Chapter 3
Bayesian Networks

3.1 Introduction

A Bayesian network is traditionally understood to be a graphical model based on a directed acyclic graph (a DAG). The term refers to its use for Bayesian inference in expert systems, where appropriate use of conditional independencies enable rapid and efficient computation of updated probabilities for states of unobserved variables, a calculation which in principle is forbiddingly complex. The term is also used in contrast to the once fashionable neural networks which used quite a different inference mechanism. In principle there is nothing Bayesian about a Bayesian network.

It should be pointed out that the DAG is only used to give a simple and transparent way of specifying a probability model, whereas the simplification in the computations are based on exploiting conditional independencies in an undirected graph. Thus, as we shall illustrate, methods for building undirected graphical models can just as easily be used for building probabilistic inference machines.

The **gRain** package (`gRaphical independence network`) is an `R` implementation of such networks. The package implements the propagation algorithm described in Lauritzen and Spiegelhalter (1988). Most of the exposition here is based on the package **gRain**, but **RHugin** is also used, see below. The networks in **gRain** are restricted to discrete variables, each with a finite state space. The package has a similar functionality to that of the `GRAPPA` suite of functions (Green 2005).

The package **RHugin** provides an `R`-interface to the (commercial) `HUGIN` software, enabling access to the full functionality of `HUGIN` through `R`. **RHugin** is not on CRAN but is available from http://rhugin.r-forge.r-project.org/. **RHugin** requires a version of `HUGIN` to be pre-installed. The examples in this chapter which use **RHugin** work with the free version `HUGIN Lite`, which has full functionality but has size limitations on the networks.

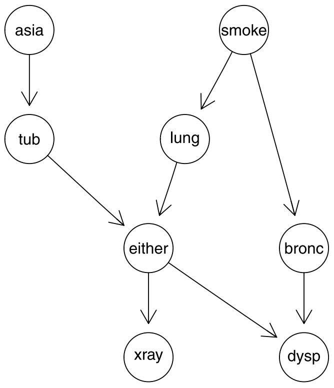


Fig. 3.1 The directed acyclic graph corresponding to the chest clinic example from Lauritzen and Spiegelhalter (1988). The arrows indicate a formalization of the relationships expressed in the narrative

3.1.1 The Chest Clinic Example

This section reviews the chest clinic example of Lauritzen and Spiegelhalter (1988) (illustrated in Fig. 3.1) and shows one way of specifying a network in **gRain**. Details of the steps will be given in later sections. Other ways of specifying a network are described in Sect. 3.3.1. Lauritzen and Spiegelhalter (1988) motivate the example with the following narrative:

> Shortness-of-breath (dyspnoea) may be due to tuberculosis, lung cancer or bronchitis, or none of them, or more than one of them. A recent visit to Asia increases the chances of tuberculosis, while smoking is known to be a risk factor for both lung cancer and bronchitis. The results of a single chest X-ray do not discriminate between lung cancer and tuberculosis, as neither does the presence or absence of dyspnoea.

This narrative is represented by the directed acyclic graph in Fig. 3.1 which forms the basis for the Bayesian network constructed in this example.

3.1.2 Models Based on Directed Acyclic Graphs

We focus on Bayesian networks for discrete variables and we shall, in accordance with Chap. 2, use the following notation: Let $X = X_V = (X_v; v \in V)$ be a discrete random vector. The labels of X_v are generically denoted by i_v so the levels of X are denoted $i = i_V = (i_v, v \in V)$ and the set of possible values of X is denoted $\mathcal{I}$.

The multivariate distribution associated with a Bayesian network is constructed by combining univariate (conditional) distributions using the structure of the directed acyclic graph (DAG) with vertices V. To be precise, probability distributions $p(i_V)$ *factorizes* w.r.t. a directed acyclic graph if it can be expressed as

$$p(i_V) = \prod_{v \in V} p(i_v \,|\, i_{\mathrm{pa}(v)}) \tag{3.1}$$

i.e. if the joint density or probability mass function is a product of conditional densities of individual variables given their parents in the DAG, see also Sect. 1.3.

For the chest clinic example, write the variables as $A =$ Asia, $S =$ smoker, $T =$ tuberculosis, $L =$ lung cancer, $B =$ bronchitis, $D =$ dyspnoea, $X =$ X-ray and $E =$ either tuberculosis or lung cancer. Each variable can take the values "yes" and "no". Note that E is a logical variable which is true ("yes") if either T or L are true ("yes") and false ("no") otherwise. The DAG in Fig. 3.1 now corresponds to a factorization of the joint probability function $p(i_V)$, where $V = \{A, S, T, L, B, E, D, X\}$ (apologies for using X with two different meanings here) as

$$p(i_A)p(i_S)p(i_T|i_A)p(i_L|i_S)p(i_B|i_S)p(i_E|i_T, i_L)p(i_D|i_E, i_B)p(i_X|i_E). \qquad (3.2)$$

In **gRain**, each conditional distribution in (3.2) is specified as a table called a conditional probability table or a CPT for short.

Distributions given as in (3.1) automatically satisfy the *global directed Markov property* so that whenever two sets of nodes A and B are d-separated by a set of nodes S, see Sect. 1.3 for this notion, then $A \perp\!\!\!\perp B \mid S$.

The directed acyclic graph in Fig. 3.1 can be specified as:

```
> g<-list(~asia, ~tub | asia, ~smoke, ~lung | smoke, ~bronc | smoke,
+      ~either | lung : tub, ~xray | either, ~dysp | bronc : either)
> chestdag<-dagList(g)
```

We can query conditional independences using the function `d.separates()` constructed in Sect. 1.3:

```
> d.separates("tub", "smoke", c("dysp","xray"), chestdag)

[1] FALSE
```

whereas

```
> d.separates("tub", "lung", "smoke", chestdag)

[1] TRUE
```

3.1.3 Inference

Suppose we are given evidence that a set of variables $E \subset V$ have a specific value i_E^*. For the chest clinic example, evidence could be that a person has recently visited Asia and suffers from dyspnoea, i.e. $i_A =$ yes and $i_D =$ yes.

With this evidence, we may be interested in the conditional distribution $p(i_v \mid X_E = i_E^*)$ (or $p(i_v \mid i_E^*)$ is short) for some of the variables $v \in V \setminus E$ or in $p(i_U \mid i_E^*)$ for a set $U \subset V \setminus E$. In the chest clinic example, interest might be in $p(i_L \mid i_E^*)$, $p(i_T \mid i_E^*)$ and $p(i_B \mid i_E^*)$, or possibly in the joint (conditional) distribution $p(i_L, i_T, i_B \mid i_E^*)$. Interest might also be in calculating the probability of a specific event, e.g. $p(i_E^*) = p(X_E = i_E^*)$.

As noticed above, each conditional distribution in (3.2) is in **gRain** specified as a conditional probability table. A brute force approach to calculating $p(i_U \mid i_E^*)$ is to calculate the joint distribution given by (3.2) by multiplying the conditional probability tables. Finding $p(i_U \mid i_E^*)$ then reduces to first finding the slice defined by $i_E = i_E^*$ of the joint table and then marginalizing over the variables not in U that slice.

