```
##
## Attaching package:  'igraph'
## The following objects are masked from 'package:gRbase':
##
##     edges, is_dag, topo_sort
## The following objects are masked from 'package:stats':
##
##     decompose, spectrum
## The following object is masked from 'package:base':
##
##     union
```

Chapter 1

Graphs and Conditional Independence

As of major version 2 of **gRbase**, *that is versions 2.x.y,* **gRbase** *no longer depends on the packages* graph, Rgraphviz, *and* RBGL *packages. Graph functionality in these packages now relies either on the* igraph *package or on graph algorithms implemented in* gRbase. *This document reflects these changes.*

As a consequence, this document provides an up-to-date version of Chapter 1 in the book Graphical Models with R (2012); hereafter abbreviated GMwR, see Højsgaard et al. [2012].

This document also reflects that since GMwR was published in 2012, some packages that are mentioned in GMwR are no longer on CRAN. This includes the packages lcd *and* sna.

In this document it has been emphasized if a function has been imported from **igraph** *or if it is native function from* **gRbase** *by writing* **'igraph::this_function()'** *and* **'gRbase::this_function()'**

One notable feature that is not available in this version of **gRbase** *are functions related to maximal prime subgraph decomposition. They may be reimplented at a later stage.*

Contents

1.1 Introduction

A graph as a *mathematical* object may be defined as a pair $\mathcal{G} = (V, E)$, where V is a set of *vertices* or *nodes* and E is a set of *edges*. Each edge is associated with a pair of nodes, its *endpoints*. Edges may in general be directed, undirected, or bidirected. Graphs are typically visualized by representing nodes by circles or points, and edges by lines, arrows, or bidirected arrows. We use the notation $\alpha - \beta$, $\alpha \to \beta$, and $\alpha \leftrightarrow \beta$ to denote edges between α and β. Graphs are useful in a variety of applications, and a number of packages for working with graphs are available in `R`.

In statistical applications we are particularly interested in two special graph types: undirected graphs and directed acyclic graphs (often called DAGs).

The **gRbase** package supplements **igraph** by implementing some algorithms useful in graphical modelling. **gRbase** also provides two wrapper functions, *ug()* and *dag()*, for easily creating undirected graphs and DAGs represented either as `igraph` objects or adjacency matrices.

The first sections of this chapter describe some of the most useful functions available when working with graphical models. These come variously from the **gRbase** and **igraph**, but it is not usually necessary to know which.

As *statistical* objects, graphs are used to represent models, with nodes representing model variables (and sometimes model parameters) in such a way that the independence structure of the model can be read directly off the graph. Accordingly, a section of this chapter is devoted to a brief description of the key concept of conditional independence and explains how this is linked to graphs. Throughout the book we shall repeatedly return to this in more detail.

1.2 Graphs

Our graphs have a finite node set V and for the most part they are *simple graphs* in the sense that they have no loops nor multiple edges. Two vertices α and β are said to be *adjacent*, written $\alpha \sim \beta$, if there is an edge between α and β in $\mathcal{G}$, i.e. if either $\alpha - \beta$, $\alpha \to \beta$, or $\alpha \leftrightarrow \beta$.

In this chapter we primarily represent graphs as `igraph` objects, and except where stated otherwise, the functions we describe operate on these objects.

