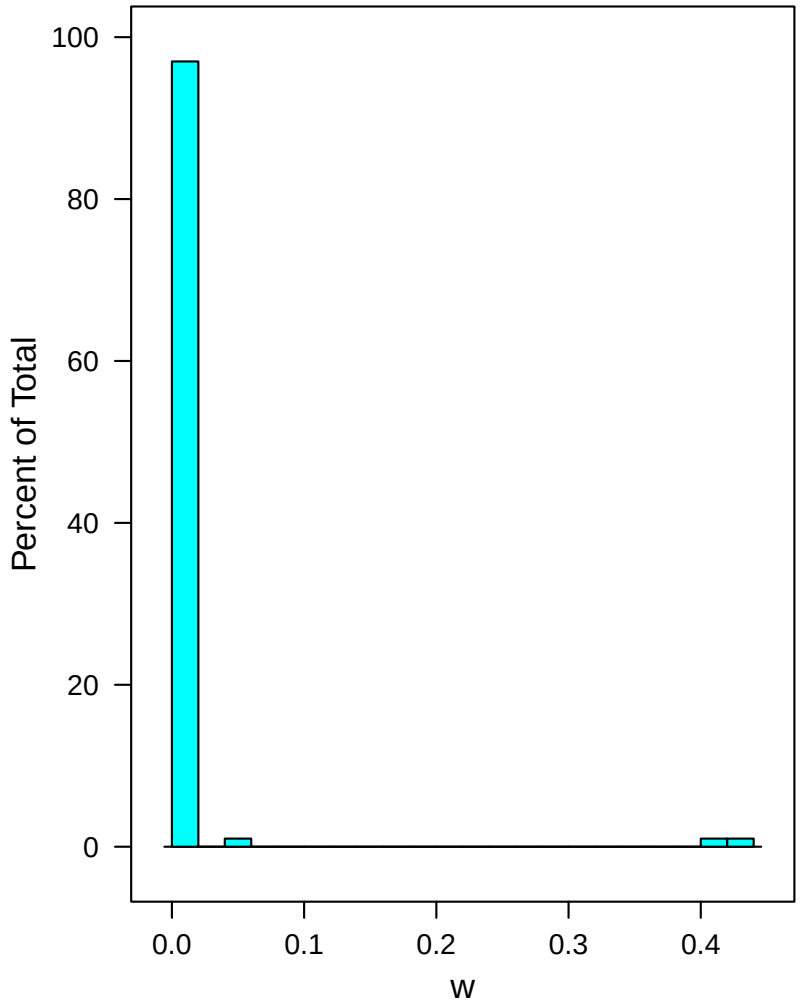
This approach requires two separate steps:

1. we can estimate the $\mathcal{G}_b$ with `bn.boot()`, without computing any statistic on them (the `I()` function does literally nothing);
2. and then we can iterate with `sapply()` over the DAGs to compute the $\mathrm{P}(\mathcal{G}_b \mid \mathcal{D})$.

```
Gb = bn.boot(alarm, algorithm = "hc", statistic = I,
        algorithm.args = list(score = "bde", iss = 1), R = 100)
w = sapply(Gb, score, data = alarm, type = "bde", iss = 1)
library(Rmpfr)
w = mpfr(w, precBits = 160)
w = asNumeric(exp(w) / sum(exp(w)))
wstr = custom.strength(Gb, weights = w, nodes = names(alarm))
```

Note that `score()` returns $\log \mathrm{BDe}(\mathcal{G}_b)$ but we need $\exp(\log \mathrm{BDe}(\mathcal{G}_b))$; the $\log \mathrm{BDe}(\mathcal{G}_b)$ are so small that it impossible to exponentiate them without using an arbitrary precision library.

bnlearn: Arc Strength and Weights (II)



Unfortunately, for any middle-sized and large BN (say, 10 or more nodes) the $\mathrm{P}(\mathcal{G}_b \mid \mathcal{D})$ will be so small that once normalised only 1-3 weights will be significantly different from zero.

The reason is that the space of the possible DAGs is extremely large and $\mathrm{P}(\mathcal{G}(\mathcal{E}) \mid \mathcal{D})$ will be extremely flat, so $\mathrm{P}(\mathcal{G}_b \mid \mathcal{D}) \to 0$, with a few networks having values e.g. 10^{-200} compared to e.g. 10^{-205} for the rest.

Identifying Significant Arcs

- The confidence values $\hat{\mathbf{p}} = \{\hat{p}_i\}$ do not sum to one and are dependent on one another in a nontrivial way; the value of the confidence threshold (i.e. the minimum confidence for an arc to be accepted as an arc of $\mathcal{G}_0$ regardless of direction) is an unknown function of both the data and the structure learning algorithm.
- The ideal/asymptotic configuration $\tilde{\mathbf{p}}$ of confidence values would be

$$\tilde{p}_i = \begin{cases} 1 & \text{if } e_i \in E_0 \\ 0 & \text{otherwise} \end{cases},$$

  i.e. all the networks $\mathcal{G}_b$ have exactly the same structure.
- Therefore, identifying the configuration $\tilde{\mathbf{p}}$ "closest" to $\hat{\mathbf{p}}$ provides a principled way of identifying significant arcs and the confidence threshold.

The Confidence Threshold

Consider the order statistics $\tilde{\mathbf{p}}_{(\cdot)}$ and $\hat{\mathbf{p}}_{(\cdot)}$ and the cumulative distribution functions (CDFs) of their elements:

$$F_{\hat{\mathbf{p}}_{(\cdot)}}(x) = \frac{1}{k}\sum_{i=1}^{k} \mathbb{1}_{\{\hat{p}_{(i)}<x\}}$$

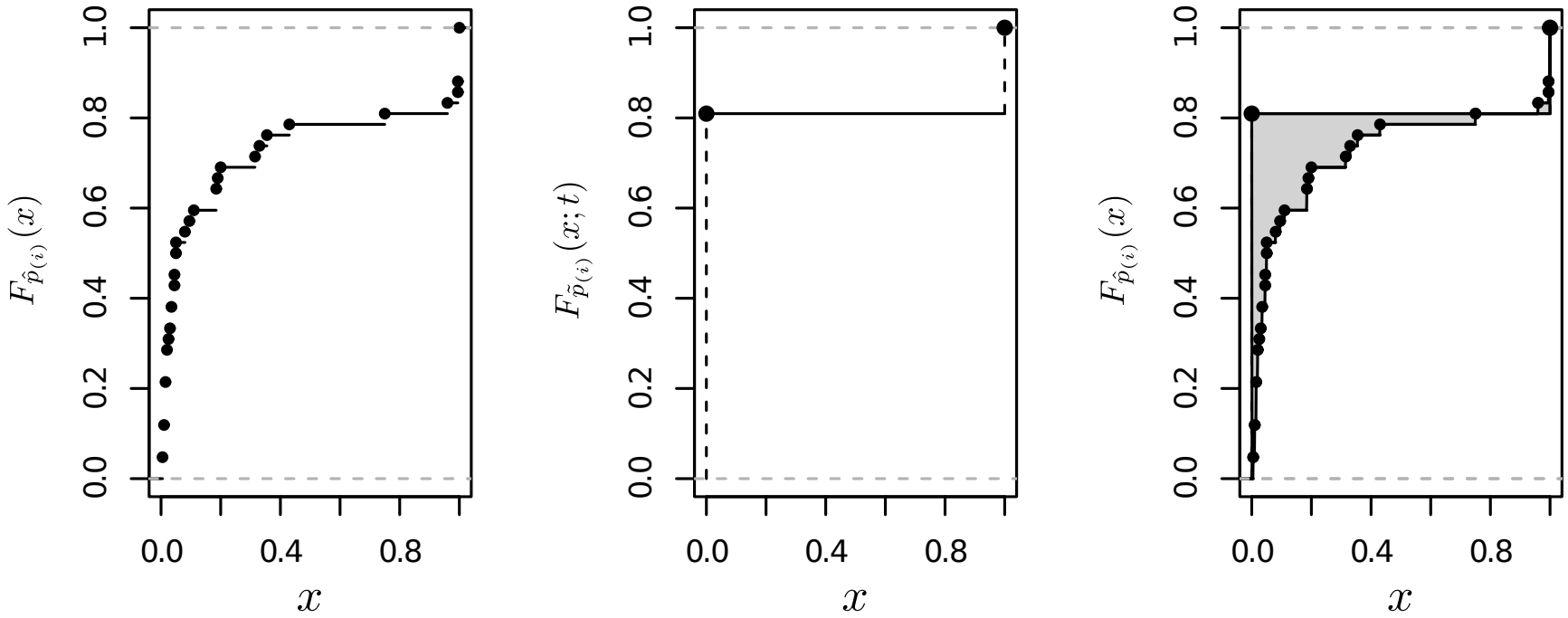
and

$$F_{\tilde{\mathbf{p}}_{(\cdot)}}(x;t) = \begin{cases} 0 & \text{if } x \in (-\infty, 0) \\ t & \text{if } x \in [0, 1) \\ 1 & \text{if } x \in [1, +\infty) \end{cases} .$$

t corresponds to the fraction of elements of $\tilde{\mathbf{p}}_{(\cdot)}$ equal to zero and is a measure of the fraction of non-significant arcs, and provides a threshold for separating the elements of $\tilde{\mathbf{p}}_{(\cdot)}$:

$$e_{(i)} \in E_0 \iff \hat{p}_{(i)} > F^{-1}_{\tilde{\mathbf{p}}_{(\cdot)}}(t).$$

The CDFs $F_{\hat{\mathbf{p}}_{(\cdot)}}(x)$ and $F_{\tilde{\mathbf{p}}_{(\cdot)}}(x;t)$



One possible estimate of t is the value $\hat{t}$ that minimises some distance between $F_{\hat{\mathbf{p}}_{(\cdot)}}(x)$ and $F_{\tilde{\mathbf{p}}_{(\cdot)}}(x;t)$; an intuitive choice is using the L_1 norm of their difference (i.e. the shaded area in the picture on the right).

An L_1 Estimator for the Confidence Threshold

Since $F_{\hat{\mathbf{p}}_{(\cdot)}}$ is piece-wise constant and $F_{\tilde{\mathbf{p}}_{(\cdot)}}$ is constant in $[0, 1]$, the L_1 norm of their difference simplifies to

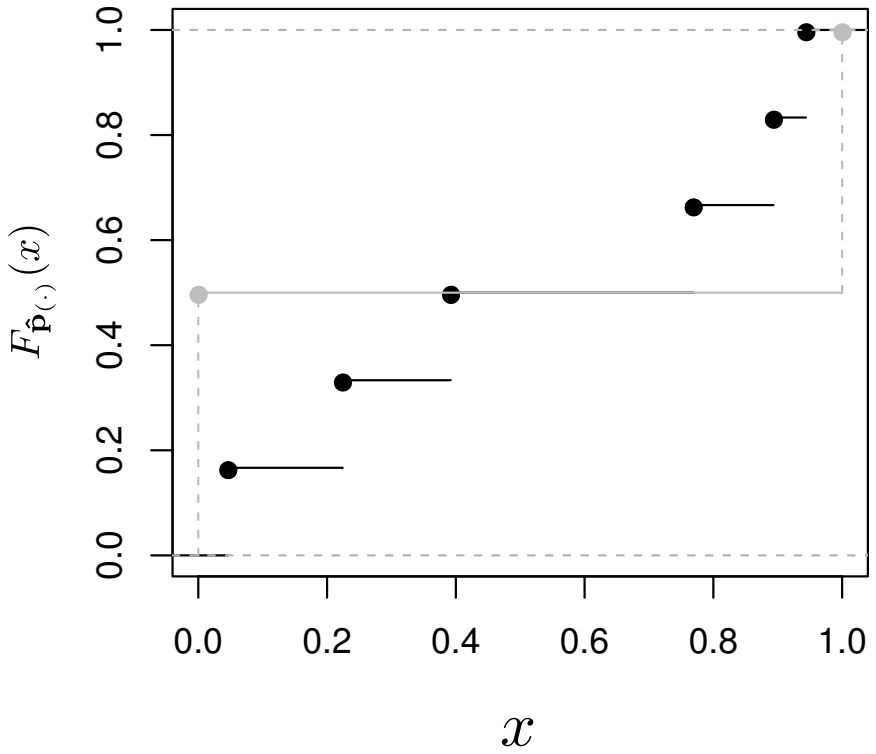
$$
\begin{aligned}
L_1\left(t; \hat{\mathbf{p}}_{(\cdot)}\right) &= \int \left| F_{\hat{\mathbf{p}}_{(\cdot)}}(x) - F_{\tilde{\mathbf{p}}_{(\cdot)}}(x; t) \right| dx \\
&= \sum_{x_i \in \left\{\{0\} \cup \hat{\mathbf{p}}_{(\cdot)} \cup \{1\}\right\}} \left| F_{\hat{\mathbf{p}}_{(\cdot)}}(x_i) - t \right| (x_{i+1} - x_i).
\end{aligned}
$$

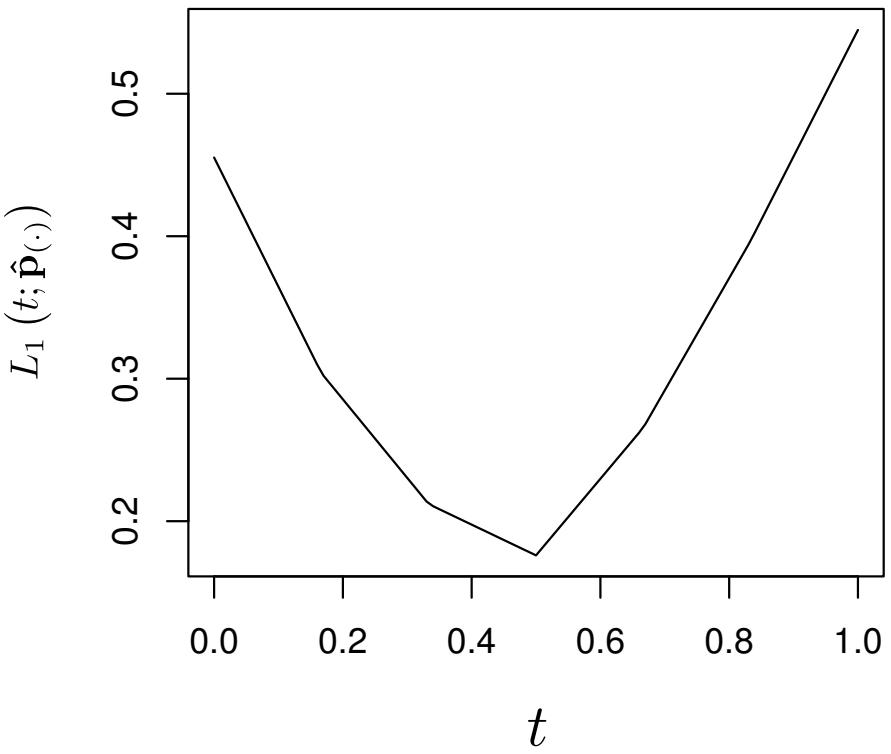
This form has two important properties:

- can be computed in linear time from $\hat{\mathbf{p}}_{(\cdot)}$;
- its minimisation is straightforward using linear programming.

Furthermore, the L_1 norm does not place as much weight on large deviations as other norms (L_2, L_∞), making it robust against a wide variety of configurations of $\hat{\mathbf{p}}_{(\cdot)}$.