As all variables in the chest clinic example are binary, the joint distribution will have $2^8 = 256$ states but for larger networks/more levels of the variables the joint state space becomes prohibitively large. In most practical cases the set U will be much smaller than V (U might consist of only one or two variables while V can be very large). Combined with the observation that the factorization in (3.2) implies conditional independence restrictions, this implies that $p(i_U \mid i_E^*)$ can be found without ever actually calculating the joint distribution. See Sect. 3.2.3 for details.

3.2 Building and Using Bayesian Networks

3.2.1 Specification of Conditional Probability Tables

One simple way of specifying a model for the chest clinic example is as follows. First we specify conditional probability tables with values as given in Lauritzen and Spiegelhalter (1988). This can be done with `array()` or as here with the `cptable()` function, which offers some additional features:

```
> library(gRain)
> yn <- c("yes","no")
> a    <- cptable(~asia, values=c(1,99), levels=yn)
> t.a  <- cptable(~tub+asia, values=c(5,95,1,99), levels=yn)
> s    <- cptable(~smoke, values=c(5,5), levels=yn)
> l.s  <- cptable(~lung+smoke, values=c(1,9,1,99), levels=yn)
> b.s  <- cptable(~bronc+smoke, values=c(6,4,3,7), levels=yn)
> e.lt <- cptable(~either+lung+tub,values=c(1,0,1,0,1,0,0,1),
                  levels=yn)
> x.e  <- cptable(~xray+either, values=c(98,2,5,95), levels=yn)
> d.be <- cptable(~dysp+bronc+either, values=c(9,1,7,3,8,2,1,9),
                  levels=yn)
```

Notice that the "+" operator used above is slightly misleading in the sense, for example, that the operator does not commute (the order of the variables is important). We use the "+" operator merely as a separator of the variables. The following forms are also valid specifications:

```
> cptable(~tub|asia, values=c(5,95,1,99), levels=yn)
> cptable(c("tub","asia"), values=c(5,95,1,99), levels=yn)
```

Notice that since E is a logical variable which is true if either T or L are true and false otherwise, the corresponding CPT can be created with the special function `ortable()` (there is also an corresponding `andtable()` function):

```
> e.lt <- ortable(~either+lung+tub, levels=yn)
```

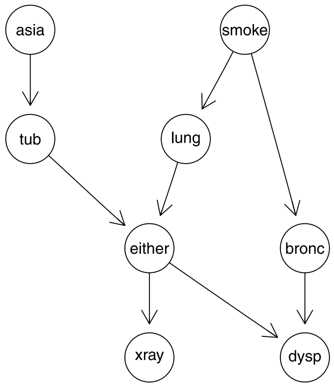
3.2.2 Building the Network

A network is created with the function `grain()` which returns an object of class `grain`:

```
> plist <- compileCPT(list(a, t.a, s, l.s, b.s, e.lt, x.e, d.be))
> grn1  <- grain(plist)
> summary(grn1)
```

```
Independence network: Compiled: FALSE Propagated: FALSE
 Nodes : chr [1:8] "asia" "tub" "smoke" "lung" "bronc" ...
```

```
> plot(grn1)
```



The `compileCPT()` function does some checking of the specified CPT's. (For example, it is checked that the graph defined by the CPT's is acyclic. Furthermore, the specification of `t.a` gives a table with four entries and the variable `tub` is specified to be binary. Hence it is checked that the variable `asia` is also binary.) The object `plist` is a list of arrays and it is from this list that the `grain` object is created.

3.2.2.1 Compilation—Finding the Clique Potentials

A `grain` object must be compiled and propagated before queries can be made. These steps are performed by the `querygrain()` function if necessary, but for some purposes it is advantageous to perform them explicitly. Compilation of a network is done with the `compile()` method for `grain` objects:

```
> grn1c <- compile(grn1)
> summary(grn1c)
```

```
Independence network: Compiled: TRUE Propagated: FALSE
 Nodes : chr [1:8] "asia" "tub" "smoke" "lung" "bronc" ...
 Number of cliques:                 6
 Maximal clique size:               3
 Maximal state space in cliques:    8
```

Compilation of a `grain` object based on CPTs involves the following steps: First it is checked whether the list of CPTs defines a directed acyclic graph (a DAG). If so, then the DAG is created; it is *moralized* and *triangulated* to form a chordal (triangulated) graph. The CPTs are transformed into *clique potentials* defined on the cliques of this chordal graph. The chordal graph together with the corresponding clique potentials are the most essential components of a `grain` object, and one may indeed construct a `grain` object directly from a specification of these two components, see Sect. 3.3.1.

We again consider the Bayesian network of Sect. 3.2.1: The factorization (3.2) into a *clique potential representation* follows by simply noticing that in (3.2) each of the conditional probability tables can be considered a function of the variables it involves. These potentials are simply non-negative functions.

The dependence graph of the Bayesian network is derived from the potentials. For example, the presence of the term $p(x_D \mid x_E, x_B)$ implies that there must be edges between all pairs in $\{D, E, B\}$. Algorithmically, the dependence graph can be formed from the DAG by moralization: The moral graph of a DAG is obtained by first joining all parents of each node by a line and then dropping the directions on the arrows. For the chest clinic example, the edges between `tub` and `lung`, and between `either` and `bronc` are added.

The next step is to triangulate the dependence graph if it is not already so by adding additional edges, so-called fill-ins. This is done to enable simple computation of marginals from the clique potentials, cf. Sect. 3.2.2.2 below. Finding an optimal triangulation (in terms of a minimal number of fill-ins) of a given graph is NP-complete, but various good heuristics exist. The **gRbase** package implements a Minimum Clique Weight Heuristic method inspired by Kjærulff (1990). Two possible fill-ins are the edge between `lung` and `bronc`, and the edge between `either` and `smoke`. The triangulated graph is also a dependence graph for (3.2); the graph just conceals some conditional independence restrictions implied by the model.

The steps described above can alternatively be carried out separately, and Fig. 3.2 illustrates the process:

```
> g   <- grn1$dag
> mg  <- moralize(g)
> tmg <- triangulate(mg)
```

Recall from Sect. 1.2.1 that an ordering $C_1, \dots, C_T$ of the cliques of a graph is a RIP ordering if $S_j = (C_1 \cup \cdots \cup C_{j-1}) \cap C_j$ is contained in one (but possibly several) of the cliques $C_1, \dots, C_{j-1}$, obtained with:

```
> rip(tmg)

cliques
  1 : tub asia
  2 : either tub lung
  3 : bronc lung either
  4 : smoke lung bronc
  5 : dysp bronc either
  6 : xray either
```

