A simple example for package “bnlearn”

Here bnlearn will be used to analyze a small data set, learning.test. It is included in the package itself along with other real word and synthetic data sets, and is used in the example sections throughout the manual pages due to its simple structure.

1. Loading the package

Bnlearn are available from CRAN.

> library(bnlearn)

2. Learning a Bayesian network from data

Once bnlearn is loaded, learning.test itself can be loaded into a data frame of the same name with a call to data.

> data(learning.test)

> str(learning.test)

```
> str(learning.test)
'data.frame': 5000 obs. of  6 variables:
 $ A: Factor w/ 3 levels "a","b","c": 2 2 1 1 1 3 3 2 2 2 ...
 $ B: Factor w/ 3 levels "a","b","c": 3 1 1 1 1 3 3 2 2 1 ...
 $ C: Factor w/ 3 levels "a","b","c": 2 3 1 1 2 1 2 1 2 2 ...
 $ D: Factor w/ 3 levels "a","b","c": 1 1 1 1 3 3 3 2 1 1 ...
 $ E: Factor w/ 3 levels "a","b","c": 2 2 1 2 1 3 3 2 3 1 ...
 $ F: Factor w/ 2 levels "a","b": 2 2 1 2 1 1 1 2 1 1 ...
```

learning.test contains six discrete variables, stored as factors, each with 2 (for F) or 3 (for A, B, C, D and E) levels. The structure of the Bayesian network associated with this data set can be learned for example with the Grow-Shrink algorithm, implemented in the gs function, and stored in an object of class bn.

```
> bn.gs <- gs(learning.test)
> bn.gs

  Bayesian network learned via Constraint-based methods

  model:
    [partially directed graph]
  nodes:                                   6
  arcs:                                    5
    undirected arcs:                       1
    directed arcs:                         4
  average markov blanket size:             2.33
  average neighbourhood size:              1.67
  average branching factor:                0.67

  learning algorithm:
                                      Grow-Shrink
  conditional independence test:
                        Mutual Information (disc.)
  alpha threshold:                         0.05
  tests used in the learning procedure:    60
  optimized:                               TRUE
```

Other constraint-based algorithms return the same partially directed network structure (again as an object of class bn).

> bn2 <- iamb(learning.test)

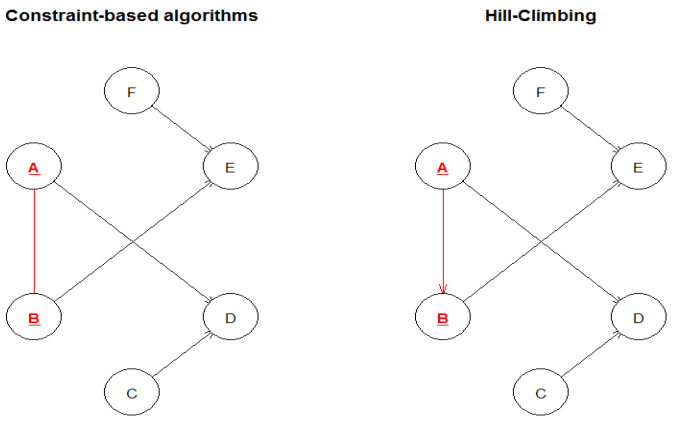
> bn3 <- fast.iamb(learning.test)

> bn4 <- inter.iamb(learning.test)

```
> bn.hc <- hc(learning.test, score = "aic")
> bn.hc

  Bayesian network learned via Score-based methods

  model:
   [A][C][F][B|A][D|A:C][E|B:F]
  nodes:                                   6
  arcs:                                    5
    undirected arcs:                       0
    directed arcs:                         5
  average markov blanket size:             2.33
  average neighbourhood size:              1.67
  average branching factor:                0.83

  learning algorithm:
                                    Hill-Climbing
  score:
                                      AIC (disc.)
  penalization coefficient:              1
  tests used in the learning procedure:  40
  optimized:                             TRUE
```

A way to compare the two network structures is to plot them side by side and highlight the differing arcs. This can be done either with the plot function.

```
> par(mfrow = c(1,2))
> plot(bn.gs, main = "Constraint-based algorithms", highlight = c("A", "B"))
> plot(bn.hc, main = "Hill-Climbing", highlight = c("A", "B"))
```



Arcs which differ between the two network structures are plotted in red.

The network structure learned by gs, iamb, fast.iamb and inter.iamb is equivalent to the one lea rned by hc; changing the arc A − B to either A → B or to B → A results in networks with the sam e score because of the score equivalence property (which holds for all the implemented score f unctions with the exception of K2). Therefore if there is any prior information about the relatio nship between A and B the appropriate direction can be whitelisted (or its reverse can blacklist ed, which is equivalent in this case).

3. Network analysis and manipulation

The structure of a Bayesian network is uniquely specified if its graph is completely directed. In this case it can be represented as a string with the modelstring function.

```
> modelstring(bn.hc)
[1] "[A][C][F][B|A][D|A:C][E|B:F]"
```

whose output is also included in the print method for the objects of class bn. Each node is printed in square brackets along with all its parents (which are reported after a pipe as a colon-separated list), and its position in the string depends on the partial ordering defined by the network structure.

Partially directed graphs can be transformed into completely directed ones with the set.arc, drop.arc and reverse.arc functions. For example the direction of the arc A − B in the bn.gs object can be set to A → B, so that the resulting network structure is identical to the one learned by the hill-climbing algorithm.

```
> modelstring(bn.hc)
[1] "[A][C][F][B|A][D|A:C][E|B:F]"
> undirected.arcs(bn.gs)
     from to
[1,] "A"  "B"
[2,] "B"  "A"
> bn.dag<-set.arc(bn.gs,"A","B")
> modelstring(bn.dag)
[1] "[A][C][F][B|A][D|A:C][E|B:F]"
```

Acyclicity is always preserved, as these commands return an error if the requested changes would result in a cyclic graph.