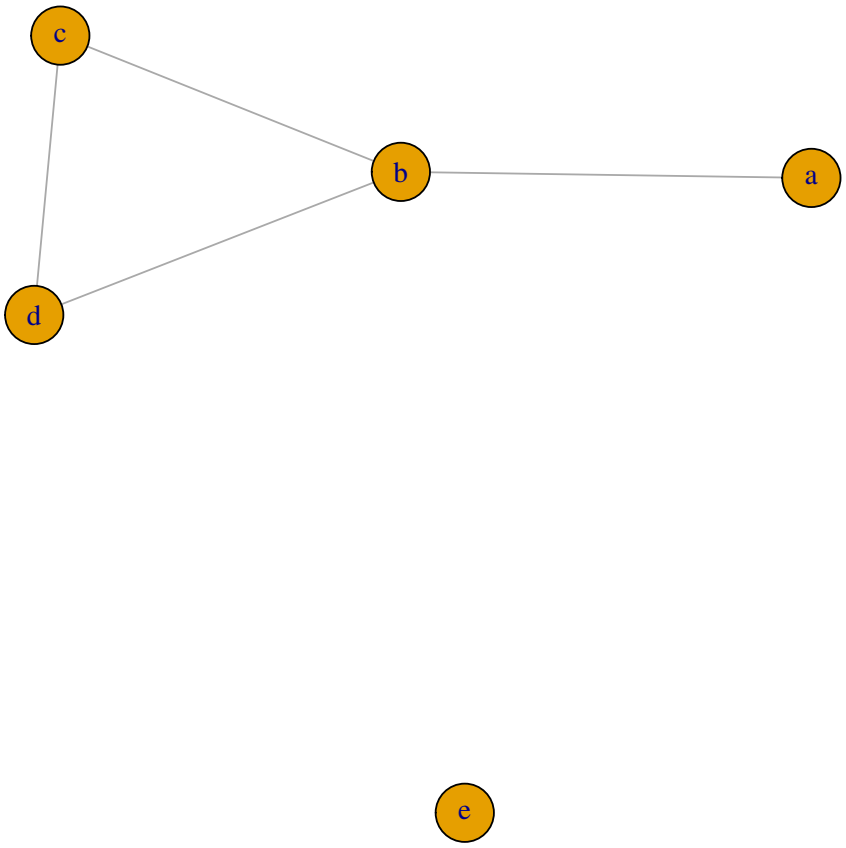
1.2.1 Undirected Graphs

An undirected graph may be created using the *ug()* function. The graph can be specified using a list of formulas, a single formula or a list of vectors. Thus the following forms are equivalent:

```
library(gRbase)
ug0 <- gRbase::ug(~a:b, ~b:c:d, ~e)
```

```
ug0 <- gRbase::ug(~a:b + b:c:d + e)
ug0 <- gRbase::ug(~a*b + b*c*d + e)
ug0 <- gRbase::ug(c("a", "b"), c("b", "c", "d"), "e")
ug0

## IGRAPH 3c38e2b UN-- 5 4 --
## + attr: name (v/c)
## + edges from 3c38e2b (vertex names):
## [1] a--b b--c b--d c--d
```

```
plot(ug0)
```



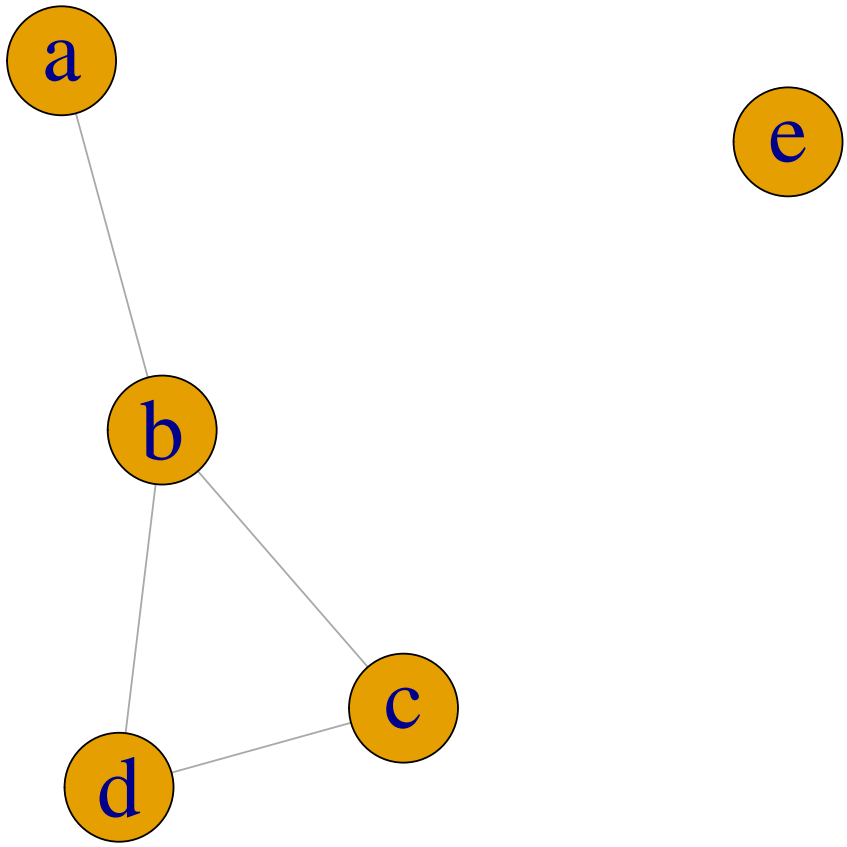
The default size of vertices and their labels is quite small. This is easily changed by setting certain attributes on the graph, see Sect. 1.4.2 for examples. However, to avoid changing these attributes for all the graphs shown in the following we have defined a small plot function *myplot()*. There are also various facilities for controlling the layout. For example, we may use a layout algorithm called `layout.fruchterman.reingold` as follows:

```
myplot <- function(x, layout=layout.fruchterman.reingold(x), ...) {
  V(x)$size <- 30
  V(x)$label.cex <- 3
  plot(x, layout=layout, ...)
```

```
  return(invisible())
}
```