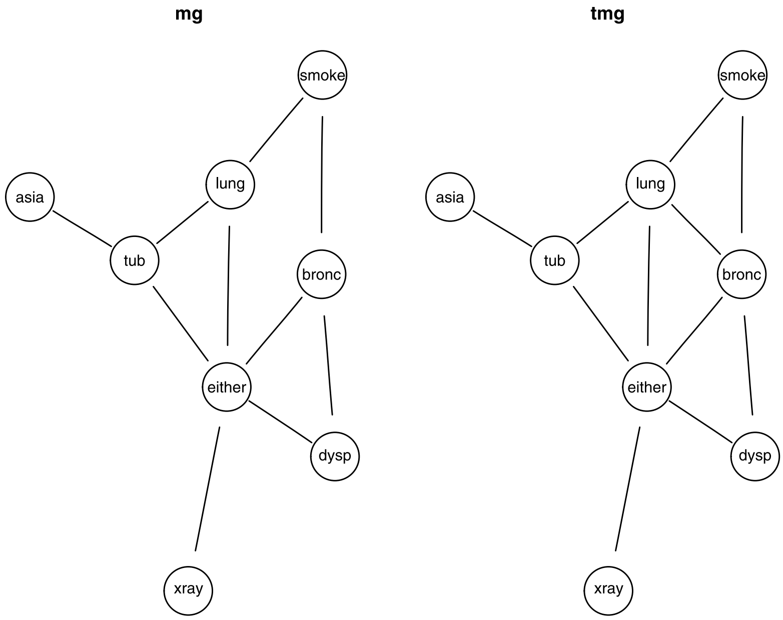


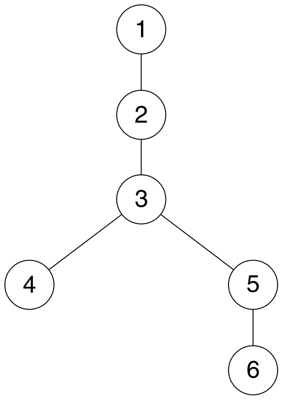
Fig. 3.2 *Left*: moralized DAG; *Right*: triangulated moralized DAG. The chect clinic example of Lauritzen and Spiegelhalter (1988)

```
separators
  1 :
  2 : tub
  3 : lung either
  4 : lung bronc
  5 : bronc either
  6 : either
parents
  1 : 0
  2 : 1
  3 : 2
  4 : 3
  5 : 3
  6 : 5
```

Picking a particular clique, say C_k, with $S_j \subseteq C_k$ and naming this as the *parent clique* of C_j, with C_j being the child of C_k, organizes the cliques of the triangulated graph in a rooted tree with the cliques as nodes and arrows from parent to child. We call S_j the separator and $R_j = C_j \setminus S_j$ the residual, where $S_1 = \emptyset$. The junction tree is formed by ignoring the root and the directions on the edges. It is a tree with the property that for any pair (A, B) of cliques and any clique C on the unique path between A and B it holds that $A \cap B \subseteq C$. It can be shown that *the cliques of a graph can be organized in a junction tree if and only if the graph is triangulated.*

The junction tree can be displayed by `plot()`,

```
> plot(grn1c,type="jt")
```



where the numbers on the nodes refer to the clique numbers in the RIP-ordering. Other RIP-orderings of the cliques can be found by choosing an arbitrary clique as the first and then numbering the cliques in any way which is increasing as one moves outward from the first clique in this tree. For example $C_3, C_2, C_5, C_1, C_6, C_4$ would be another RIP-ordering.

The functions $p(i_v \mid i_{\mathrm{pa}(v)})$ are hence defined on complete sets of the triangulated graph. For each clique C we collect the conditional probability tables $p(i_v \mid i_{\mathrm{pa}(v)})$ into a single term $\psi_C(i_C)$ by multiplying them. Triangulation may have created cliques to which no CPT corresponds. For each such clique the corresponding potential is identically equal to 1. Thus we have obtained the clique potential representation of $p(i_V)$ as

$$p(i_V) = \prod_{j=1}^{T} \psi_{C_j}(i_{C_j}). \tag{3.3}$$

The representation (3.3) is the fundamental representation for the subsequent computations. As such, a DAG and a corresponding factorization as in (3.2) is just one way of getting to the representation in (3.3) and one may alternatively specify this directly as shall be illustrated in Sect. 3.3.1.

3.2.2.2 Propagation—from Clique Potentials to Clique Marginals

To be able to answer queries, the `grain` object must be propagated, which means that the clique potentials must be calibrated (adjusted) to each other. Propagation is done with the `propagate()` method for `grain` objects:

```
> grn1c <- propagate(grn1c)
> summary(grn1c)

Independence network: Compiled: TRUE Propagated: TRUE
 Nodes : chr [1:8] "asia" "tub" "smoke" "lung" "bronc" ...
 Number of cliques:                 6
 Maximal clique size:               3
 Maximal state space in cliques:    8
```

The propagation algorithm works by turning the clique potential representation (3.3) into a representation in which each potential ψ_{C_j} is replaced by the marginal distribution $p(i_{C_j})$. This representation is called a *clique marginal representation*. This is done by working twice through the set of cliques and passing 'messages' between neighbouring cliques: first from the last clique in the RIP-ordering towards the first, i.e. inwards in the junction tree, and subsequently passing messages in the other direction.

In detail, the process is as follows. We start with the last clique C_T in the RIP ordering where $C_T = S_T \cup R_T$, $S_T \cap R_T = \emptyset$. The factorization (3.3) together with the factorization criterion (1.1) implies that $R_T \perp\!\!\!\perp (C_1 \cup \cdots \cup C_{T-1}) \setminus S_T \mid S_T$. Marginalizing over i_{R_T} gives

$$p(i_{C_1 \cup \ldots \cup C_{T-1}}) = \left(\prod_{j=1}^{T-1} \psi_{C_j}(i_{C_j}) \right) \sum_{i_{R_T}} \psi_{C_T}(i_{S_T}, i_{R_T}).$$

Let $\psi_{S_T}(i_{S_T}) = \sum_{i_{R_T}} \psi_{C_T}(i_{S_T}, i_{R_T})$. Then from the expression above we have

$$p(i_{R_T} \mid i_{S_T}) = \psi_{C_T}(i_{S_T}, i_{R_T}) / \psi_{S_T}(i_{S_T})$$

and hence

$$p(i_V) = p(i_{C_1 \cup \cdots \cup C_{T-1}}) p(i_{R_T} \mid i_{S_T}) = \left\{ \left(\prod_{j=1}^{T-1} \psi_{C_j}(i_{C_j}) \right) \psi_{S_T}(i_{S_T}) \right\} \frac{\psi_{C_T}(i_{C_T})}{\psi_{S_T}(i_{S_T})}.$$

The RIP ordering ensures that S_T is contained in the neighbour of C_T in the junction tree (one of the cliques $C_1, \ldots, C_{T-1}$), say C_k. We can therefore absorb ψ_{S_T} into ψ_{C_k} by setting $\psi_{C_k}(i_{C_k}) \leftarrow \psi_{C_k}(i_{C_k}) \psi_{S_T}(i_{S_T})$. We can think of the clique C_T *passing the message* ψ_{S_T} to its neighbour C_k, making a note of this by changing its own potential to $\psi_{C_T} \leftarrow \psi_{C_T} / \psi_{S_T}$, and C_k absorbing the message.