Further information on the network structure can be extracted from any bn object with the following functions:

• whether the network structure is acyclic (acyclic) or completely directed (directed);

• the labels of the nodes (nodes), of the root nodes (root.nodes) and of the leaf nodes (leaf.nodes);

• the directed arcs (directed.arcs) of the network, the undirected ones (undirected.arcs) or both of them (arcs);

• the adjacency matrix (amat) and the number of parameters (nparams) associated with the network structure;

• the parents (parents), children (children), Markov blanket (mb), and neighbourhood (nbr) of each node.

The arcs, amat and modelstring functions can also be used in combination with empty.graph to create a bn object with a specific structure from scratch:

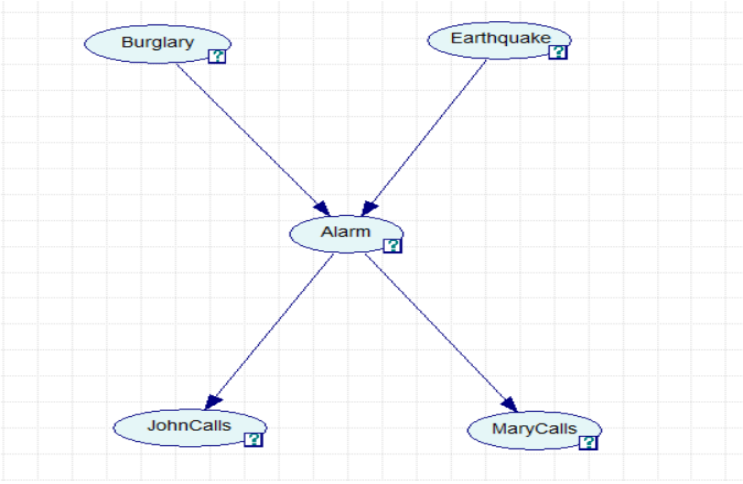
```
> other <- empty.graph(nodes = nodes(bn.hc))
> arcs(other) <- data.frame(from = c("A", "A", "B", "D"),to = c("E", "F", "C", "E"))
> other

  Random/Generated Bayesian network

  model:
   [A][B][D][C|B][E|A:D][F|A]
  nodes:                                6
  arcs:                                 4
    undirected arcs:                    0
    directed arcs:                      4
  average markov blanket size:          1.67
  average neighbourhood size:           1.33
  average branching factor:             0.67

  generation algorithm:                 Empty
```

This is particularly useful to compare different network structures for the same data, for example to verify the goodness of fit of the learned network with respect to a particular score function.

```
> score(other, data = learning.test, type = "aic")
[1] -28019.79
> score(bn.hc, data = learning.test, type = "aic")
[1] -23873.13
```

Another example:

Using GeNIe to build a Bayesian network, and then generate a data set with 10,000 cases.

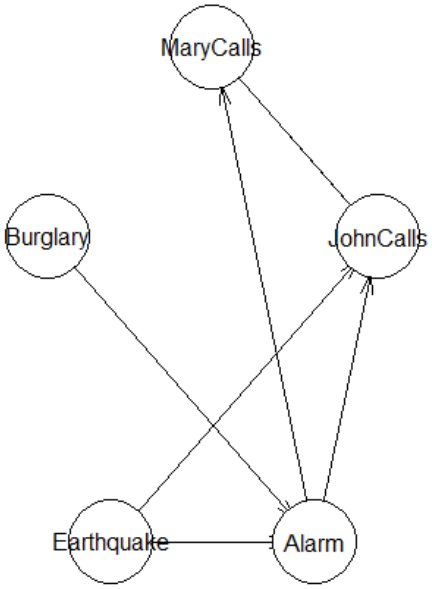


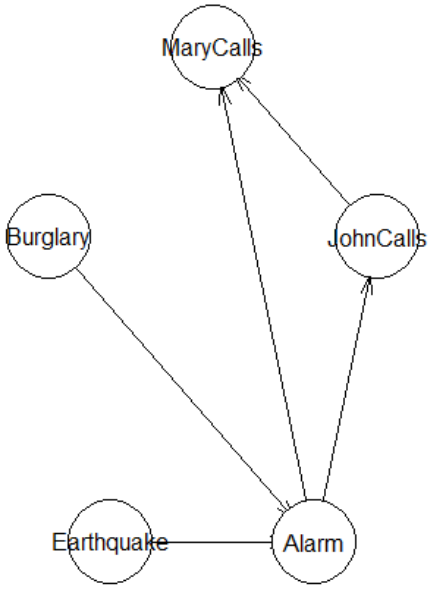
R code:

```
data<-read.table("Alarm system.txt",head=TRUE)

data<-as.matrix(data)

data[data]<-1

data<-as.data.frame(data)

bn.gs <- gs(data)

bn2 <- iamb(data)
```

```
bn3 <- fast.iamb(data)

bn4 <- inter.iamb(data)

compare(bn.gs, bn2)

compare(bn.gs, bn3)

compare(bn.gs,bn4)

bn.hc <- hc(data)

par(mfrow = c(1,2))

plot(bn.gs)

plot(bn.hc)

modelstring(bn.hc)

undirected.arcs(bn.gs)

other <- empty.graph(nodes = nodes(bn.hc))

arcs(other) <- data.frame(from = c("Burglary", "Earthquake", "Alarm", "Alarm"),to = c("Alarm", "Alarm", "JohnCalls", "MaryCalls"))

other

score(other, data = data)

score(bn.hc, data = data)

plot(other)
```

These plots are the results from bnlearn. The right side is almost the same as original structure.





This is what the structure should be look like.