A Simple Example





Consider a graph with 4 nodes and confidence values

$$\hat{\mathbf{p}}_{(\cdot)} = \{0.0460, 0.2242, 0.3921, 0.7689, 0.8935, 0.9439\}$$

Then $\hat{t} = \min_t L_1\left(t; \hat{\mathbf{p}}_{(\cdot)}\right) = 0.4999816$ and $F^{-1}_{\tilde{\mathbf{p}}_{(\cdot)}}(0.4999816) = 0.3921$; only three arcs are considered significant.

bnlearn: Model Averaging with averaged.network()

```
averaged.network(wstr)
```

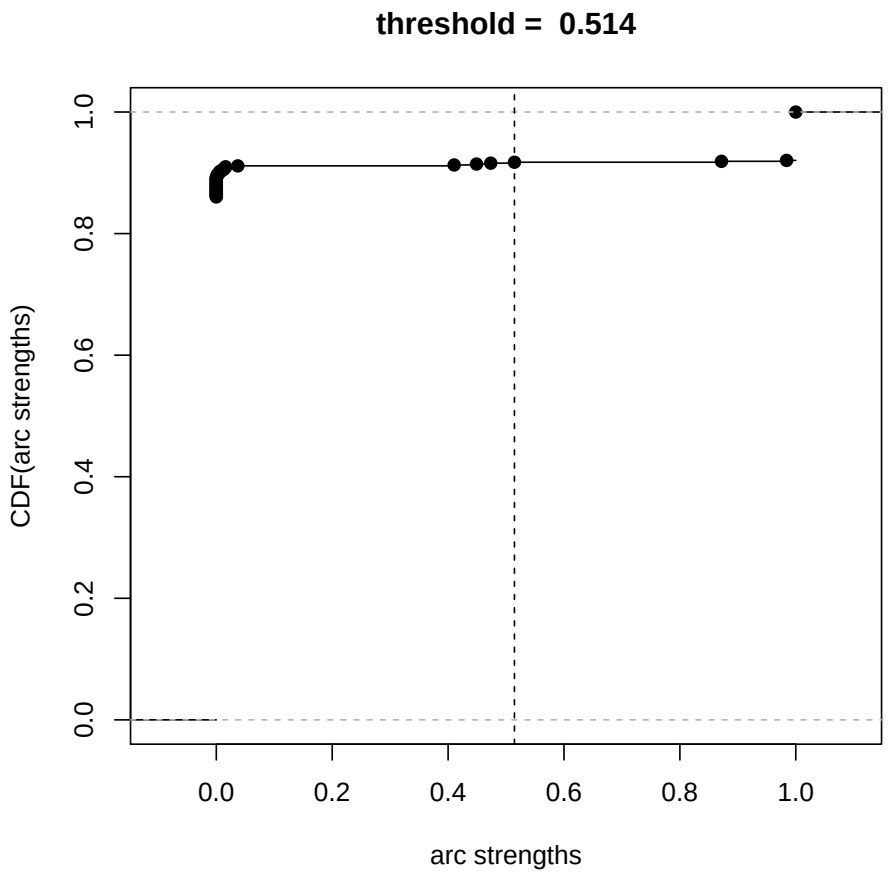
```
##
##   Random/Generated Bayesian network
##
##   model:
##     [partially directed graph]
##   nodes:                                  37
##   arcs:                                   55
##     undirected arcs:                      3
##     directed arcs:                        52
##   average markov blanket size:            3.57
##   average neighbourhood size:             2.97
##   average branching factor:               1.35
##
##   generation algorithm:                   Model Averaging
##   significance threshold:                 0.514
```

```
head(wstr[wstr$strength > 0.514 & wstr$direction >= 0.50, ], n = 3)
```

```
##     from  to strength direction
## 60  PCWP LVV        1       0.5
## 112  TPR  BP        1       1.0
## 126  TPR APL        1       1.0
```

bnlearn: Plotting the ECDF

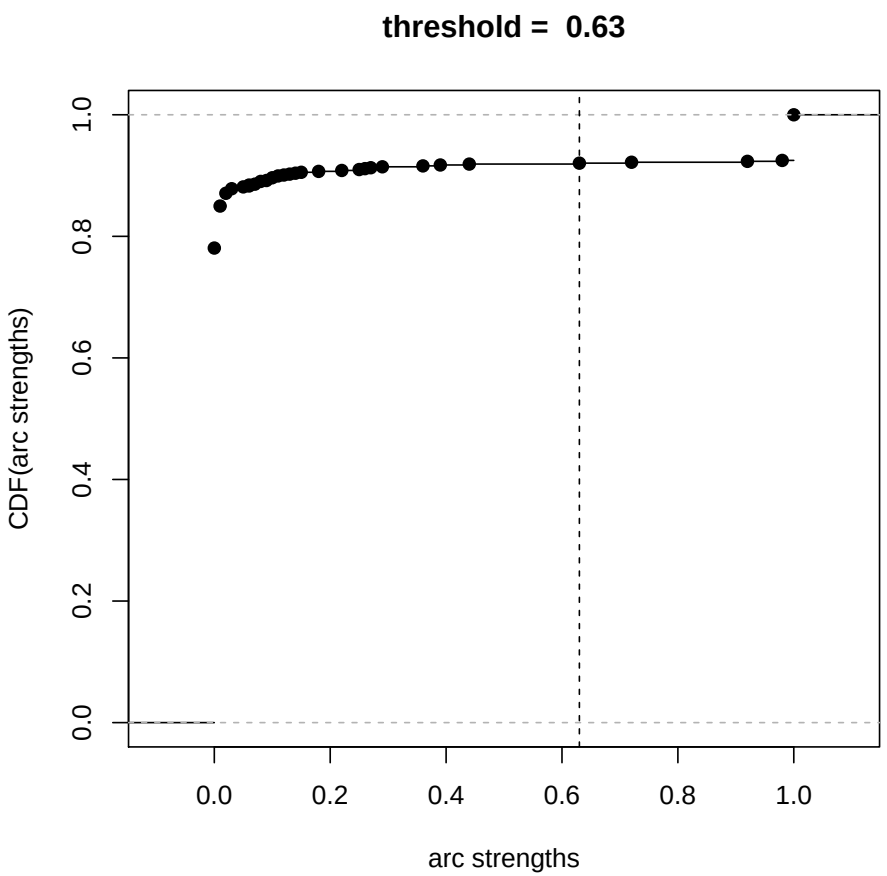
```
plot(wstr)
```



The effect of the uneven posterior probability is apparent from the fact that the arc weights are essentially either zero or one.

bnlearn: Plotting the ECDF

```
plot(str)
```



With the frequentist approach the weights are more spread out, and the threshold is different as a result.

bnlearn: Custom Thresholds

`averaged.network()` accepts custom values for the threshold, so we can investigate its on the resulting (CP)DAG.

```
unlist(compare(averaged.network(wstr), true.dag))
```

```
## tp fp fn
## 23 23 32
```

```
unlist(compare(averaged.network(str), true.dag))
```

```
## tp fp fn
## 22 24 31
```

```
unlist(compare(averaged.network(str, threshold = 0.4), true.dag))
```

```
## tp fp fn
## 22 24 33
```

```
unlist(compare(averaged.network(str, threshold = 0.8), true.dag))
```

```
## tp fp fn
## 22 24 30
```

There is not guarantee that the L_1 norm with produce the best DAG, say, that with the lowest SHD, but simulations and real-world data analyses suggest it performs well enough for practical purposes.

Summary

- Scoring the DAGs we evaluate in structure learning algorithms is crucial, but so are our assumptions on their prior probability.
- We can incorporate prior knowledge in structure learning in many ways with hard constraints (arcs being present or absent, maximum number of arcs) and/or informative priors (probability of parents and arcs). If the prior knowledge we have is not wrong, this augments the information present in the data and improves the quality of the BN.
- Even if we have no prior knowledge, we can do better than assuming a uniform prior.
- Estimating the parameters of a BN given the DAG is comparatively easy; smooth estimates are preferable over maximum likelihood estimates as usual.
- We can use resampling to remove noisy arcs with model averaging, typically along the lines of bagging. Averaged models tend to be more robust and better at prediction.

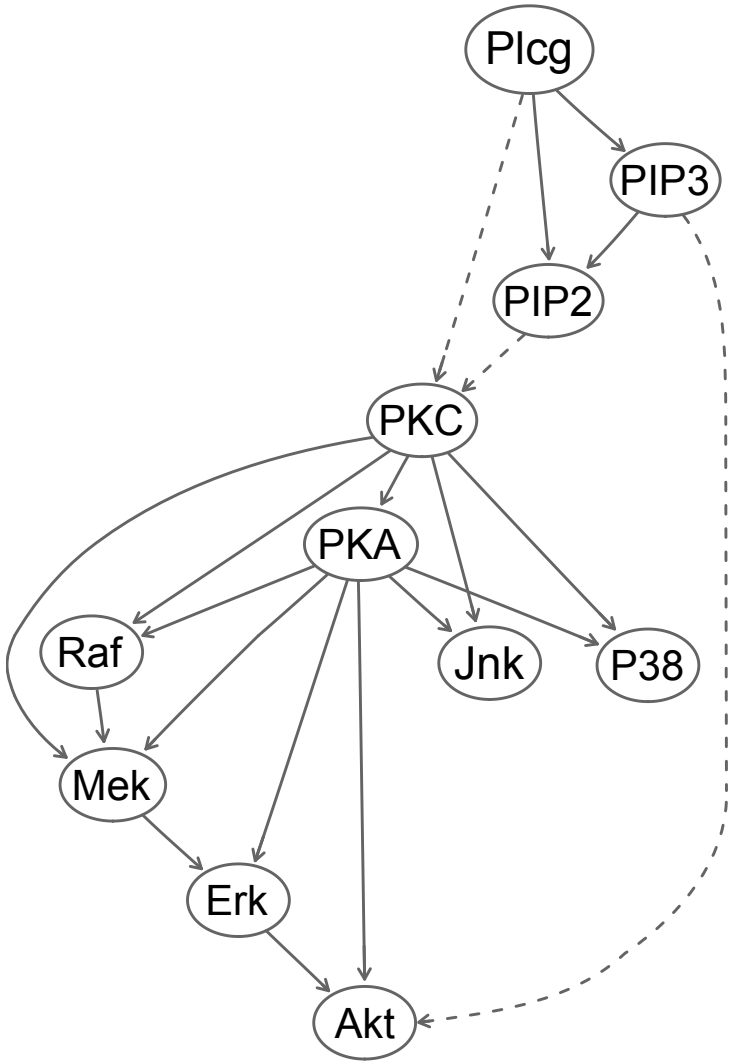
Hands-On Examples

Case Study: Human Physiology

Causal Protein-Signalling Networks Derived from Multiparameter Single Cell Data.
Karen Sachs, *et al.*, Science, **308**, 523 (2005).

That is a landmark application of BNs because it highlights the use of interventional data; and because results are validated. The data consist in the 5400 simultaneous measurements of 11 phosphorylated proteins and phospholypids; 1800 are subjected to spiking and knock-outs to control expression.

The goal of the analysis is to learn what relationships link these 11 proteins, that is, the signalling pathways they are part of.



Exploring the Data

```
sachs = read.table("sachs.data.txt", header = TRUE)
head(sachs, n = 5)
```

```
##    Raf  Mek  Plcg  PIP2  PIP3   Erk  Akt PKA   PKC  P38  Jnk
## 1 26.4 13.2  8.82 18.30 58.80  6.61 17.0 414 17.00 44.9 40.0
## 2 35.9 16.5 12.30 16.80  8.13 18.60 32.5 352  3.37 16.5 61.5
## 3 59.4 44.1 14.60 10.20 13.00 14.90 32.5 403 11.40 31.9 19.5
## 4 73.0 82.8 23.10 13.50  1.29  5.83 11.8 528 13.70 28.6 23.1
## 5 33.7 19.8  5.19  9.73 24.80 21.10 46.1 305  4.66 25.7 81.3
```

The variables represent concentrations of the proteins and the phospholypids, and take positive values. For some variables, and observations, the cells were stimulated to produce artificially high or low levels of particular proteins:

- 1800 data subject only to general stimolatory cues, so that the protein signalling paths are active;
- 600 data with with specific stimolatory/inhibitory cues for each of the following 4 proteins: Mek, PIP2, Akt, PKA;
- 1200 data with specific cues for PKA.

A First Try

```
dag.hiton = si.hiton.pc(sachs, test = "cor", undirected = FALSE)
directed.arcs(dag.hiton)
```

```
##      from  to
## [1,] "P38" "PKC"
## [2,] "Jnk" "PKC"
```

```
undirected.arcs(dag.hiton)
```

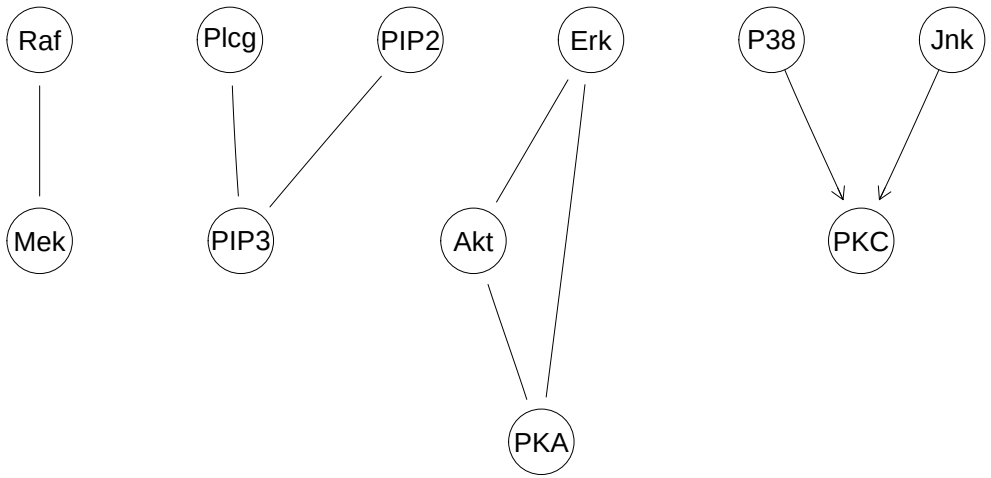
```
##       from   to
##  [1,] "Raf"  "Mek"
##  [2,] "Mek"  "Raf"
##  [3,] "Plcg" "PIP3"
##  [4,] "PIP2" "PIP3"
##  [5,] "PIP3" "Plcg"
##  [6,] "PIP3" "PIP2"
##  [7,] "Erk"  "Akt"
##  [8,] "Erk"  "PKA"
##  [9,] "Akt"  "Erk"
## [10,] "Akt"  "PKA"
## [11,] "PKA"  "Erk"
## [12,] "PKA"  "Akt"
```

Compare with the Validated Model

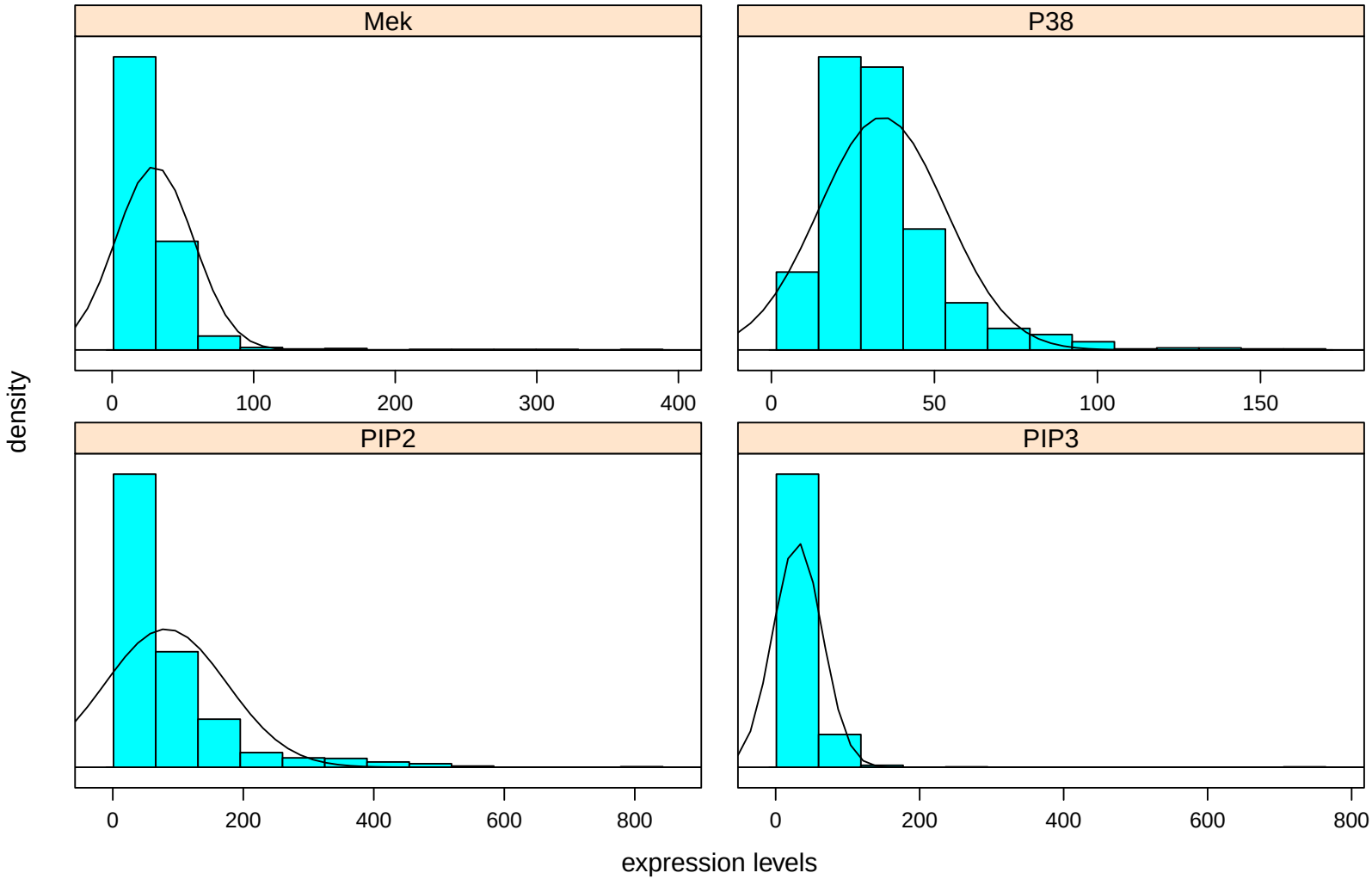
```
sachs.modelstring =
  paste("[PKC][PKA|PKC][Raf|PKC:PKA][Mek|PKC:PKA:Raf][Erk|Mek:PKA]",
        "[Akt|Erk:PKA][P38|PKC:PKA][Jnk|PKC:PKA][Plcg][PIP3|Plcg]",
        "[PIP2|Plcg:PIP3]")
dag.sachs = model2network(sachs.modelstring)
unlist(compare(dag.sachs, dag.hiton))
```

```
## tp fp fn
##  0  8 17
```

```
graphviz.plot(dag.hiton)
```