After this we now have $p(i_{C_1 \cup \cdots \cup C_{T-1}}) = \prod_{j=1}^{T-1} \psi_{C_j}(i_{C_j})$. We can then apply the same scheme to the part of the junction tree which has not yet been traversed. Continuing in this way until we reach the root of the junction tree yields

$$p(i_V) = p(i_{C_1}) p(i_{R_2} \mid i_{S_2}) p(i_{R_3} \mid i_{S_3}) \ldots p(i_{R_T} \mid i_{S_T}) \tag{3.4}$$

where $p(i_{C_1}) = \psi_{C_1}(i_{C_1}) / \sum_{i_{C_1}} \psi_{C_1}(i_{C_1})$. The resulting expression (3.4) is called a *set chain representation*. Note that the root potential now yields the joint marginal distribution of its nodes.

For some purposes we do not need to proceed further and the set chain representation is fully satisfactory. However, if we wish to calculate marginals to other cliques than the root clique, we need a second passing of messages. This time we work from the root towards the leaves in the junction tree and send messages with a slight twist, in the sense that this time we do not change the potential in the sending clique. Rather we do as follows:

Suppose C_1 is the parent of C_2 in the rooted junction tree. Then we have that $p(i_{S_2}) = \sum_{i_{C_1 \setminus S_2}} p(i_{C_1})$ and so

$$p(i_V) = p(i_{C_1}) \frac{p(i_{C_2})}{p(i_{S_2})} p(i_{R_3} \mid i_{S_3}) \ldots p(i_{R_T} \mid i_{S_T}).$$

Thus when the clique C_2 has absorbed its message by the operation

$$\psi_{C_2}(i_{C_2}) \leftarrow \psi_{C_2}(i_{C_2}) p(i_{S_2})$$

its potential is equal to the marginal distribution of its nodes. Proceeding in this way until we reach the leaves of the junction tree yields the clique marginal representation

$$p(i_V) = \prod_{j=1}^{T} p(i_{C_j}) / \prod_{j=2}^{T} p(i_{S_j}). \tag{3.5}$$

3.2.3 Absorbing Evidence and Answering Queries

Consider absorbing the evidence $i_E^* = (i_v^*, v \in E)$, i.e. that nodes in E are known to have a specific value. We note that

$$p(i_{V \setminus E} \mid i_E^*) \propto p(i_{V \setminus E}, i_E^*)$$

and hence evidence can be absorbed into the model by modifying the terms ψ_{C_j} in the clique potential representation (3.3) as follows: for every $v \in E$, we take an arbitrary clique C_j with $v \in C_j$. Entries in ψ_{C_j} which are locally inconsistent with the evidence, i.e. entries i_{C_j} for which $i_v \neq i_v^*$, are set to zero and all other entries are unchanged. Evidence can be entered before or after propagation without changing final results.

To answer queries, we carry out the propagation steps above leading to a clique marginal representation where the potentials become $\psi_{C_j}(i_{C_j}) = p(i_{C_j} | i_E^*)$. In this process we note that processing of the root potential to find $p(i_{C_1} \mid i_E^*)$ involves computation of $\sum_{i_{C_1}} \psi_1(i_{C_1})$ which is equal to $p(i_E^*)$. Hence the probability of the evidence comes at no extra computational cost.

Evidence is entered with `setFinding()` which creates a new `grain` object:

```
> grn1c.ev  <-
+ setFinding(grn1c,nodes=c("asia","dysp"), states=c("yes","yes"))
```

To obtain $p(i_v \mid i_E^*)$ for some $v \in V \setminus E$, we locate a clique C_j containing v and marginalize as $\sum_{i_{C_j \setminus \{v\}}} p(i_{C_j})$. Based on (3.5) the `grain` objects with and without evidence can now be queried to give marginal probabilities using `querygrain()`:

```
> querygrain(grn1c.ev,nodes=c("lung","bronc"), type="marginal")

$lung
lung
    yes      no
0.09953 0.90047

$bronc
bronc
   yes     no
0.8114 0.1886
```

```
> querygrain(grn1c,nodes=c("lung","bronc"), type="marginal")

$lung
lung
  yes    no
0.055 0.945

$bronc
bronc
 yes   no
0.45 0.55
```

The evidence in a `grain` object can be retrieved with the `getFinding()` function while the probability of observing the evidence is obtained using the `pFinding()` function:

```
> getFinding(grn1c.ev)

Finding:
     variable state
[1,] asia     yes
[2,] dysp     yes
Pr(Finding)= 0.004501

> pFinding(grn1c.ev)

[1] 0.004501
```

Suppose we want the distribution $p(i_U \mid i^*_E)$ for a set $U \subset V \setminus E$. If there is a clique C_j such that $U \subset C_j$ then the distribution is simple to find by summing $p(i_{C_j})$ over the variables in $C_j \setminus U$. If no such clique exists we can obtain $p(i_U \mid i^*_E)$ by calculating $p(i^*_U, i^*_E)$ for all possible configurations i^*_U of U and then normalizing the result: this can be computationally demanding if U has a large state space.

```
> querygrain(grn1c.ev,nodes=c("lung","bronc"), type="joint")

     bronc
lung      yes      no
  yes 0.06298 0.03654
  no  0.74842 0.15205
> querygrain(grn1c.ev,nodes=c("lung","bronc"), type="conditional")

     bronc
lung      yes     no
  yes 0.07762 0.1938
  no  0.92238 0.8062
```

Note that the latter result is the conditional distribution of `lung` given `bronc`—but also conditional on the evidence.

However, if it is known beforehand that interest will be in the joint distribution of a specific set U of variables, one can ensure that the set U is contained in a single clique in the triangulated graph. This can for example be done by first moralizing, then adding edges between all nodes in U, and finally triangulating the resulting graph. The price to be paid is that the cliques may become larger and since computational complexity is exponential in the largest clique, this can be prohibitive.

To do this in practice we first need to compile the grain again

```
> grn1c2 <- compile(grn1, root=c("lung", "bronc", "tub"),
                    propagate=TRUE)
> grn1c2.ev  <- setFinding(grn1c2,nodes=c("asia","dysp"),
+                          states=c("yes","yes"))
```

Now compare the computing times: the second method is much faster:

```
> system.time({for (i in 1:50)
+                querygrain(grn1c.ev,nodes=c("lung","bronc","tub"),
+                           type="joint")
+            })

   user  system elapsed
    1.5     0.0     1.5

> system.time({for (i in 1:50)
+                querygrain(grn1c2.ev,nodes=c("lung","bronc","tub"),
+                           type="joint")
+            })

   user  system elapsed
   0.02    0.00    0.01
```

Evidence can be entered incrementally by calling setFinding() repeatedly. It is most efficient to set propagate=FALSE in setFinding() and then only call the propagate() method for grain objects at the end:

```
> grn1c.ev <- setFinding(grn1c,nodes="asia", states="yes",
+                        propagate=FALSE)
> grn1c.ev <- setFinding(grn1c.ev,nodes="dysp", states="yes",
+                        propagate=FALSE)
> grn1c.ev <- propagate(grn1c.ev)
```

Evidence can be retracted (removed) using the retractFinding() function:

```
> grn1c.ev <- retractFinding(grn1c.ev, nodes="asia")
> getFinding(grn1c.ev)

Finding:
     variable state
[1,] dysp     yes
Pr(Finding)= 0.004501
```

Omitting nodes implies that all evidence is retracted, i.e. that the grain object is reset to its original status.

3.3 Further Topics

3.3.1 Building a Network from Data

A `grain` object can also be built from a dataframe of cases in various ways, as illustrated below.

One way is to build it is to use data in combination with a graph such as, for example, the directed acyclic graph `chestdag` specified in Sect. 3.1.2.

The data `chestSim500` from the **gRbase** package is generated from our fictitious example using the command `simulate()` method described in Sect. 3.3.3 below.

When building a `grain` object this way, the CPTs are estimated from data in `chestSim500` as the relative frequencies. To avoid zeros in the CPTs one can choose to add a small number, e.g. `smooth=0.1` to all values, corresponding to a Bayesian estimate based on prior Dirichlet distributions for the CPT entries:

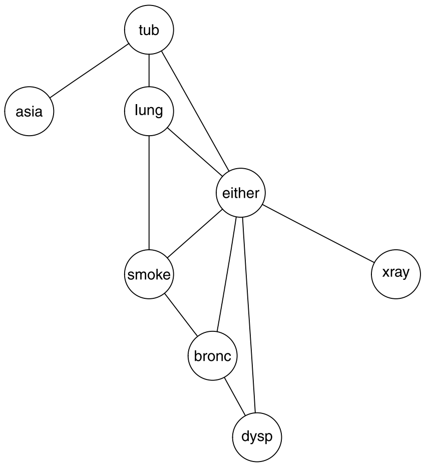
```
> library(gRbase)
> data(chestSim500, package='gRbase')
> simdagchest <- grain(chestdag, data=chestSim500)
> simdagchest <- compile(simdagchest, propagate=TRUE, smooth=.1)
> querygrain(simdagchest, nodes =c("lung","bronc"),type="marginal")

$lung
lung
  yes    no
0.046 0.954

$bronc
bronc
  yes    no
0.454 0.546
```

Alternatively, a `grain` object can be built from an undirected (but triangulated) graph rather than a Bayesian network, making some steps of the process of compilation redundant. The undirected triangulated graph for the compiled chest clinic example can be specified as:

```
> g<-list(~asia : tub, ~either : lung : tub, ~either : lung : smoke,
+      ~bronc : either : smoke, ~bronc : dysp : either, ~either :
+         xray)
> myug <- ugList(g)
```

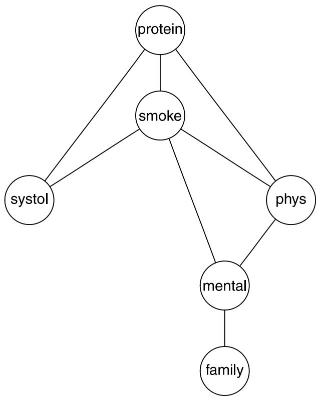
A `grain` object can now be built from the graph and the data. In this process, the clique potentials are directly estimated by the appropriate frequencies:

```
> simugchest <- grain(myug, data=chestSim500)
> simugchest <- compile(simugchest, propagate=TRUE)
> plot(simugchest)
```



This is natural when directions are not known beforehand. For example, using the `reinis` data with a model selection procedure yields

```
> data(reinis, package='gRbase')
> m0 <- dmod(~.^., data=reinis)
> m1 <- stepwise(m0)
> reinisgrain <- grain(as(m1,"graphNEL"), data=reinis)
> plot(reinisgrain)
> reinisgrain <- compile(reinisgrain, propagate=TRUE)
> querygrain(reinisgrain,nodes=c("phys","protein"), type="marginal")

$protein
protein
     y      n
0.5763 0.4237

$phys
phys
     y      n
0.5035 0.4965
```



Now evidence can be entered and revised probabilities found as usual:

```
> reinisgrain.ev  <-
+ setFinding(reinisgrain,
+            nodes=c("systol","smoke","mental"), states=c("y","y","y"))
> querygrain(reinisgrain.ev,nodes=c("phys","protein"), type="marginal")
```

```
$protein
protein
     y      n 
0.6744 0.3256 

$phys
phys
     y      n 
0.2776 0.7224 
```

3.3.2 Bayesian Networks with RHugin

The package **RHugin** (see http://rhugin.r-forge.r-project.org) provides an Application Programmer's Interface (API) to `HUGIN` in the `R` language. It consists of a basic library of functions which mirrors the `C` API

http://www.hugin.com/developer/documentation/API_Manuals/

provided with `HUGIN`. More precisely, every command in the `C` API of `HUGIN` of the form `h_something` has an `R`-variant called `RHugin_something`, e.g. `RHugin_domain_propagate` uses `.Call` to invoke the `HUGIN` function `h_domain_propagate` etc. In this way, the full functionality of `HUGIN` becomes available within `R`.

In addition, **RHugin** provides a few higher level functions based on this API which enables simple operations for Bayesian networks to be carried out, for example such as those described in the previous sections. For the first simple illustrations we repeat the basic steps above using **RHugin** instead of **gRain**.

We first create the domain

```
> library(RHugin)
> RHchestClinic <- hugin.domain()
```

and subsequently create nodes and give them states

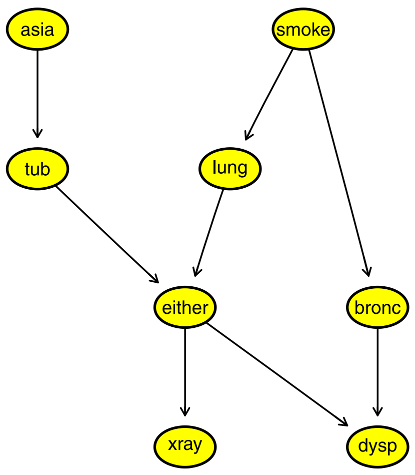
```
> chestNames <- c("asia", "smoke", "tub", "lung", "bronc",
+                 "either", "xray", "dysp")
> for(node in chestNames)
+   add.node(RHchestClinic, node, states = c("yes", "no"))
```

Then nodes are connected with edges to form the DAG

```
> add.edge(RHchestClinic, "tub", "asia")
> add.edge(RHchestClinic, "lung", "smoke")
> add.edge(RHchestClinic, "bronc", "smoke")
> add.edge(RHchestClinic, "either", c("tub", "lung"))
> add.edge(RHchestClinic, "xray", "either")
> add.edge(RHchestClinic, "dysp", c("either", "bronc"))
```

The network now exists and can be displayed using **Rgraphviz**

```
> library(Rgraphviz)
> plot(RHchestClinic)
```



At this point the network has default (uniform) probability tables:

```
> get.table(RHchestClinic, "dysp")
```

```
  dysp either bronc Freq
1  yes    yes   yes    1
2   no    yes   yes    1
3  yes     no   yes    1
4   no     no   yes    1
5  yes    yes    no    1
6   no    yes    no    1
7  yes     no    no    1
8   no     no    no    1
```