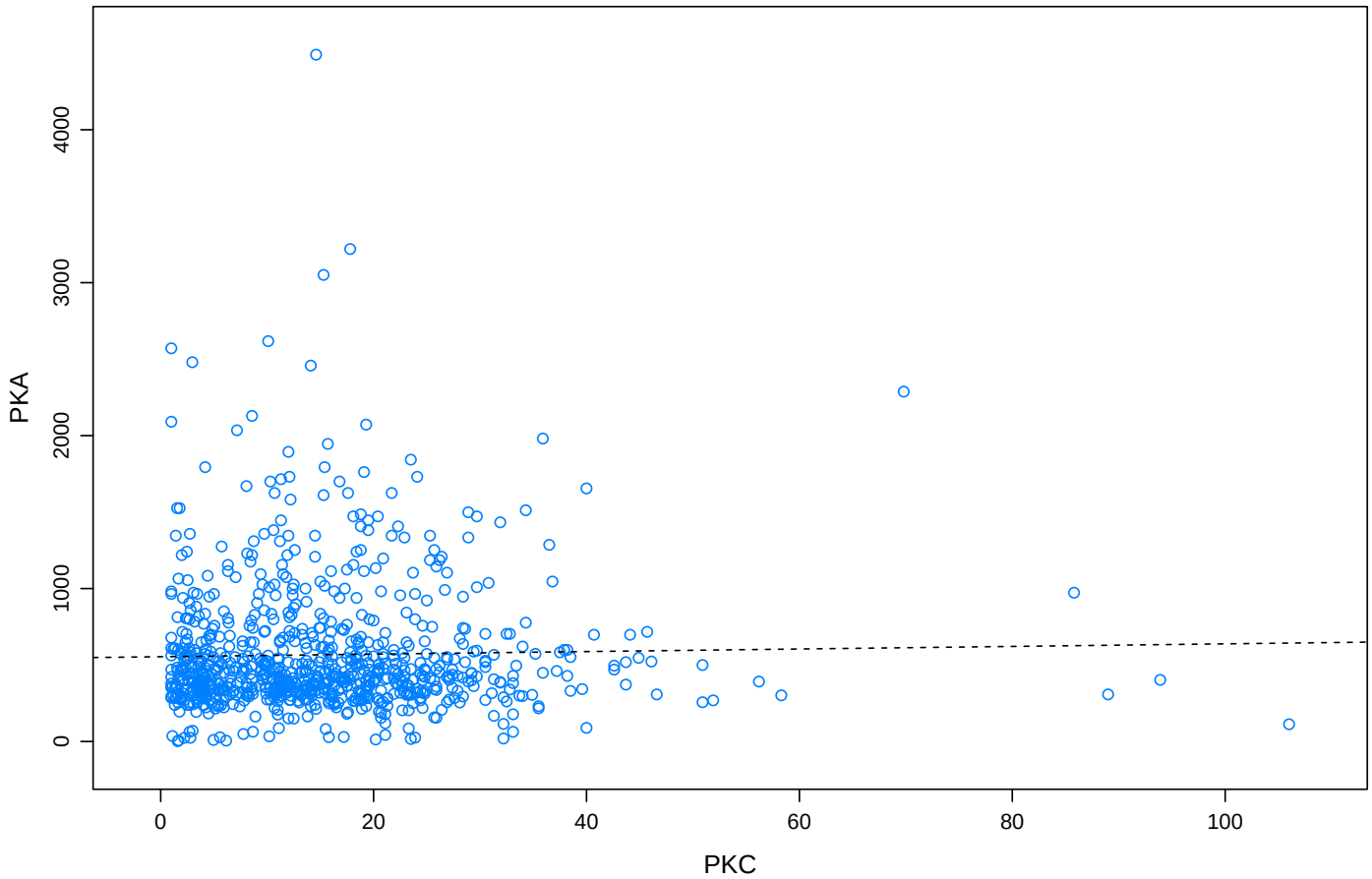


Are Variables Normally Distributed?



Variables are skewed and bounded below by zero, which makes them very different from a normal distribution. So, using a GBN may not be a good idea...

Are Dependencies Linear?



There is a $\mathrm{PKC} \rightarrow \mathrm{PKA}$ arc in the validated network, and PKC is the only parent of PKA. However, we cannot see any linear relationship...

What to Do Now?

Since GBNs are not appropriate, we must now consider alternatives:

- We explore monotone transformations like the $\log_1 0$ (tried, no improvements).
- We specify an appropriate conditional distribution for each variable using prior knowledge on the signalling pathways (which may or may not be available). However, the aim of the analysis was to use BNs as an automated probabilistic method to verify such information, not to build a BN with prior information and use it as an expert system.
- Discretise the data and to model them with a DBN, which can accommodate skewness and nonlinear relationships at the cost of potentially losing the ordering information. Since the variables in the BN represent concentration levels, Sachs et al. used three levels corresponding to low, average and high concentrations.

Hartemink's Information-Preserving Discretisation

Input: a data set $\mathbf{X} = X_i, i = 1, \ldots, N$ where all X_i are continuous variables.
Output: a data set with N discrete variables, each with k_2 levels.

1. Discretise each variable independently using quantile discretisation and a large number k_1 of intervals, e.g., $k_1 = 50$ or even $k_1 = 100$.
2. Repeat the following steps until each variable has $k_2 \ll k_1$ intervals, iterating over each variable X_i, $i = 1, \ldots, N$ in turn:
   2.1 compute
   $$\mathrm{M}_{X_i} = \sum_{j \neq i} \mathrm{MI}(X_i, X_j);$$
   2.2 for each pair l of adjacent intervals of X_i, collapse them in a single interval, and with the resulting variable $X_i^*(l)$ compute
   $$\mathrm{M}_{X_i^*(l)} = \sum_{j \neq i} \mathrm{MI}(X_i^*(l), X_j);$$
   2.3 set $X_i = \mathrm{argmax}_{X_i(l)} \mathrm{M}_{X_i^*(l)}$.

bnlearn: Discretising Data

An implementation of Hartemink's algorithm is provided in `discretize()`, which takes k_2 (`breaks`), k_1 (`ibreaks`) and the initial discretisation algorithm (`idisc`).

```
dsachs = discretize(sachs, method = "hartemink",
            breaks = 3, ibreaks = 60, idisc = "quantile")
head(dsachs)
```

```
##            Raf         Mek       Plcg        PIP2         PIP3          Erk
## 1 (1.61,39.5]   (1,21.1]     (1,12] (1.11,34.9]  (50.9,764]    (1,15.3]
## 2 (1.61,39.5]   (1,21.1]  (12,23.1] (1.11,34.9]    (1,18.9] (15.3,29.4]
## 3 (39.5,62.6] (27.4,389]  (12,23.1] (1.11,34.9]    (1,18.9]    (1,15.3]
## 4  (62.6,552] (27.4,389] (23.1,167] (1.11,34.9]    (1,18.9]    (1,15.3]
## 5 (1.61,39.5]   (1,21.1]     (1,12] (1.11,34.9] (18.9,50.9] (15.3,29.4]
## 6 (1.61,39.5]   (1,21.1]  (12,23.1] (1.11,34.9]    (1,18.9]    (1,15.3]
##            Akt         PKA          PKC          P38          Jnk
## 1  (1.7,23.5] (1.95,547] (9.73,20.2]  (33.4,170]  (35.9,343]
## 2 (23.5,46.1] (1.95,547]    (1,9.73] (1.53,19.9]  (35.9,343]
## 3 (23.5,46.1] (1.95,547] (9.73,20.2] (19.9,33.4] (18.4,35.9]
## 4  (1.7,23.5] (1.95,547] (9.73,20.2] (19.9,33.4] (18.4,35.9]
## 5 (23.5,46.1] (1.95,547]    (1,9.73] (19.9,33.4]  (35.9,343]
## 6 (23.5,46.1]  (547,777] (9.73,20.2]  (33.4,170]  (35.9,343]
```

Structure Learning and Model Averaging

However, HITON is still not working...

```
dag.hiton = si.hiton.pc(dsachs, test = "x2", undirected = FALSE)
unlist(compare(dag.hiton, dag.sachs))
```

```
## tp fp fn
##  0 17 10
```

... so we switch to a score-based algorithm ...

```
dag.hc = hc(dsachs, score = "bde", iss = 10, undirected = FALSE)
unlist(compare(dag.hc, dag.sachs))
```

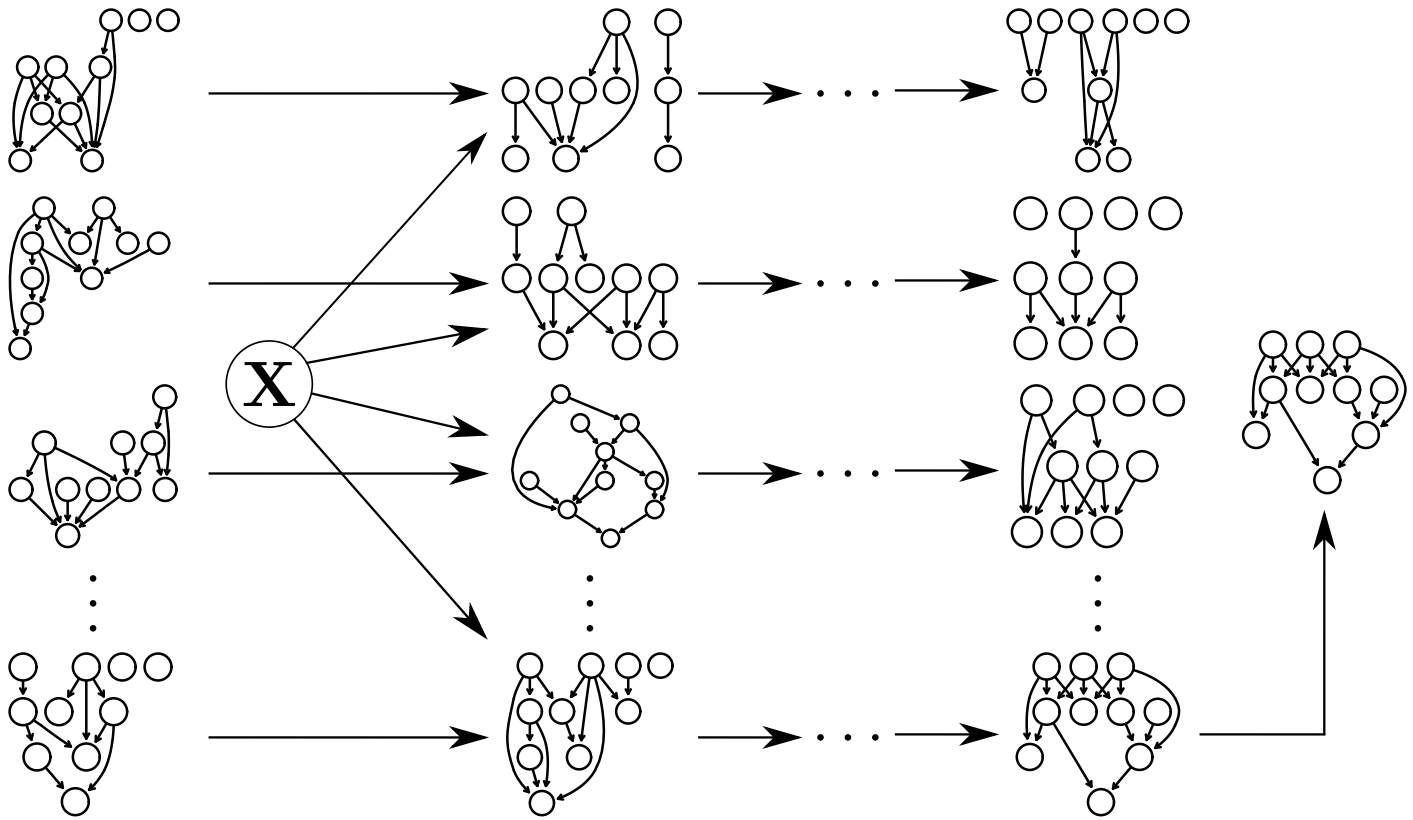
```
## tp fp fn
##  6 11  4
```

... and frequentist model averaging to remove spurious arcs.

```
boot = boot.strength(dsachs, R = 500, algorithm = "hc",
          algorithm.args = list(score = "bde", iss = 10))
head(boot[(boot$strength > 0.85) & (boot$direction >= 0.5), ], n = 3)
```

```
##    from   to strength direction
## 1   Raf  Mek    1.000     0.512
## 23 Plcg PIP2    0.998     0.510
## 24 Plcg PIP3    1.000     0.527
```

Learning Multiple DAGs from the Data



Searching from different starting points increases our coverage of the space of the possible DAGs; the frequency with which an arc appears is a measure of the strength of the dependence.

Model Averaging from Multiple Searches

While there is no function in **bnlearn** that does exactly this, we can combine `random.graph()` and `sapply()` to generate the random starting points and call `hc()` on each of them.

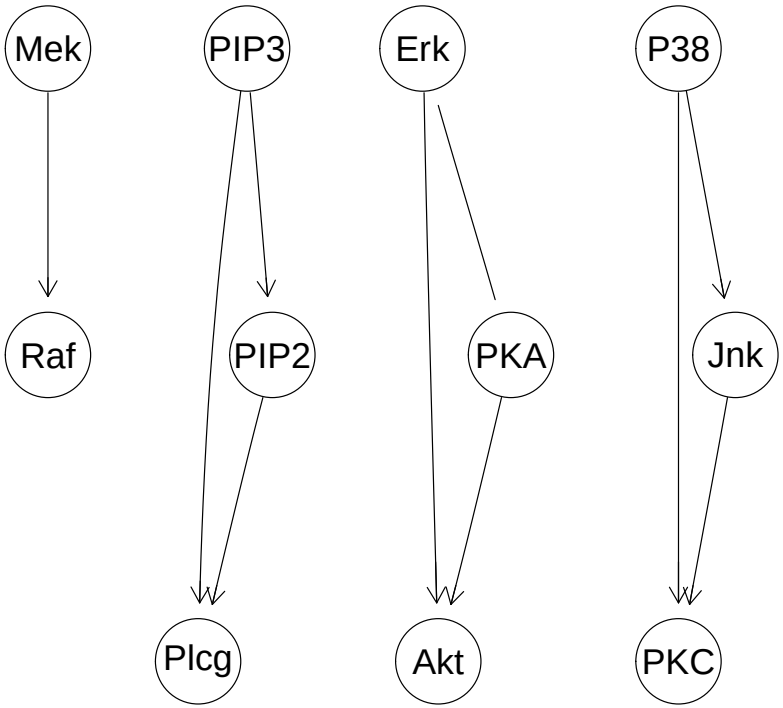
```
nodes = names(dsachs)
start = random.graph(nodes = nodes, method = "ic-dag",
           num = 500, every = 50)
netlist = lapply(start,
  function(net) {
    hc(dsachs, score = "bde", iss = 10, start = net)
  }
)
```

Then we can take the resulting list and pass it to `custom.strength()` to compute arc strengths.