These can now be changed manually:

```
> cpt <- get.table(RHchestClinic, "asia")
> cpt$Freq <- c(0.01, 0.99)
> set.table(RHchestClinic, "asia", cpt)
> cpt <- get.table(RHchestClinic, "tub")
> cpt$Freq <- c(5, 95, 1, 99)
> set.table(RHchestClinic, "tub", cpt)
> cpt <- get.table(RHchestClinic, "either")
> cpt
```

```
  either tub lung Freq
1    yes yes  yes    1
2     no yes  yes    1
3    yes  no  yes    1
4     no  no  yes    1
5    yes yes   no    1
6     no yes   no    1
```

```
7    yes  no   no    1
8     no  no   no    1
> cpt$Freq <- c(1,0,1,0,1,0,0,1)
> set.table(RHchestClinic, "either", cpt)
>
```

or using available data to populate one of the tables:

```
> set.table(RHchestClinic,"dysp",chestSim500)
```

leading to

```
> get.table(RHchestClinic, "dysp")
  dysp either bronc Freq
1  yes    yes   yes   10
2   no    yes   yes    2
3  yes     no   yes  176
4   no     no   yes   39
5  yes    yes    no   12
6   no    yes    no    5
7  yes     no    no   29
8   no     no    no  227
```

Note that the CPTs are not yet normalized. In HUGIN this happens at the stage of compilation. We can also let most (or all) tables be based on frequencies in the dataframe:

```
> set.table(RHchestClinic, "smoke", chestSim500)
> set.table(RHchestClinic, "bronc", chestSim500)
> set.table(RHchestClinic, "lung", chestSim500)
> set.table(RHchestClinic, "xray", chestSim500)
> get.table(RHchestClinic, "smoke")

  smoke Freq
1   yes  238
2    no  262
```

If we compile the network we find that tables have become normalized:

```
> compile(RHchestClinic)
> get.table(RHchestClinic, "dysp")

  dysp either bronc   Freq
1  yes    yes   yes 0.8333
2   no    yes   yes 0.1667
3  yes     no   yes 0.8186
4   no     no   yes 0.1814
5  yes    yes    no 0.7059
6   no    yes    no 0.2941
7  yes     no    no 0.1133
8   no     no    no 0.8867
```

The network is now ready for absorbing evidence and calculating revised probabilities:

```
> set.finding(RHchestClinic, "asia","yes")
> set.finding(RHchestClinic, "dysp","yes")
> propagate(RHchestClinic)
> get.belief(RHchestClinic, "lung")
```

```
    yes      no
0.07729 0.92271
```

```
> get.belief(RHchestClinic, "bronc")
```

```
  yes    no
0.806 0.194
```

Note the values are somewhat different from those previously obtained. This is due to the fact that probabilities are estimated from the (simulated) data rather than specified exactly.

3.3.3 Simulation

It is possible to simulate data from a Bayesian network model. The methods use the current clique potentials to do this and thus generates values conditional on all evidence entered in the `grain` object. It uses the method of random propagation as described in Dawid (1992); see also Cowell et al. (1999, p. 99). If a domain is not propagated when `simulate()` is applied, `simulate()` will force this to happen automatically.

```
> simulate(grn1c.ev, nsim=5)
```

```
  asia tub smoke lung bronc either xray dysp
1  yes  no   yes   no   yes     no   no  yes
2  yes yes   yes   no   yes    yes  yes  yes
3  yes  no   yes   no    no     no   no  yes
4  yes  no    no   no   yes     no   no  yes
5  yes  no    no   no   yes     no   no  yes
```

One application of such a simulation is to obtain the joint distribution of `lung` and `bronc` conditional on the finding:

```
> xtabs(~lung+bronc, data=simulate(grn1c.ev, nsim=1000))/1000
```

```
     bronc
lung    yes    no
  yes 0.070 0.033
  no  0.757 0.140
```

The result of the simulation is close to the exact result given in Sect. 3.2.3. A simulate() method is also available with **RHugin**, but this only works if the domain has been propagated.

```
> simulate(RHchestClinic, nsim=5)
```

```
  asia smoke tub lung bronc either xray dysp
1  yes   yes  no   no   yes     no   no  yes
2  yes    no  no   no   yes     no  yes  yes
3  yes    no  no   no   yes     no   no  yes
4  yes    no  no   no   yes     no   no  yes
5  yes    no  no   no   yes     no   no  yes
```

3.3.4 Prediction

A `predict` method is available for `grain` objects for predicting a set of "responses" from a set of "explanatory variables". Two types of predictions can be made. The default is `type="class"` which assigns the value to the class with the highest probability:

```
> mydata

  bronc dysp either lung tub asia xray smoke
1   yes  yes    yes  yes  no   no  yes   yes
2   yes  yes    yes  yes  no   no  yes    no
3   yes  yes    yes   no yes   no  yes   yes
4   yes  yes     no   no  no  yes  yes    no

> predict(grn1c, response=c("lung","bronc"), newdata=mydata,
+  predictors=c("smoke", "asia", "tub", "dysp", "xray"), type="class")

$pred
$pred$lung
[1] "yes" "no"  "no"  "no"

$pred$bronc
[1] "yes" "yes" "yes" "yes"


$pFinding
[1] 0.0508476 0.0111697 0.0039778 0.0001083
```

The output should be read carefully: Conditional on the first observation in `mydata`, the most probable value of `lung` is `"yes"` and the same is the case for `bronc`. This is not in general the same as saying that the most likely configuration of the two variables `lung` and `bronc` is `"yes"`.

The entire conditional distribution can be obtained in **gRain** by setting `type='dist'`:

```
> predict(grn1c, response=c("lung","bronc"), newdata=mydata,
+  predictors=c("smoke", "asia", "tub", "dysp", "xray"), type="dist")

$pred
$pred$lung
        yes     no
[1,] 0.7745 0.2255
[2,] 0.3268 0.6732
[3,] 0.1000 0.9000
[4,] 0.3268 0.6732

$pred$bronc
        yes     no
[1,] 0.7182 0.2818
[2,] 0.6373 0.3627
[3,] 0.6585 0.3415
[4,] 0.6373 0.3627


$pFinding
[1] 0.0508476 0.0111697 0.0039778 0.0001083
```

The jointly most probably configuration can be found by using the option `equilibrium ="max"` with **RHugin**. `HUGIN` uses the max-propagation algorithm described in Dawid (1992); see also Cowell et al. (1999, p. 97 ff.). For the third datapoint we get

```
> initialize(RHchestClinic)

A Hugin domain
Nodes: asia smoke tub lung bronc either xray dysp
Edges:
  asia -> tub
  smoke -> bronc
  smoke -> lung
  tub -> either
  lung -> either
  bronc -> dysp
  either -> dysp
  either -> xray

> set.finding(RHchestClinic,"smoke","yes")
> set.finding(RHchestClinic,"asia","no")
> set.finding(RHchestClinic,"tub","yes")
> set.finding(RHchestClinic,"dysp","yes")
> set.finding(RHchestClinic,"xray","yes")
```