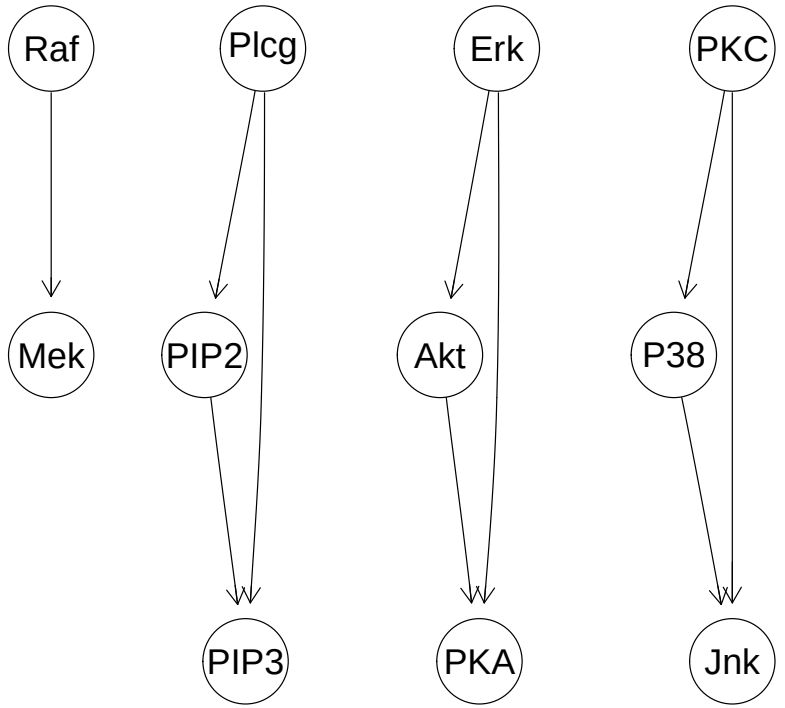
```
start = custom.strength(netlist, nodes = nodes)
```

Compare Both Approaches with the Validated Network

```
avg.start = averaged.network(start)
graphviz.plot(avg.start)
```




```
avg.boot = averaged.network(boot)
graphviz.plot(avg.boot)
```



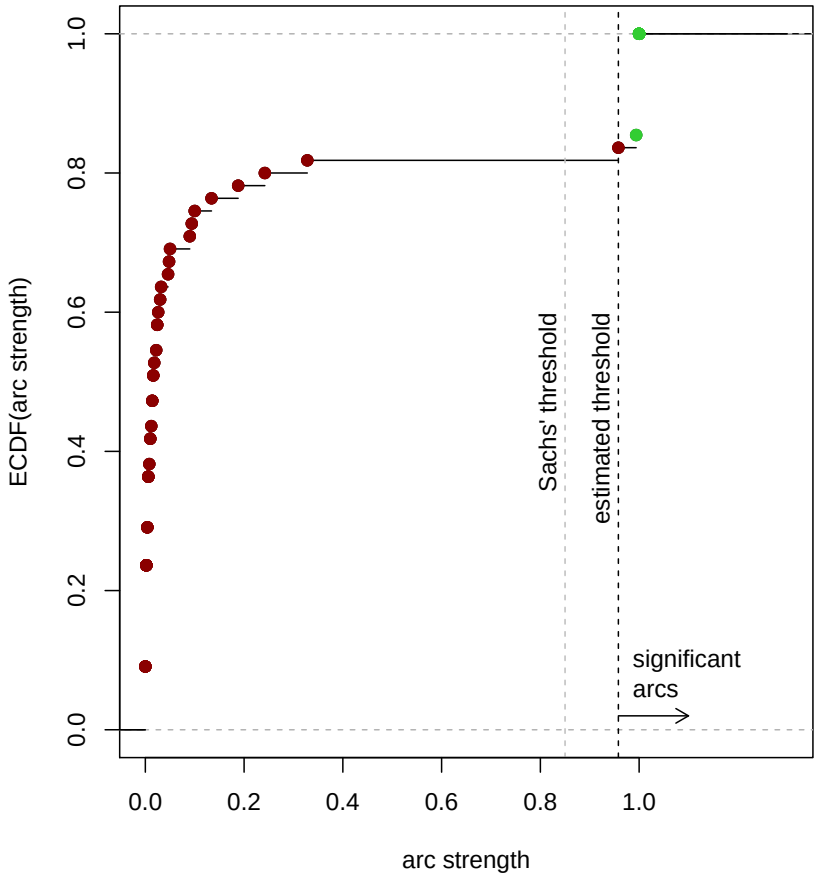

```
unlist(compare(avg.start, dag.sachs))
```

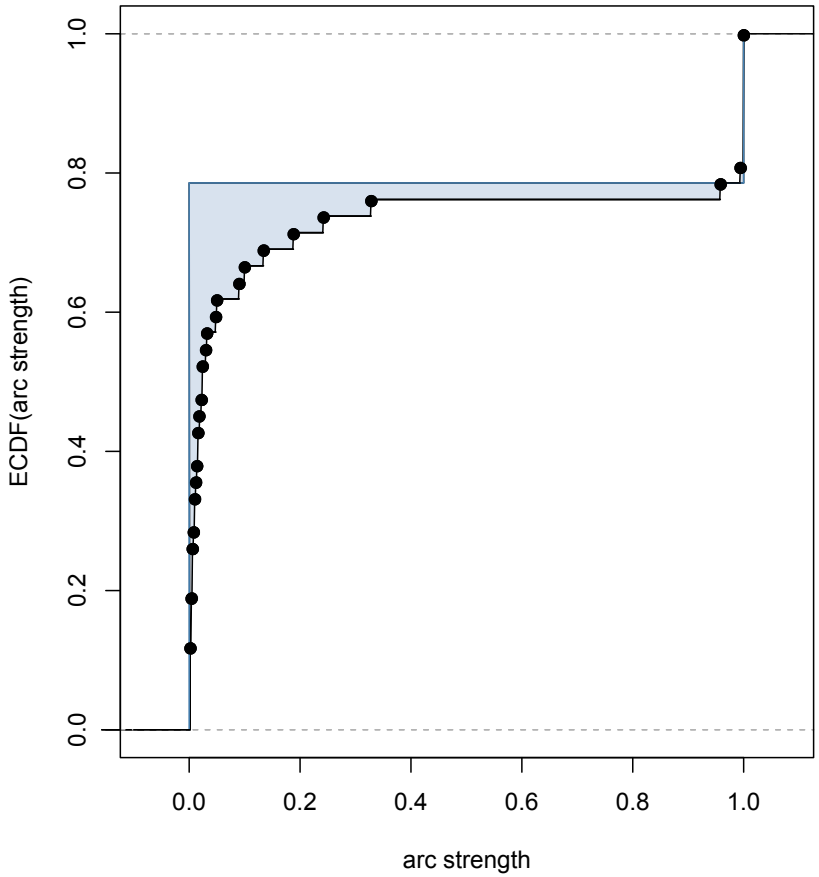
```
## tp fp fn
##  3 14  7
```

```
unlist(compare(avg.boot, dag.sachs))
```

```
## tp fp fn
##  6 11  4
```

Model Averaging for the Bootstrapped DAGs





Arcs with significant strength can be identified using a threshold estimated from the data by minimising the distance from the observed ECDF and the ideal, asymptotic one (the blue area in the right panel).

Taking the Interventions into Account

Both networks look nothing like the validated network, and in fact fall in the same equivalence class.

```
all.equal(cpdag(avg.boot), cpdag(avg.start))
## [1] TRUE
```

The only piece of information we have not taken into account yet are the stimulations and the inhibitions, that is, the interventions on the variables.

```
isachs = read.table("sachs.interventional.txt",
            header = TRUE, colClasses = "factor")
```

With the discretised data, for each variable:

- an inhibition is an ideal intervention that sets the value to "low";
- a stimulations is an ideal intervention that sets the value to "high".

A Naive Approach with Whitelists

A naive approach to consider the intervention variable `INT` would be to include it as a node in the DAG and whitelist outgoing arcs to all other variables to have different conditional probabilities depending on whether each observation is subject to an intervention.

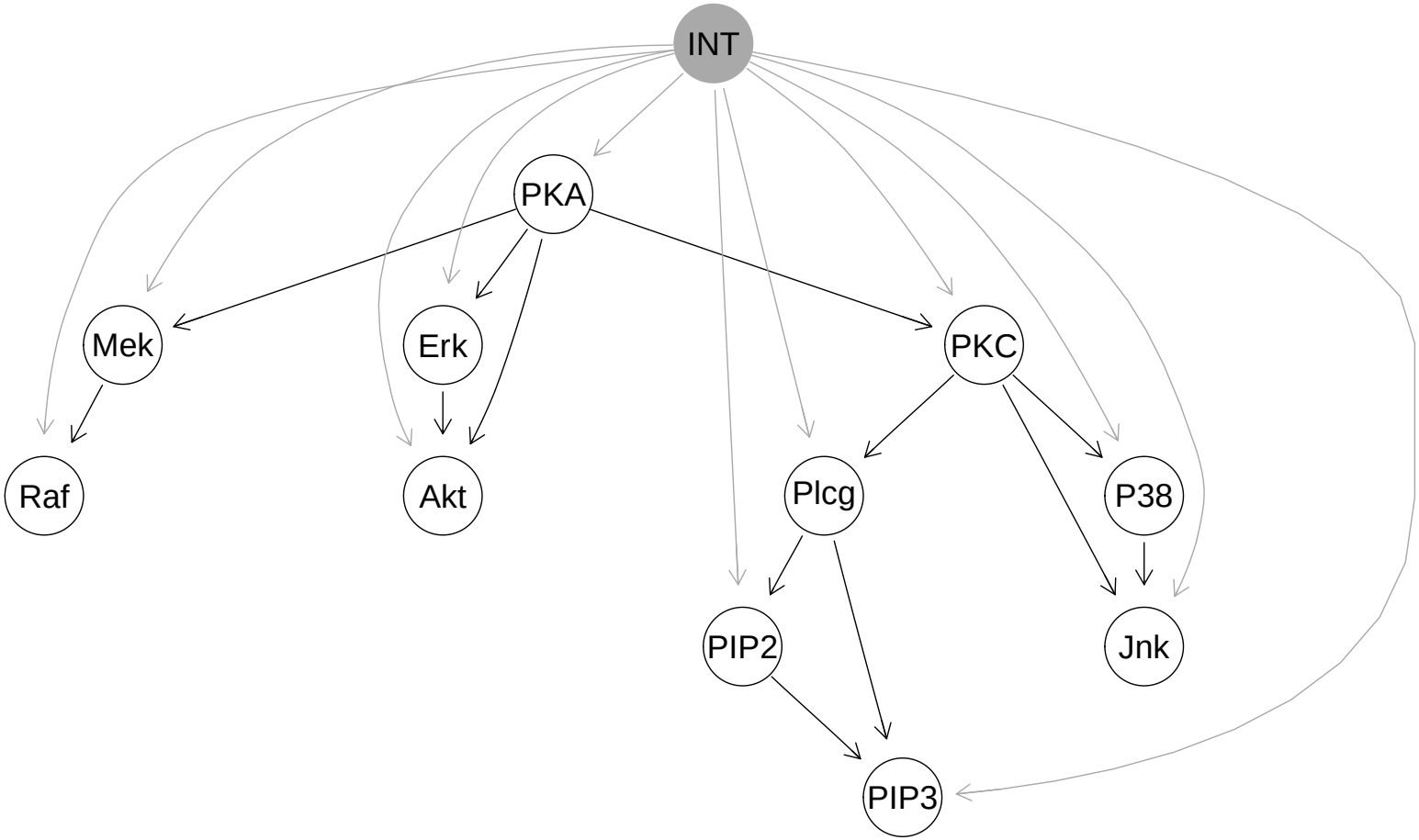
```
wh = matrix(c(rep("INT", 11), names(isachs)[1:11]), ncol = 2)
dag.wh = tabu(isachs, whitelist = wh, score = "bde",
            iss = 10, tabu = 50)
unlist(compare(subgraph(dag.wh, names(isachs)[1:11]), dag.sachs))
```

```
## tp fp fn
##  8  9  5
```

This works better than before, but we still do not get the validated network. Note that in this case we compare DAGs directly and not CPDAGs because the interventions break score equivalence by blocking the effect encoded by incoming arcs for some combinations of nodes and observations.

A Naive Approach with Whitelists

```
graphviz.plot(dag.wh, highlight = list(nodes = "INT",
  arcs = outgoing.arcs(dag.wh, "INT"), col = "darkgrey", fill = "darkgrey"))
```



Mixed Observational and Interventional Data

A more granular way of doing the same thing is to use the mixed observational and interventional data posterior score from Cooper & Yoo, which creates an implicit intervention binary node for each variable.

```
INT = sapply(1:11, function(x) which(isachs$INT == x) )
nodes = names(isachs)[1:11]
names(INT) = nodes
```

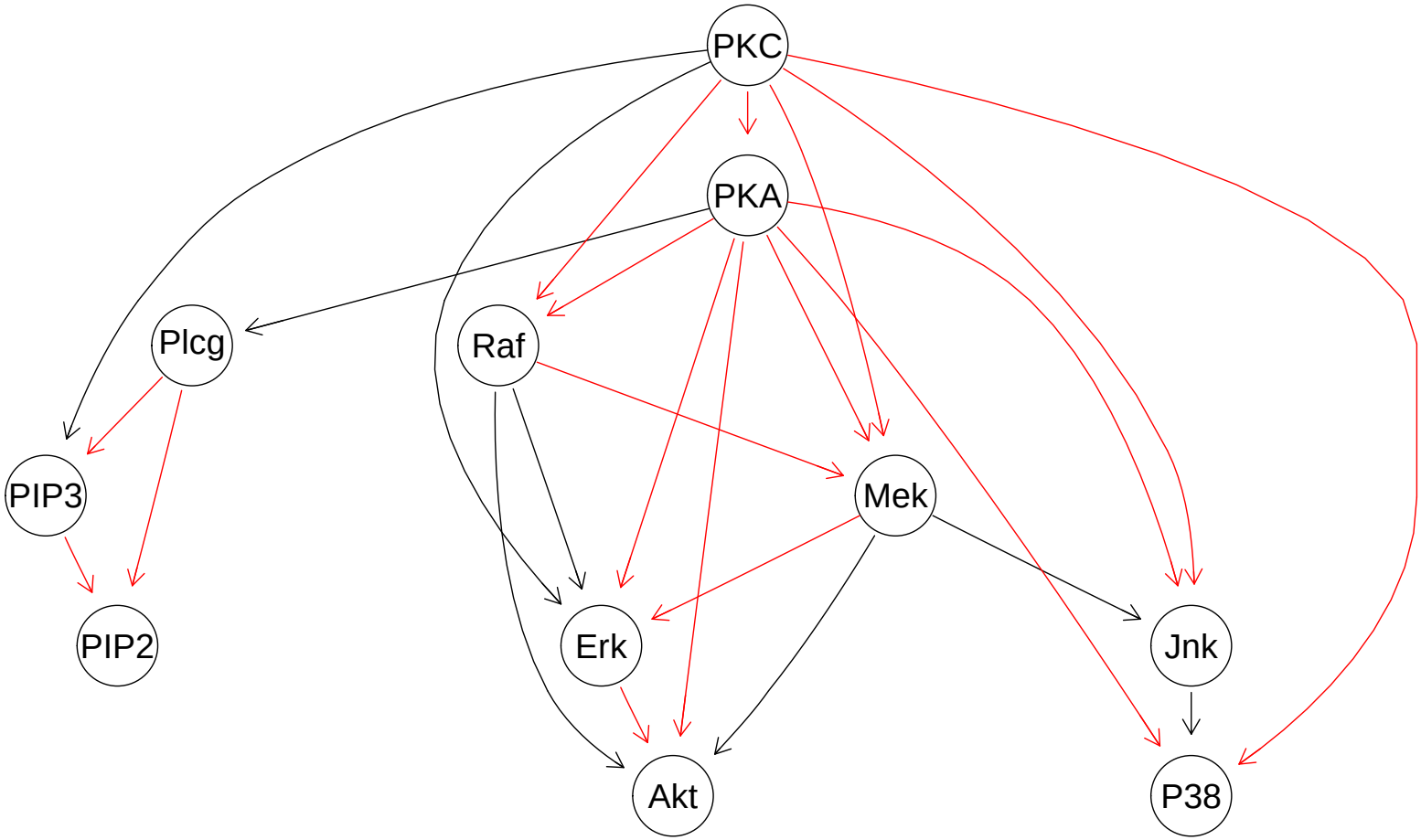
Then we perform model averaging of the resulting causal DAGs, with better results.

```
netlist = lapply(start, function(net) {
  tabu(isachs[, 1:11], score = "mbde", exp = INT, iss = 1,
    start = net, tabu = 50)
})
intscore = custom.strength(netlist, nodes = nodes, cpdag = FALSE)
dag.mbde = averaged.network(intscore)
unlist(compare(dag.sachs, dag.mbde))
```

```
## tp fp fn
## 17  8  0
```

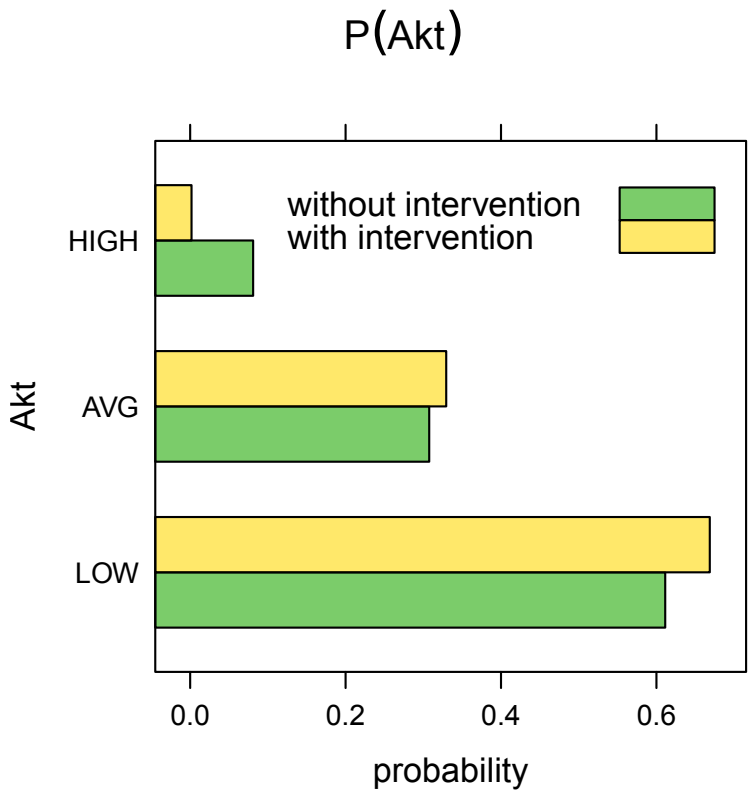
The Final DAG

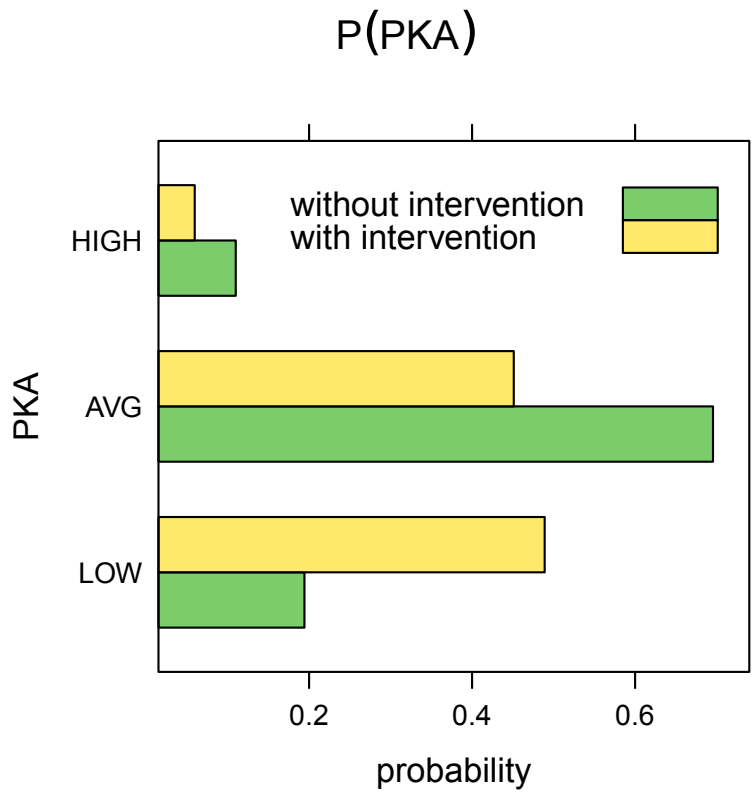
```
graphviz.plot(dag.mbde, highlight = list(arcs = arcs(dag.sachs)))
```



Using The Protein Network to Plan Experiments

This idea goes by the name of hypothesis generation: using a statistical model to decide which follow-up experiments to perform. BNs are especially easy to use for this because they automate the computation of arbitrary events.





Fitting the Parameters and Performing Queries

First, we need to learn the parameters of the BN given the DAG.

```
isachs = isachs[, 1:11]
for (i in names(isachs))
  levels(isachs[, i]) = c("LOW", "AVG", "HIGH")
fitted = bn.fit(dag.sachs, isachs, method = "bayes")
```

Then we can proceed to perform queries using **gRain**, on the original BN

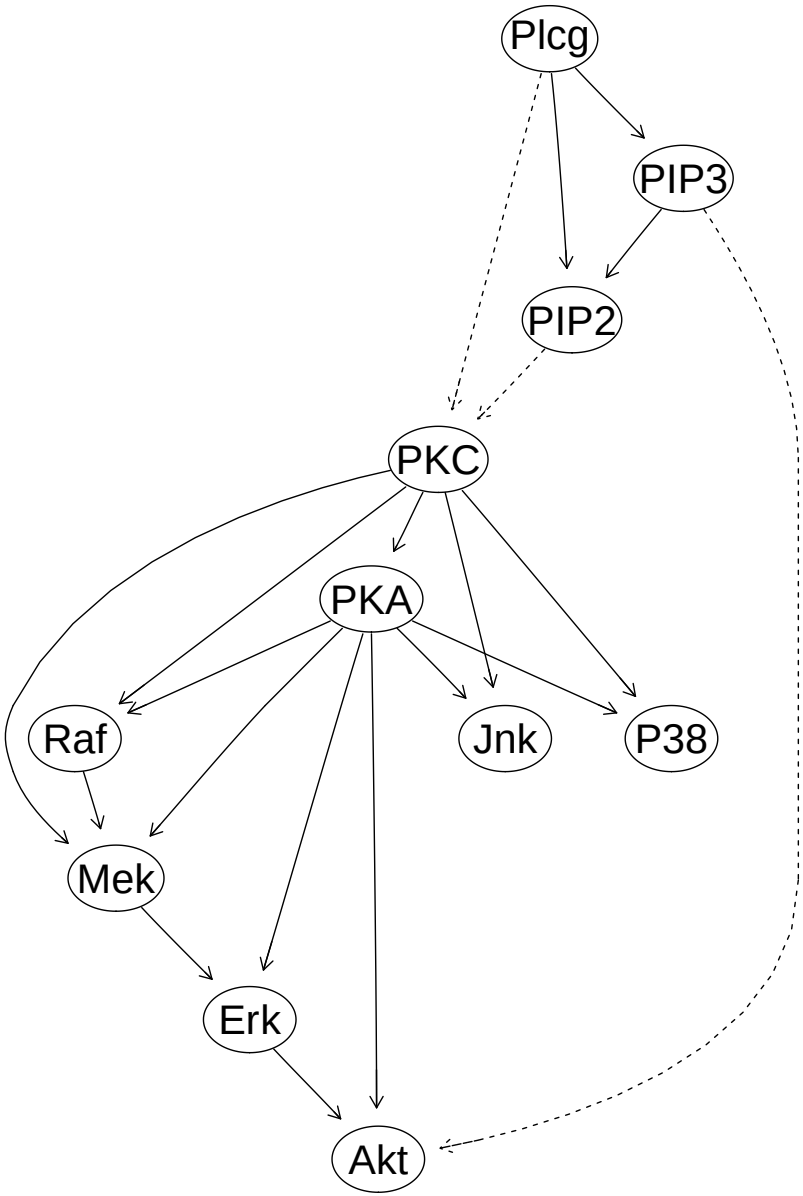
```
library(gRain)
jtree = compile(as.grain(fitted))
```

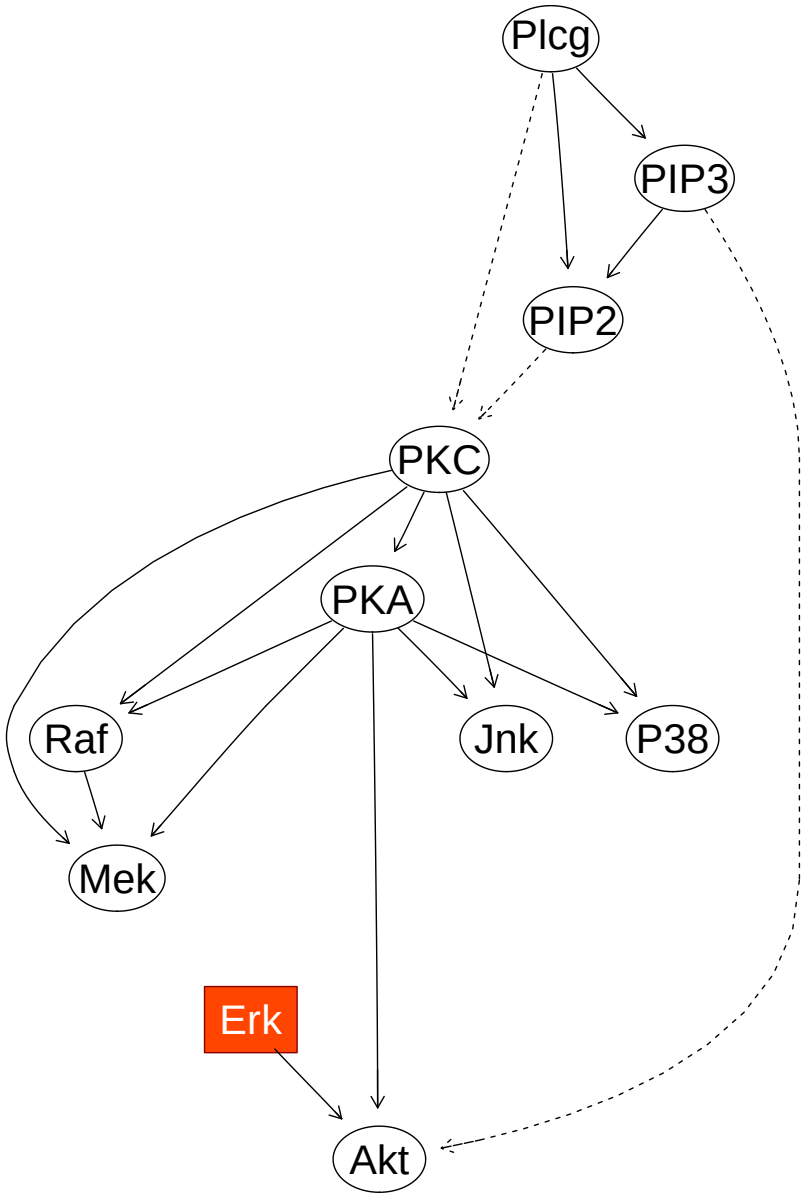
and on a mutilated BN in which we set `Erk` to `LOW` with an ideal intervention.

```
jlow = compile(as.grain(mutilated(fitted, evidence = list(Erk = "LOW"))))
```

In other words, we simulate a lab experiment in which we inhibit `Erk` (called a knock-out experiment). Much cheaper than actually doing it for real!

Interventions and Mutilated Graphs





Variables That are Downstream are Untouched

The marginal distribution of `Akt` changes depending on whether we take the evidence (intervention) into account or not.

```
querygrain(jtree, nodes = "Akt")$Akt
```

```
## Akt
##    LOW    AVG   HIGH
## 0.6089 0.3104 0.0807
```

```
querygrain(jlow, nodes = "Akt")$Akt
```

```
## Akt
##    LOW    AVG   HIGH
## 0.6671 0.3310 0.0019
```

The slight inhibition of `Akt` induced by the inhibition of `Erk` agrees with both the direction of the arc linking the two nodes and the additional experiments performed by Sachs et al. In causal terms, the fact that changes in `Erk` affect `Akt` supports the existence of a causal link from the former to the latter.

Causal Inference, Posterior Inference

If there is no causal link from the variable subject to intervention (`Erk`) to another variable (say `PKA`), the distribution of that variable will not be impacted by the intervention.

```
querygrain(jtree, nodes = "PKA")$PKA
```

```
## PKA
##   LOW   AVG  HIGH
## 0.194 0.696 0.110
```

```
querygrain(jlow, nodes = "PKA")$PKA
```

```
## PKA
##   LOW   AVG  HIGH
## 0.194 0.696 0.110
```

This is unlike posterior inference, because we do not remove `Erk`'s parents in that case.

```
jlow = setEvidence(jtree, nodes = "Erk", states  = "LOW")
querygrain(jlow, nodes = "PKA")$PKA
```

```
## PKA
##    LOW    AVG   HIGH
## 0.4891 0.4512 0.0597
```

Case Study: Plant Genetics

DNA data (e.g. SNP markers) is routinely used in statistical genetics to understand the genetic basis of human diseases, and to breed traits of commercial interest in plants and animals. Multiparent (MAGIC) populations are ideal for the latter. Here we consider a wheat population: 721 varieties, 16K genetic markers, 7 traits. (I ran the same analysis on a rice population, 1087 varieties, 4K markers, 10 traits, with similar results.)

Phenotypic traits for plants typically include flowering time, height, yield, a number of disease scores. The goal of the analysis is to find key genetic markers controlling the traits; to identify any causal relationships between them; and to keep a good predictive accuracy.

Genetics Society of America

Multiple Quantitative Trait Analysis Using Bayesian Networks
Marco Scutari, *et al.*, Genetics, **198**, 129–137 (2014);
DOI: 10.1534/genetics.114.165704

Bayesian Networks in Genetics

If we have a set of traits and markers for each variety, all we need are the Markov blankets of the traits; most markers are discarded in the process. Using common sense, we can make some assumptions:

- traits can depend on markers, but not vice versa;
- dependencies between traits should follow the order of the respective measurements (e.g. longitudinal traits, traits measured before and after harvest, etc.);
- dependencies in multiple kinds of genetic data (e.g. SNP + gene expression or SNPs + methylation) should follow the central dogma of molecular biology.

Assumptions on the direction of the dependencies allow to reduce Markov blankets learning to learning the parents and the children of each trait, which is a much simpler task.

Parametric Assumptions

In the spirit of classic additive genetics models, we use a Gaussian BN. Then the local distribution of each trait T_i is a linear regression model

$$\begin{aligned} T_i &= \boldsymbol{\mu}_{T_i} + \Pi_{T_i}\boldsymbol{\beta}_{T_i} + \boldsymbol{\varepsilon}_{T_i} \\ &= \boldsymbol{\mu}_{T_i} + \underbrace{T_j\beta_{T_j} + \ldots + T_k\beta_{T_k}}_{\text{traits}} + \underbrace{G_l\beta_{G_l} + \ldots + G_m\beta_{G_m}}_{\text{markers}} + \boldsymbol{\varepsilon}_{T_i} \end{aligned}$$

and the local distribution of each marker G_i is likewise

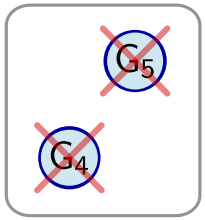
$$\begin{aligned} G_i &= \boldsymbol{\mu}_{G_i} + \Pi_{G_i}\boldsymbol{\beta}_{G_i} + \boldsymbol{\varepsilon}_{G_i} = \\ &= \boldsymbol{\mu}_{G_i} + \underbrace{G_l\beta_{G_l} + \ldots + G_m\beta_{G_m}}_{\text{markers}} + \boldsymbol{\varepsilon}_{G_i} \end{aligned}$$

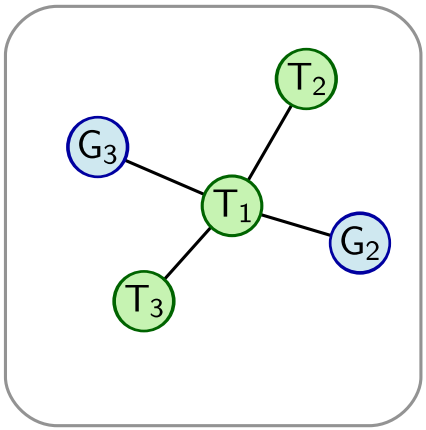
in which the regressors (Π_{T_i} or Π_{G_i}) are treated as fixed effects. Π_{T_i} can be interpreted as causal effects for the traits, Π_{G_i} as markers being in linkage disequilibrium with each other.