The joint probability of the evidence is

```
> propagate(RHchestClinic)
> pev<-get.normalization.constant(RHchestClinic)
> pev

[1] 0.003687
```

and the most likely configuration is

```
> propagate(RHchestClinic,equilibrium ="max")
> get.belief(RHchestClinic,"either")
yes  no
  1   0

> get.belief(RHchestClinic,"lung")

    yes      no
0.08676 1.00000

> get.belief(RHchestClinic,"bronc")

   yes     no
1.0000 0.5627
```

The most probable configuration of the unobserved nodes `either, lung, bronc` is found by combining states where `get.belief()` returns 1.00, in this case `yes, no, yes`. The second number indicates how much the joint probability decreases if the state at that particular node is changed, i.e. the joint probability of `yes,yes,yes`, is .08676 times the maximal probability. The probability of the most probable configuration and evidence jointly is obtained via the normalization constant again

```
> pmax<-get.normalization.constant(RHchestClinic)
> pmax
```

```
[1] 0.002171
```

So the conditional probability of the most probable configuration given the evidence is

```
> predprob<-pmax/pev
> predprob
```

```
[1] 0.5888
```

To `simulate` with **RHugin**, we now need to propagate again with the default `"sum"` option.

3.3.5 Working with HUGIN Files

With `HUGIN`, the specifications of a BN are read or saved in a textfile in a format known as a `.net`. HUGIN can also save and read domains in its own binary format `.hkb` which can contain further information in the form of compilation, evidence, and propagation results.

A `grain` object saved in this format can be loaded into `R` using the `loadHuginNet()` function in **gRain**:

```
> chest <- loadHuginNet("ChestClinic.net")

> chest
```

```
Independence network: Compiled: FALSE Propagated: FALSE
 Nodes: chr [1:8] "PositiveXray" "Bronchitis" "Dyspnoea" ...
```

`HUGIN` distinguishes between node names and node labels. Node names have to be unique; node labels need not be so. When creating a BN in `HUGIN` node names are generated automatically as `C1`, `C2` etc. The user can choose to give more informative labels or to give informative names. Typically one would do the former. Therefore `loadHuginNet` uses node labels (if given) from the netfile and otherwise node names.

This causes two types of problems. First, `HUGIN` allows spaces and special characters (e.g. “?”) in variable labels, but these are not allowed in **gRain**. If such a name is found by `loadHuginNet`, it is converted as follows: special characters are removed, the first letter after a space is capitalized and then spaces are removed. Hence the label “visit to Asia?” in a `net` file will be converted to “visitToAsia”. The same convention applies to states of variables. Secondly, because node labels in the net file are used as node names in **gRain** we may end up with two nodes having the same name, which is obviously not permitted. To resolve this **gRain** will in such cases force the node names in **gRain** to be the node names rather than the node labels from the net file. For example, if nodes A and B in a net file both have label foo, then the nodes in **gRain** will be denoted A and B. Note that this approach is not entirely foolproof: If there is a node C with label A, then we have just moved the problem. So the scheme above is applied recursively until all ambiguities are resolved.

A grain can be saved in the .net format with the saveHuginNet() function.

```
> saveHuginNet(reinisgrain,file="reinisgrain.net")
```

Note that reinisgrain does not have a DAG by default, so the save function constructs a DAG which has the same conditional independence structure as the triangulated graph defining the grain object.

RHugin interacts continuously with HUGIN but has also read and write functions write.rhd() and read.rhd(). For example we can now create a domain in **RHugin** as

```
> RHreinis<-read.rhd("reinisgrain.net")
> RHreinis

A Hugin domain
Nodes: family mental phys systol smoke protein
Edges:
  mental -> family
  phys -> mental
  systol -> protein
  smoke -> mental
  smoke -> phys
  smoke -> protein
  smoke -> systol
  protein -> phys
```

We can now operate fully in RHreinis with **RHugin**, for example

```
> get.table(RHreinis,"mental")
> set.finding(RHreinis,"mental","y")
> set.finding(RHreinis,"protein","n")
> compile(RHreinis)
> propagate(RHreinis)
> get.normalization.constant(RHreinis)
> get.belief(RHreinis,"smoke")
> write.rhd(RHreinis,"RHreinis.hkb",type="hkb")
> write.rhd(RHreinis,"RHreinis.net",type="net")
```

The file RHreinis.hkb will now be in a binary format readable by HUGIN (or **RHugin**) and contain a compiled and propagated domain with its evidence and the associated .net file, whereas RHreinis.net will be a textfile identical to reinisgrain.net. Similarly, **RHugin** can also read domains which are saved in .hkb format.

It is important to emphasize the relationship between **RHugin** and **gRain** on the one side and HUGIN on the other: **gRain** works entirely within R, creates R objects, and only relates to HUGIN through its facilities for reading and writing .net files. In contrast, domains, nodes, edges etc. of an **RHugin**-domain are not R objects as such. They are *external pointers* to objects which otherwise exist within HUGIN. So, for example, a statement of the form

```
> newRHreinis<-RHreinis
```

does not create a new R object, but just an extra pointer to the same HUGIN domain. This means that if anything is changed in RHreinis, it will automatically change in the same way in newRHreinis and vice versa.

3.4 Learning Bayesian Networks

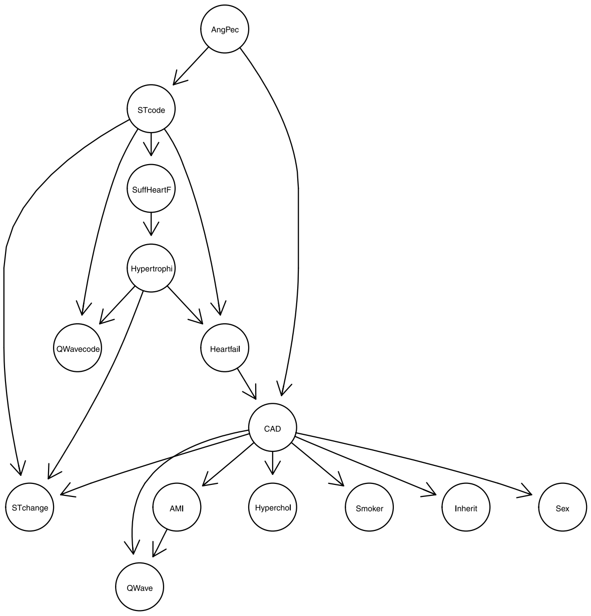
Hitherto in this chapter it has been assumed that the Bayesian network was known in advance. In practice it is often necessary to construct the network based on an observed dataset—a topic known in the machine learning community as *structural learning* (in contrast to *parameter* learning) and in the statistical community as *model selection.*

Model selection algorithms for Gaussian graphical models based on DAGs are described in Chap. 4. Available algorithms include the PC-algorithm (Spirtes and Glymour 1991) and various algorithms in the **bnlearn** package: these can also be used to select discrete Bayesian networks.