Learning the Bayesian Network (I)

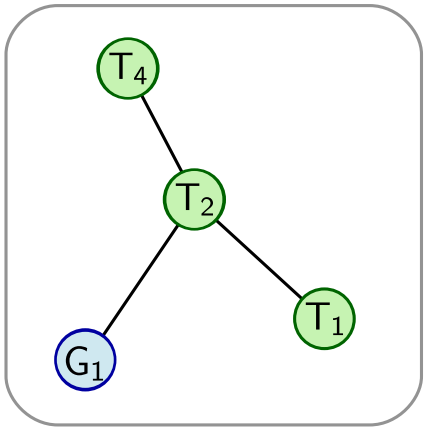
1. Feature Selection.
   - 1.1 Independently learn the parents and the children of each trait with the SI-HITON-PC algorithm; children can only be other traits, parents are mostly markers, spouses can be either. Both are selected using the exact Student's t test for partial correlations.
   - 1.2 Drop all the markers that are not parents of any trait.

Redundant markers that are not in the Markov blanket of any trait

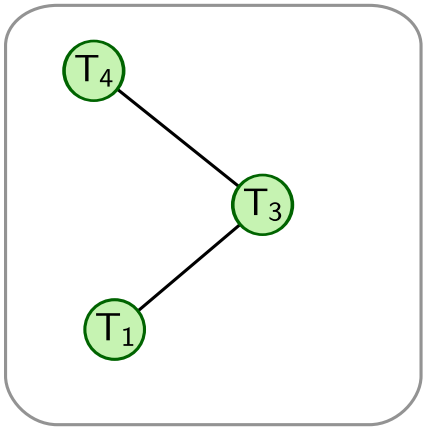




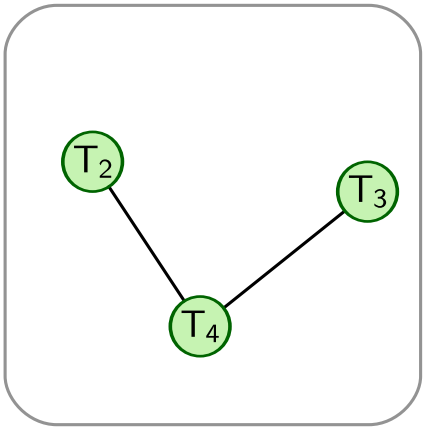
Parents and children of T_1



Parents and children of T_2



Parents and children of T_3



Parents and children of T_4

The Semi-Interleaved HITON-PC Algorithm

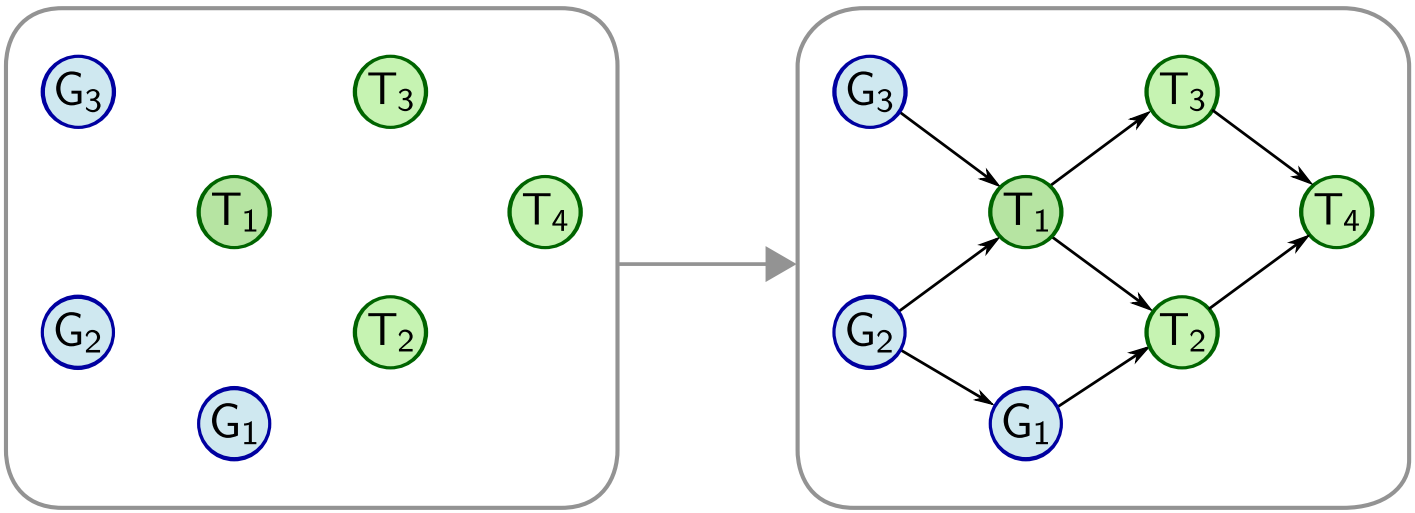
Input: each trait T_i in turn, other traits (T_j) and all markers (G_l), a significance threshold α.
Output: the set $\mathbf{CPC}$ parents and children of T_i in the BN.

1. Perform a marginal independence test between T_i and each T_j ($T_i \perp\!\!\!\perp T_j$) and G_l ($T_i \perp\!\!\!\perp G_l$) in turn.
2. Discard all T_j and G_l whose p-values are greater than α.
3. Set $\mathbf{CPC} = \{\varnothing\}$.
4. For each the T_j and G_l in order of increasing p-value:
   - 4.1 Perform a conditional independence test between T_i and T_j/G_l conditional on all possible subsets $\mathbf{Z}$ of the current $\mathbf{CPC}$ ($T_i \perp\!\!\!\perp T_j \mid \mathbf{Z} \subseteq \mathbf{CPC}$ or $T_i \perp\!\!\!\perp G_l \mid \mathbf{Z} \subseteq \mathbf{CPC}$).
   - 4.2 If the p-value is smaller than α for all subsets then $\mathbf{CPC} = \mathbf{CPC} \cup \{T_j\}$ or $\mathbf{CPC} = \mathbf{CPC} \cup \{G_l\}$.

NOTE: the algorithm is defined for a generic independence test, you can plug in any test that is appropriate for the data.

Learning the Bayesian Network (II)

2. Structure Learning. Learn the structure of the network from the nodes selected in the previous step, setting the directions of the arcs according to the assumptions above. The optimal structure can be identified with a suitable goodness-of-fit criterion such as BIC. This follows the spirit of other hybrid approaches (combining constraint-based and score-based learning) that have shown to be well-performing in the literature.

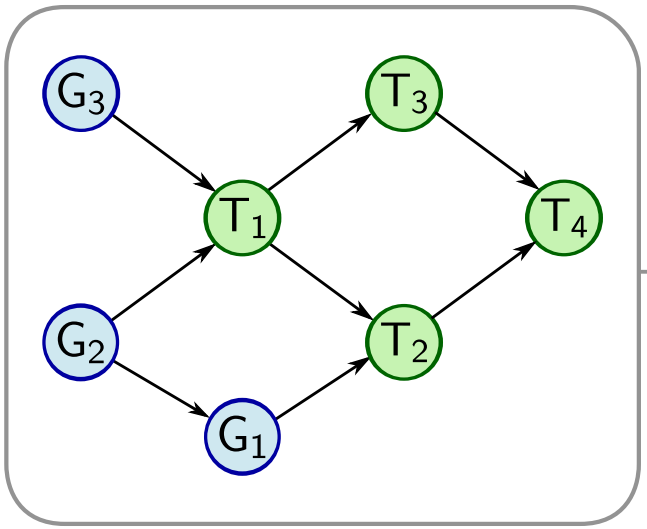


Empty network

Learned network

Learning the Bayesian Network (III)

3. Parameter Learning. Learn the parameters: each local distribution is a linear regression and the global distribution is a hierarchical linear model. Typically least squares works well because SI-HITON-PC selects sets of weakly correlated parents; ridge regression can be used otherwise.



Learned network

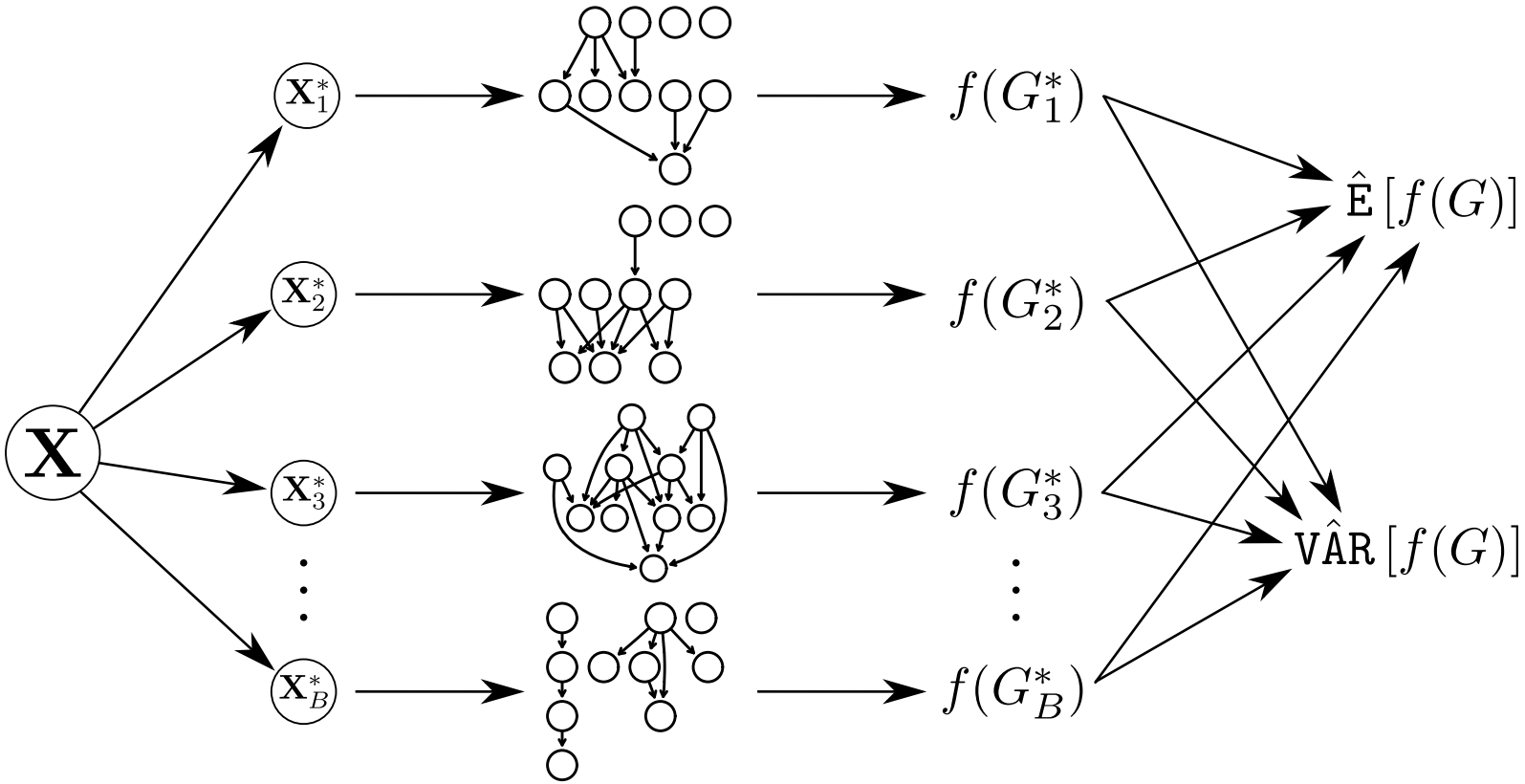
$$
\begin{aligned}
G_1 &= \mu_{G_1} + G_2\beta_{G_2} + \varepsilon_{G_1} \\
G_2 &= \mu_{G_2} + \varepsilon_{G_2} \\
G_3 &= \mu_{G_3} + \varepsilon_{G_3} \\
T_1 &= \mu_{T_1} + G_2\beta_{G_2} + G_3\beta_{G_3} + \varepsilon_{T_1} \\
T_2 &= \mu_{T_2} + T_1\beta_{T_1} + G_1\beta_{G_1} + \varepsilon_{T_2} \\
T_3 &= \mu_{T_3} + T_1\beta_{T_1} + \varepsilon_{T_3} \\
T_4 &= \mu_{T_4} + T_2\beta_{T_2} + T_3\beta_{T_3} + \varepsilon_{T_4}
\end{aligned}
$$

Local distributions

Learning The Structure

```
fit.the.model = function(data, traits, genes, alpha) {
  qtls = vector(length(traits), mode = "list")
  names(qtls) = traits
  # find the parents of each trait among the genes.
  for (q in seq_along(qtls)) {
    # BLUP away the family structure.
    m = lmer(as.formula(paste(traits[q], "~ (1|FUNNEL:PLANT)")), data = data)
    data[!is.na(data[, traits[q]]), traits[q]] = data[, traits[q]] -
      ranef(m)[[1]][paste(data$FUNNEL, data$PLANT, sep = ":"), 1]
    # find out the parents.
    qtls[[q]] = learn.nbr(data[, c(traits, genes)], node = traits[q],
                  method = "si.hiton.pc", test = "cor", alpha = alpha)
  }#FOR
  # yield has no children, and genes cannot depend on traits.
  nodes = unique(c(traits, unlist(qtls)))
  blacklist = tiers2blacklist(list(nodes[nodes %in% genes],
                c("FT", "HT"),
                traits[!(traits %in% c("YLD", "FT", "HT"))], "YLD"))
  # build the overall network.
  hc(data[, nodes], blacklist = blacklist)
}#FIT.THE.MODEL
```

Model Averaging and Assessing Predictive Accuracy



We perform all the above in 10 runs of 10-fold cross-validation to

- assess predictive accuracy with e.g. predictive correlation;
- obtain a set of DAGs to produce an averaged, de-noised consensus DAG with model averaging.

Performing Cross-Validation (Single Fold)

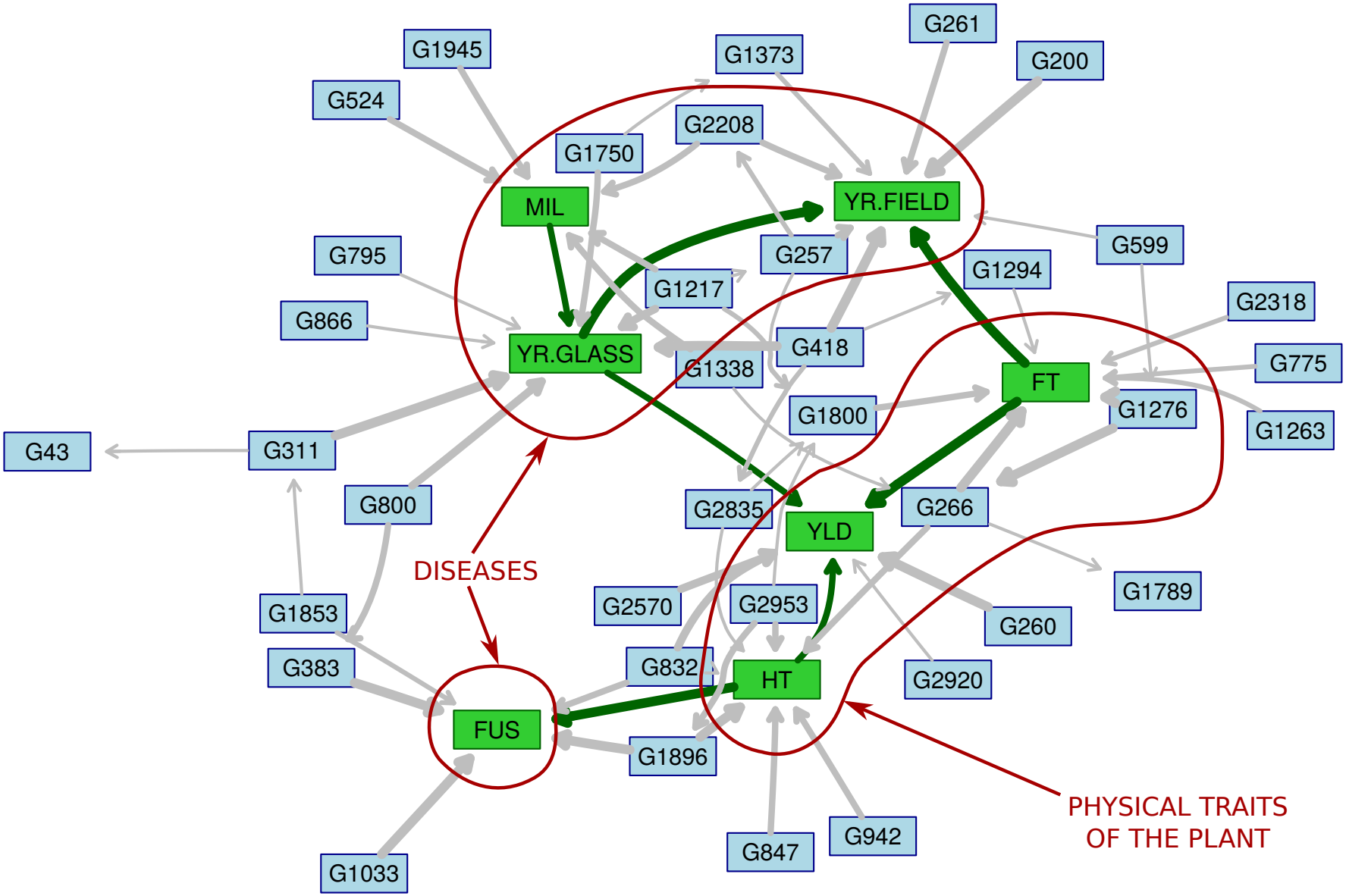
```
predicted = parLapply(kcv, cl = cluster, function(test) {
  # create matrices to store the predicted values.
  pred = matrix(0, nrow = length(test), ncol = length(traits))
  post = matrix(0, nrow = length(test), ncol = length(traits))
  colnames(pred) = colnames(post) = traits
  # split training and test.
  dtraining = data[-test, ]
  dtest = data[test, ]
  # fit the model on the training data.
  model = fit.the.model(dtraining, traits, genes, alpha = alpha)
  fitted = bn.fit(model, dtraining[, nodes(model)])
  # subset the test data.
  dtest = dtest[, nodes(model)]
  # predict each trait in turn, given all the parents.
  for (t in traits)
    pred[, t] = predict(fitted, node = t, data = dtest[, nodes(model)])
  # predict each trait in turn, given all the genes.
  for (t in traits)
    post[, t] = predict(fitted, node = t,
                        data = dtest[, names(dtest) %in% genes, drop = FALSE],
                        method = "bayes-lw", n = 1000)
  return(list(model = fitted, pred = pred, post = post))
})
```

Averaging the Models from Cross-Validation

```
average.the.model = function(batch, data) {
  # gather all the arc lists.
  arclist = list()
  for (i in seq_along(batch)) {
    # extract the models.
    run = batch[[i]]$models
    for (j in seq_along(run))
      arclist[[length(arclist) + 1]] = arcs(run[[j]])
  }#FOR
  # compute the arc strengths.
  nodes = unique(unlist(arclist))
  str = custom.strength(arclist, nodes = nodes)
  # estimate the threshold and average the networks.
  averaged = averaged.network(str)
  # subset the network to remove isolated nodes.
  relnodes = nodes(averaged)[sapply(nodes, degree, object = averaged) > 0]
  averaged2 = subgraph(averaged, relnodes)
  str2 = str[(str$from %in% relnodes) & (str$to %in% relnodes), ]
  # save the fitted averaged network.
  fitted = bn.fit(averaged2, data[, nodes(averaged2)])

  return(list(model = averaged2, strength = str2, fitted = fitted))
}#AVERAGE.THE.MODEL
```

UNIVERSITY OF OXFORD

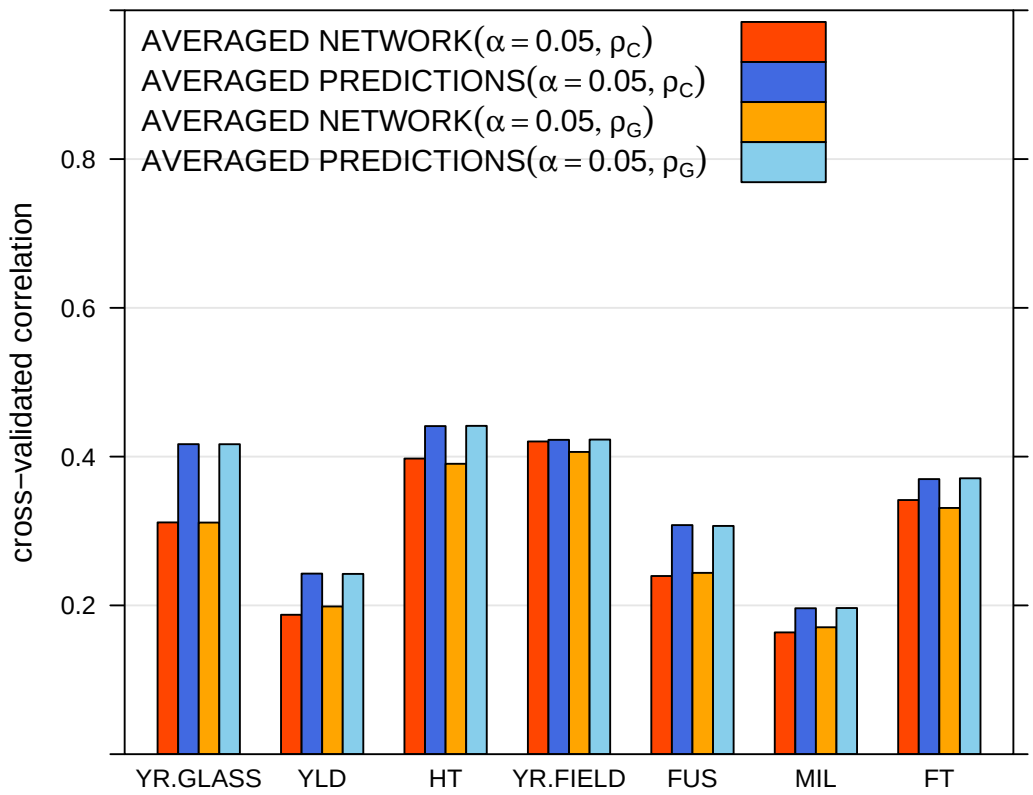
The Averaged Bayesian Network (44 nodes, 66 arcs)



Predicting Traits for New Individuals

We can predict the traits:

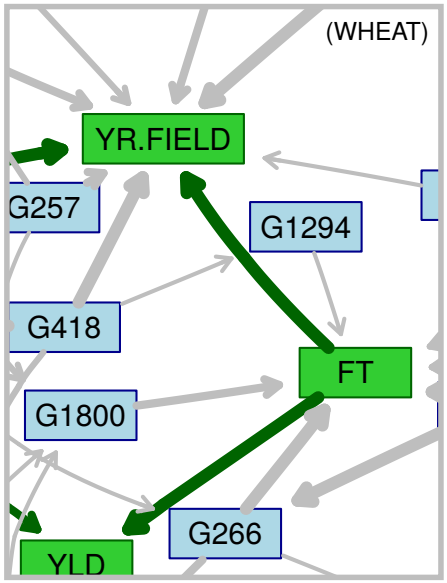
1. from the averaged consensus network;
2. from each of the 10×10 networks we learn during cross-validation, and average the predictions for each new individual and trait.



Option 2. almost always provides better accuracy than option 1.; 10×10 networks capture more information, and we have to learn them anyway. So: averaged network for interpretation, ensemble of networks for predictions.

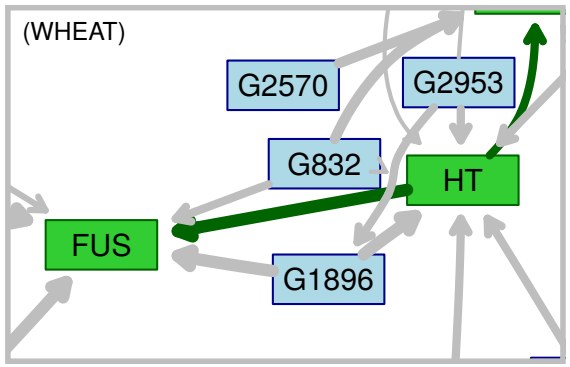
Causal Relationships Between Traits

One of the key properties of BNs is their ability to capture the direction of the causal relationships in the absence of latent confounders (the experimental design behind the data collection should take care of a number of them). Markers are causal for traits, but we do not know how traits influence each other, and we want to learn that from the data.

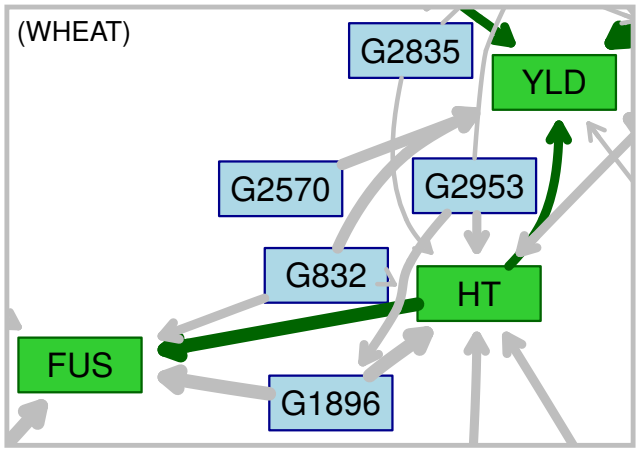


It works out because each trait will have at least one incoming arc from the markers, say $G_l \to T_j$, and then $(G_l \to) T_j \leftarrow T_k$ and $(G_l \to) T_j \to T_k$ are not probabilistically equivalent. So the network can

- suggest the direction of novel relationships;
- confirm the direction of known relationships, troubleshooting the experimental design and data collection.



Spotting Confounding Effects



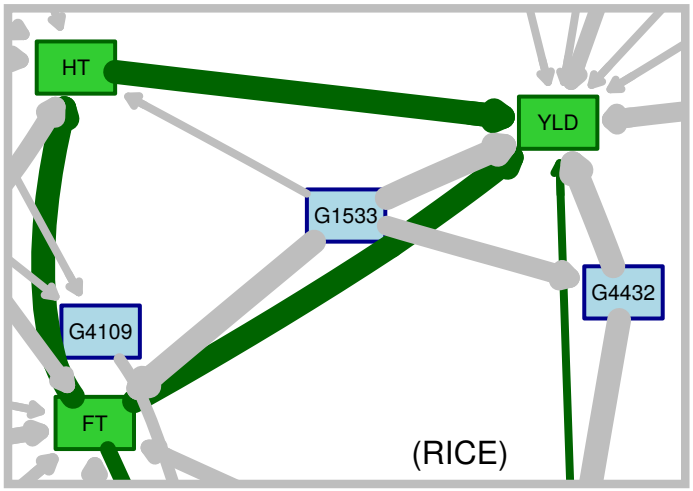
Traits can interact in complex ways that may not be obvious when they are studied individually, but that can be explained by considering neighbouring variables in the network.

An example: in the WHEAT data, the difference in the mean YLD between the bottom and top quartiles of the FUS disease scores is +0.08.

So apparently FUS is associated with increased YLD! What we are actually measuring is the confounding effect of HT (FUS $\leftarrow$ HT $\rightarrow$ YLD); conditional on each quartile of HT, FUS has a negative effect on YLD ranging from −0.04 to −0.06. This is reassuring since it is known that susceptibility to fusarium is positively related to HT, which in turn affects YLD.

Disentangling Pleiotropic Effects (I)

When a marker is shown to be associated to multiple traits in a GWAS, we should separate its direct and indirect effects on each of the traits. (Especially if the traits themselves are linked!)
Take for example G1533 in the RICE data set: it is putative causal for YLD, HT and FT.



- The difference in mean between the two homozygotes is +4.5cm in HT, +2.28 weeks in FT and +0.28 t/ha in YLD.
- Controlling for YLD and FT, the difference for HT halves (+2.1cm);
- Controlling for YLD and HT, the difference for FT is about the same (+2.3 weeks);
- Controlling for HT and FT the difference for YLD halves (+0.16 t/ha).

So, the model suggests the marker is causal for FT and that the effect on the other traits is partly indirect. This agrees from the p-values from an independent GWAS study (FT: 5.87e-28 $<$ YLD: 4.18e-10, HT:1e-11).

Disentangling Pleiotropic Effects (II)

```
control.ht = mutilated(bn.net(fitted), list("YLD" = 0, "FT" = 0))
control.ht = bn.fit(control.ht, indica[, nodes(control.ht)])
sim.aa = cpdist(control.ht, node = c("HT"), evidence = list(G1533 = 0),
        method = "lw")
sim.AA = cpdist(control.ht, node = c("HT"), evidence = list(G1533 = 2),
        method = "lw")
colMeans(sim.AA) - colMeans(sim.aa)

control.ft = mutilated(bn.net(fitted), list("YLD" = 0, "HT" = 0))
control.ft = bn.fit(control.ft, indica[, nodes(control.ft)])
sim.aa = cpdist(control.ft, node = c("FT"), evidence = list(G1533 = 0),
        method = "lw")
sim.AA = cpdist(control.ft, node = c("FT"), evidence = list(G1533 = 2),
        method = "lw")
colMeans(sim.AA) - colMeans(sim.aa)

control.yld = mutilated(bn.net(fitted), list("FT" = 0, "HT" = 0))
control.yld = bn.fit(control.yld, indica[, nodes(control.yld)])
sim.aa = cpdist(control.yld, node = c("YLD"), evidence = list(G1533 = 0),
        method = "lw")
sim.AA = cpdist(control.yld, node = c("YLD"), evidence = list(G1533 = 2),
        method = "lw")
colMeans(sim.AA) - colMeans(sim.aa)
```

Case Study:

INTED

Learning a Bayesian Structure to Model Attitudes Towards Business Creation at University

Ruiz-Ruano García *et al.*, INTED, 5242–5249 (2014).

The main objective of this paper is to test a theoretical model of business creation based on the attitudes perspective:

> *The intention to create a new business would depend on attitudinal evaluation, if someone considers that creating a new business is a positive thing, he or she will be more prone to carry out the target behaviour. Additionally, intentions also depend on normative beliefs. That is to say, intentions depend on the perceived social pressure related with a particular behaviour.*

The data contains the answers to an electronic questionnaire from 1542 university professors from Andalusian universities (unfortunately with a response rate of $\approx 10\%$).

The Questionnaire

The questionnaire contained six sections:

1. demographic data;
2. questions directly related with entrepreneurship phenomena;
3. environment attitudes;
4. obstacles and facilitators;
5. an attitudinal scale;
6. comments and details.

To measure different aspect related with the entrepreneurial attitude we used scales about perceived obstacles, perceived facilitators, self-efficacy, locus of control, attitude towards business creation and normative beliefs. Scores in all scales were individually recoded into three levels of response (low, medium and high) using k-means.

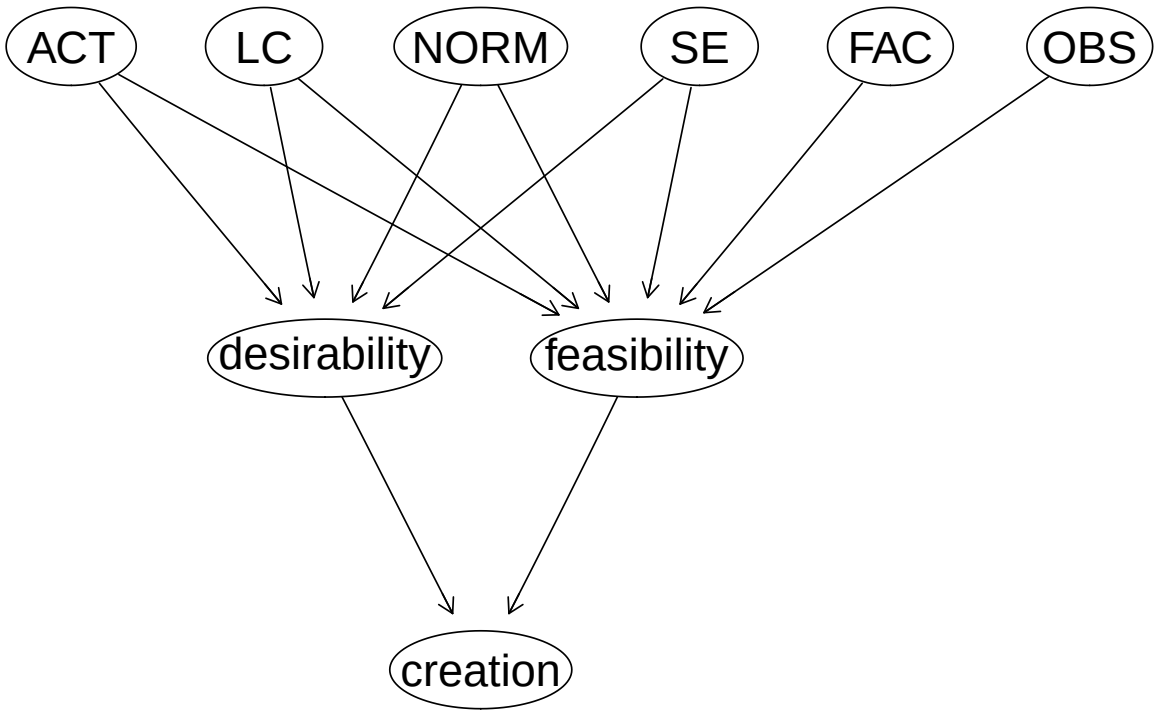
The Derived Scales

- perceived obstacles (OBS, out of 17): "Having to work too many hours", "Lack of experience", "Ignorance of activity sector", etc.
- perceived facilitators (FAC, out of 11): "Have perceived a need in the market", "The detection of a business opportunity" or "The availability of personal assets to invest", etc.
- self-efficacy (SE, 9 Likert items), the perceived difficulty to actually carry out a specific behaviour: "Working under continuous stress, pressure and conflict", "To form alliances or partnerships with other companies", etc.
- locus of control (LC, 3 Likert items): "If you want, you can easily be an entrepreneur and starting your own business", etc.
- attitude towards business creation (ACT, 6 Likert items): "To what extent do you believe that these elements are related with the creation of a new company?", "To what extent do you like assume it?", etc.
- normative beliefs (NORM, 4 Likert items): "Please, think in your family, closest friends and social environment and indicate the degree to which they are favourable to the idea that you create a company", etc.