As illustration we consider a dataframe cad1 supplied along with the **gRbase** package. This contains data on coronary artery disease from a Danish heart clinic. In all 14 discrete variables were recorded for 236 patients at the clinic including five background variables (sex, hypercholesterolemia, smoking, heridary disposition and workload), one recording whether or not the patient has coronary artery disease, four variables representing disease manifestation (hypertrophy, previous myocardial infarct, angina pectoris, other heartfailures), and four clinical measurements (Q-wave, T-wave, Q-wave informative and T-wave informative). These data were used as an example of chain graph modelling in Højsgaard and Thiesson (1995).

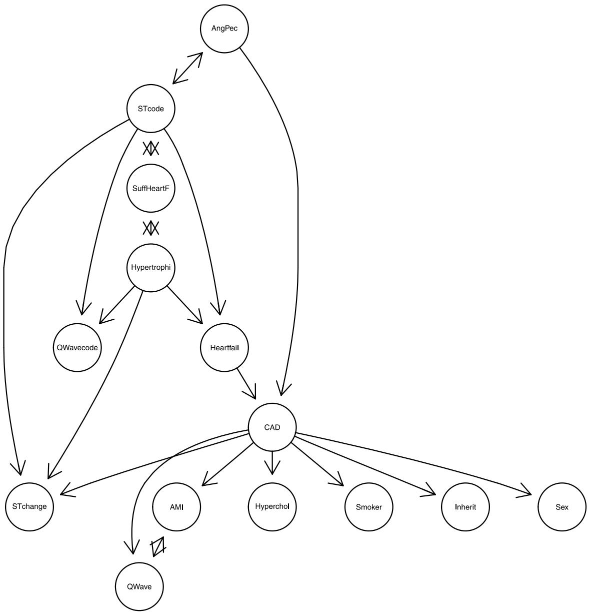
As a first attempt we can apply the hill-climbing algorithm implemented in the hc function in the **bnlearn** package. This is a greedy algorithm to find a model optimizing a score: various scores may be used, and here we choose to minimize the Bayesian Information Criterion (BIC).

```
> library(gRbase)
> data(cad1, package='gRbase')
> library(bnlearn)
> cad.bn <- hc(cad1)
> plot(as(amat(cad.bn), "graphNEL"))
```



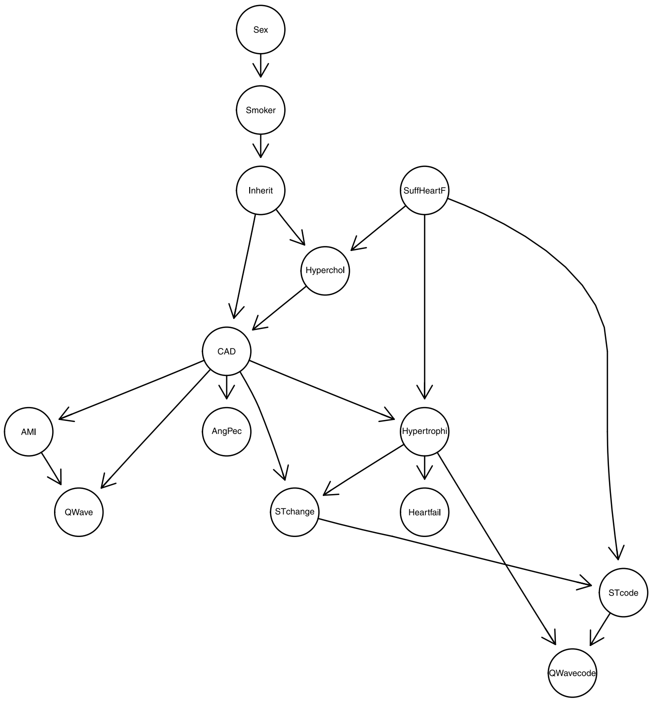
As described in more detail in Sect. 4.5.1, DAGs can only be selected up to Markov equivalence, so it is useful to see which DAGs are Markov equivalent to the selected one. These may be represented as an essential graph, using the `essentialGraph` function in the **ggm** package.

```
> library(ggm)
> plot(as(essentialGraph(amat(cad.bn)), "graphNEL"))
```



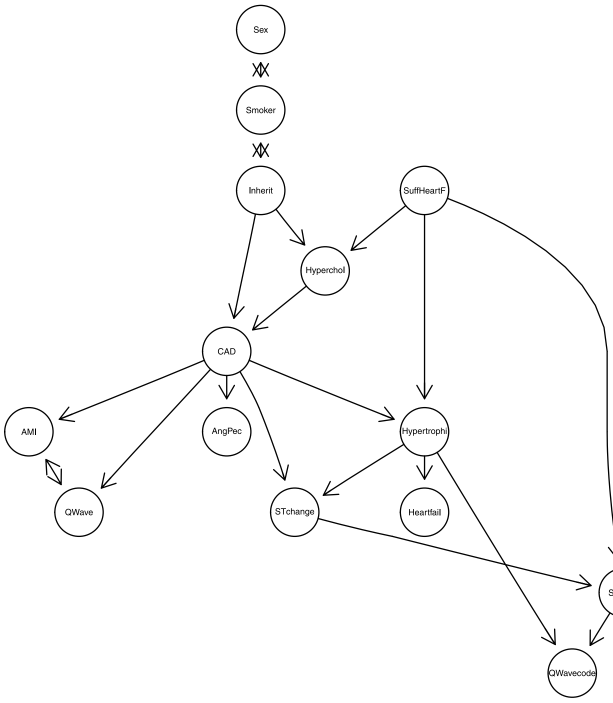
This model is implausible, since it suggests amongst other things that whether or not a patient has coronary artery disease (CAD) determines their sex and whether they smoke. A better approach is to incorporate our prior knowledge of the system under study into the model selection process. We do this by dividing the variables into the four blocks described above, namely background variables, disease, disease manifestations and clinical measurements. Note that we treat hypertrophy as a disease variable, following Højsgaard and Thiesson (1995). We restrict the model selection process by blacklisting (i.e., disallowing) arrows that point from a later to an earlier block. The following code shows how this may be done. First we construct an adjacency matrix containing the disallowed edges, then we convert this into a dataframe using the `get.edgelist` function in the **igraph** package. This is then passed to the `hc` function.

```
> block <- c(1,3,3,4,4,4,4,1,2,1,1,1,3,2)
> blM <- matrix(0, nrow=14, ncol=14)
> rownames(blM) <- colnames(blM) <- names(cad1)
> for (b in 2:4) blM[block==b, block<b] <- 1
> library(igraph)
> blackL <- data.frame(get.edgelist(as(blM, "igraph")))
> names(blackL) <- c("from", "to")
> cad.bn1 <- hc(cad1, blacklist=blackL)
> plot(as(amat(cad.bn1), "graphNEL"))
```



Finally, we again examine the essential graph of the selected DAG:

```
> library(ggm)
> plot(as(essentialGraph(amat(cad.bn1)), "graphNEL"))
```



This is more plausible. To create the corresponding grain object, we can use

```
> cad.gr <- as(amat(cad.bn1), "graphNEL")
> cad.grain <- grain(cad.gr, data=cad1)
```

and proceed as before.