A Prognostic Model

From the literature we assumed this prognostic BN for the data:

```
progn = model2network(
  paste0("[creation|desirability:feasibility][desirability|LC:SE:ACT:NORM]",
         "[feasibility|LC:SE:ACT:NORM:FAC:OBS][LC][FAC][OBS][SE][ACT][NORM]"))
graphviz.plot(progn, shape = "ellipse")
```



Running Out of Samples

The problems start when we try to learn the parameters of the BN from the data:

```
summary(inted)
```

```
##  creation   desirability                feasibility       LC
##  Yes: 480   Yes:882      Very.little.feasible:378   High  :373
##  No :1062   No :660      A.little.feasible   :672   Low   :544
##                          Feasible            :444   Medium:625
##                          A.lot.feasible      : 48
##      FAC           OBS           SE          ACT          NORM
##  Low   :561   Low   :312   Medium:412   Medium:724   High  :318
##  High  :259   Medium:793   Low   :774   Low   :226   Medium:452
##  Medium:722   High  :437   High  :356   High  :592   Low   :772
##
```

A cursory examination suggests that the sample size is too small.

```
nparams(progn, inted)
```

```
## [1] 2288
```

```
nrow(inted)
```

```
## [1] 1542
```

Small n, Large p

If we learn the parameters with the classic maximum likelihood estimator, $\approx 40\%$ of the CPT is missing values and another $\approx 40\%$ is 0-1 distributions, which clearly is not ideal.

```
fitted.progn = bn.fit(progn, inted)
ldist = coef(fitted.progn$feasibility)
length(which(is.na(ldist))) / length(ldist)
```

```
## [1] 0.396
```

```
length(which(ldist %in% c(0, 1))) / length(ldist)
```

```
## [1] 0.397
```

While we can paper over the problem by using posterior estimates...

```
fitted.progn = bn.fit(progn, inted, method = "bayes", iss = 1)
ldist = coef(fitted.progn$feasibility)
length(which(is.na(ldist))) / length(ldist)
```

```
## [1] 0
```

```
length(which(ldist %in% c(0, 1))) / length(ldist)
```

```
## [1] 0
```

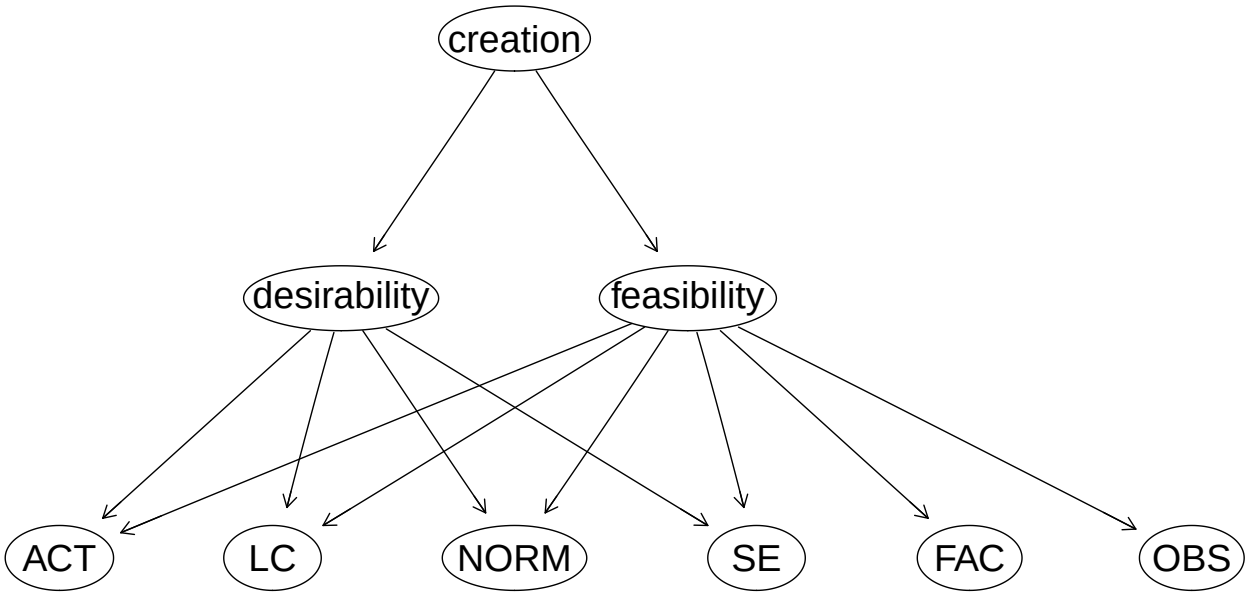
... the BN would still lack statistical power.

A Diagnostic Model

```
diagn = model2network(
  paste("[creation][desirability|creation][feasibility|creation]",
        "[LC|desirability:feasibility][FAC|feasibility][OBS|feasibility]",
        "[SE|desirability:feasibility][ACT|desirability:feasibility]",
        "[NORM|desirability:feasibility]", sep = ""))
nparams(diagn, inted)
```

```
## [1] 89
```

```
graphviz.plot(diagn, shape = "ellipse")
```



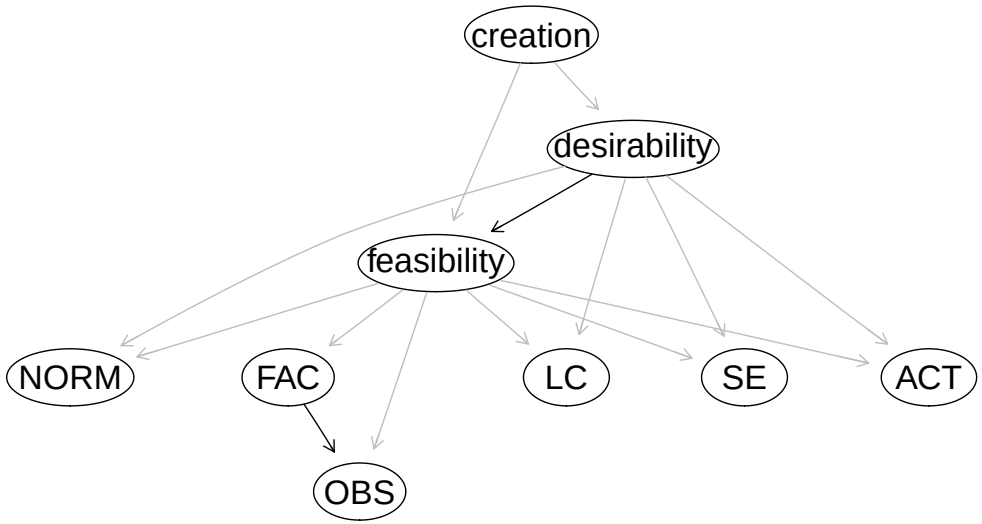
Developing the Model

The diagnostic BN has far fewer parameters, and we can estimate them with reasonable accuracy from the data.

```
fitted.diagn = bn.fit(diagn, inted)
```

Do the data support the any further arcs we may have overlooked?

```
diagn2 = tabu(inted, whitelist = arcs(diagn))
graphviz.plot(diagn2, highlight = list(arcs = arcs(diagn), col = "grey"),
  shape = "ellipse")
```



Job Creation, Goodness of Fit

The three models we are considering fit the data equally well; the classification error for `creation` is about the same (≈ 0.274).

```
pred = predict(fitted.diagn, node = "creation", data = inted,
          method = "bayes-lw")
ct = table(inted$creation, pred)
1 - sum(diag(ct)) / sum(ct)
```

```
## [1] 0.274
```

```
pred = predict(bn.fit(diagn2, inted), node = "creation", data = inted,
          method = "bayes-lw")
ct = table(inted$creation, pred)
1 - sum(diag(ct)) / sum(ct)
```

```
## [1] 0.275
```

```
pred = predict(fitted.progn, node = "creation", data = inted,
          method = "bayes-lw")
ct = table(inted$creation, pred)
1 - sum(diag(ct)) / sum(ct)
```

```
## [1] 0.273
```

Cross-Validation and Predictive Accuracy

Predictive accuracy is also similar; and note how we do not reuse `diagn2` here but we re-estimate it to avoid using the data twice.

```
xval.diagn = bn.cv(inted, diagn, loss = "pred-lw", runs = 10,
  loss.args = list(target = "creation"),
  fit = "bayes", fit.args = list(iss = 1))
mean(sapply(xval.diagn, attr, "mean"))
```

```
## [1] 0.274
```

```
xval.diagn2 = bn.cv(inted, "tabu", loss = "pred-lw", runs = 10,
  loss.args = list(target = "creation"),
  algorithm.args = list(whitelist = arcs(diagn)),
  fit = "bayes", fit.args = list(iss = 1))
mean(sapply(xval.diagn2, attr, "mean"))
```

```
## [1] 0.276
```

```
xval.progn = bn.cv(inted, progn, loss = "pred-lw", runs = 10,
  loss.args = list(target = "creation"),
  fit = "bayes", fit.args = list(iss = 1))
mean(sapply(xval.progn, attr, "mean"))
```

```
## [1] 0.278
```

Scales and Predictive Accuracy

Interestingly, the summary variables `desirability` and `feasibility` (which d-separate `creation` from the six scales) improve the predictive accuracy.

```
from = c("ACT", "LC", "NORM", "SE", "FAC", "OBS")
xval.diagn = bn.cv(inted, diagn, loss = "pred-lw", runs = 10,
  loss.args = list(target = "creation", from = from),
  fit = "bayes", fit.args = list(iss = 1))
mean(sapply(xval.diagn, attr, "mean"))
```

```
## [1] 0.307
```

```
xval.diagn2 = bn.cv(inted, "tabu", loss = "pred-lw", runs = 10,
  loss.args = list(target = "creation", from = from),
  algorithm.args = list(whitelist = arcs(diagn)),
  fit = "bayes", fit.args = list(iss = 1))
mean(sapply(xval.diagn2, attr, "mean"))
```

```
## [1] 0.309
```

```
xval.progn = bn.cv(inted, progn, loss = "pred-lw", runs = 10,
  loss.args = list(target = "creation", from = from),
  fit = "bayes", fit.args = list(iss = 1))
mean(sapply(xval.progn, attr, "mean"))
```

```
## [1] 0.31
```

Learning and Interpretability

The BN proposed by `tabu()` as an extension of the diagnostic BN produces, at least, an interesting statistical model from the theoretical point of view. There are two new arcs associating two nodes and this shed light to previously unexplored hypotheses.

- The arc `desirability` $\rightarrow$ `feasibility` makes sense because you will perceive more desirable to create a new business if it is considerate feasible.
- The arc `FAC` $\rightarrow$ `OBS` also makes sense because if you perceive few obstacles, you would perceive more facilitators to do a new venture.

This second arc is particularly interesting form a practical point of view in the context of entrepreneurship promotion. For example, it would be advisable to introduce laws or public-private incentives in order to reduce the subjective perception of difficulties in potential entrepreneurs.

Queries

Indeed increasing `feasibility` dramatically improves the attitude towards business creation.

```
fitted.diagn2 = bn.fit(diagn2, inted)
cpquery(fitted.diagn2, (creation == "Yes"),
  evidence = list(feasibility = "A.lot.feasible"), method = "lw")
```

```
## [1] 0.798
```

```
cpquery(fitted.diagn2, (creation == "Yes"),
  evidence = list(feasibility = "Very.little.feasible"), method = "lw")
```

```
## [1] 0.137
```

The same is true for decreasing `OBS`, but not as much; the reason is that `OBS` is farther away from `creation` so the effect of the conditioning is smaller.

```
cpquery(fitted.diagn2, (creation == "Yes"), evidence = list(OBS = "High"),
  method = "lw")
```

```
## [1] 0.351
```

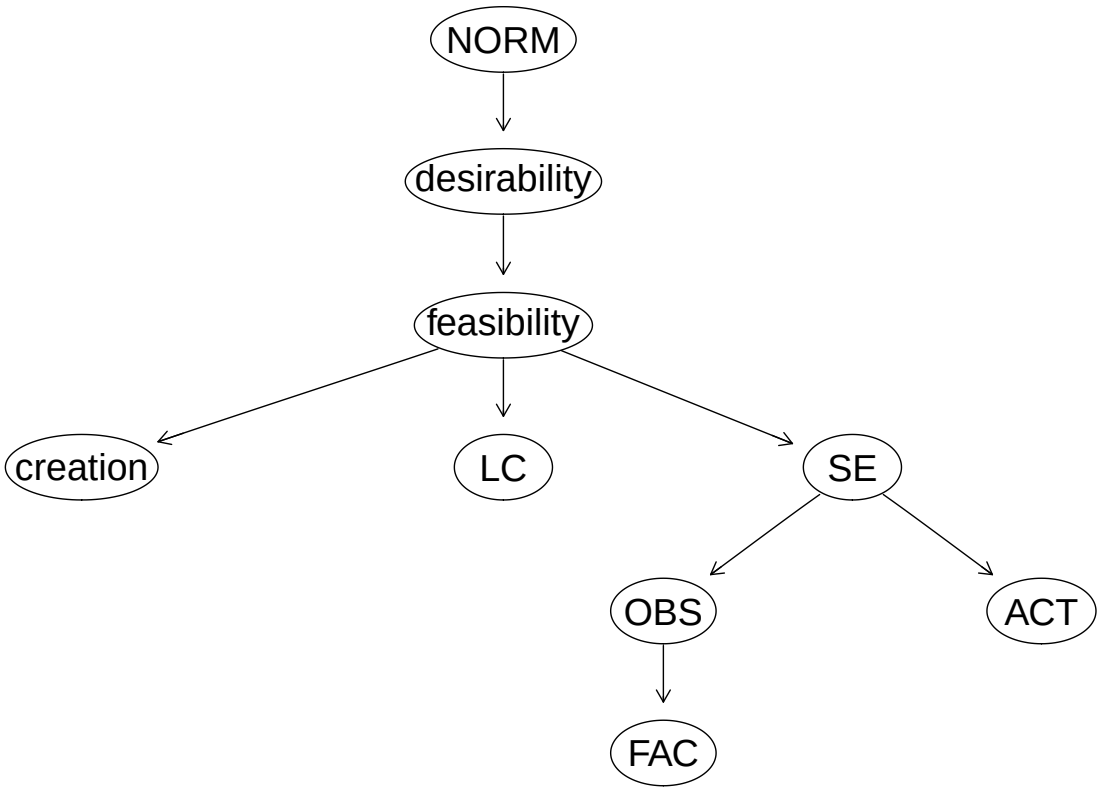
```
cpquery(fitted.diagn2, (creation == "Yes"), evidence = list(OBS = "Low"),
  method = "lw")
```

```
## [1] 0.276
```

The DAG from Structure Learning is not Interpretable

On the other hand, we can learn a DAG directly from the data, but the result has no clear interpretation because the arcs do not map well to what we know from the literature.

```
graphviz.plot(tabu(inted), shape = "ellipse")
```



That's It, Thanks!