The graph `ug0i` is then displayed with:

```
myplot(ug0)
```



Per default the *ug()* function returns an `igraph` object, but the option `result="matrix"` lead it to return an adjacency matrix instead. For example,

```
ug0i <- gRbase::ug(~a:b + b:c:d + e, result="matrix")
ug0i

##   a b c d e
## a 0 1 0 0 0
## b 1 0 1 1 0
## c 0 1 0 1 0
## d 0 1 1 0 0
## e 0 0 0 0 0
```

Different represents of a graph can be obtained by coercion:

```
as(ug0, "matrix")

##   a b c d e
## a 0 1 0 0 0
## b 1 0 1 1 0
## c 0 1 0 1 0
## d 0 1 1 0 0
## e 0 0 0 0 0

as(ug0, "dgCMatrix")

## 5 x 5 sparse Matrix of class "dgCMatrix"
##   a b c d e
## a . 1 . . .
## b 1 . 1 1 .
## c . 1 . 1 .
## d . 1 1 . .
## e . . . . .

as(ug0i, "igraph")

## IGRAPH e0b67ed UN-- 5 4 --
## + attr: name (v/c), label (v/c)
## + edges from e0b67ed (vertex names):
## [1] a--b b--c b--d c--d
```

Edges can be added and deleted using the *addEdge()* and *removeEdge()* functions:

```
## Using gRbase
ug0a <- gRbase::addEdge("a", "c", ug0)
ug0a <- gRbase::removeEdge("c", "d", ug0)
```

```
## Using igraph
ug0a <- igraph::add_edges(ug0, c("a", "c"))
ug0a <- igraph::delete_edges(ug0, c("c|d"))
```

The nodes and edges of a graph can be retrieved with *nodes()* and *edges()* functions.

```
## Using gRbase
gRbase::nodes(ug0)

## [1] "a" "b" "c" "d" "e"

gRbase::edges(ug0) |> str()

## List of 5
##  $ a: chr "b"
##  $ b: chr [1:3] "a" "c" "d"
##  $ c: chr [1:2] "b" "d"
##  $ d: chr [1:2] "b" "c"
##  $ e: chr(0)
```

```
## Using igraph
igraph::V(ug0)

## + 5/5 vertices, named, from 3c38e2b:
## [1] a b c d e

igraph::V(ug0) |> attr("names")

## [1] "a" "b" "c" "d" "e"

igraph::E(ug0)

## + 4/4 edges from 3c38e2b (vertex names):
## [1] a--b b--c b--d c--d

igraph::E(ug0) |> attr("vnames")

## [1] "a|b" "b|c" "b|d" "c|d"
```

```
gRbase::maxClique(ug0) ## |> str()

## $maxCliques
```

```
## $maxCliques[[1]]
## [1] "e"
##
## $maxCliques[[2]]
## [1] "a" "b"
##
## $maxCliques[[3]]
## [1] "b" "c" "d"

gRbase::get_cliques(ug0) |> str()

## List of 3
##  $ : chr "e"
##  $ : chr [1:2] "a" "b"
##  $ : chr [1:3] "b" "c" "d"

## Using igraph
igraph::max_cliques(ug0) |>
    lapply(function(x) attr(x, "names"))  |> str()

## List of 3
##  $ : chr "e"
##  $ : chr [1:2] "a" "b"
##  $ : chr [1:3] "b" "c" "d"
```

A *path* (of length n) between α and β in an undirected graph is a set of vertices $\alpha = \alpha_0, \alpha_1, \ldots, \alpha_n = \beta$ where $\alpha_{i-1} - \alpha_i$ for $i = 1, \ldots, n$. If a path $n\alpha = \alpha_0, \alpha_1, \ldots, \alpha_n = \beta$ has $\alpha = \beta$ then the path is said to be a *cycle* of length n. nnn A subset $D \subset V$ in an undirected graph is said to *separate* $A \subset V$ from $B \subset V$ if every path between a vertex in A and a vertex in B contains a vertex from D.

```
gRbase::separates("a", "d", c("b", "c"), ug0)

## [1] TRUE
```

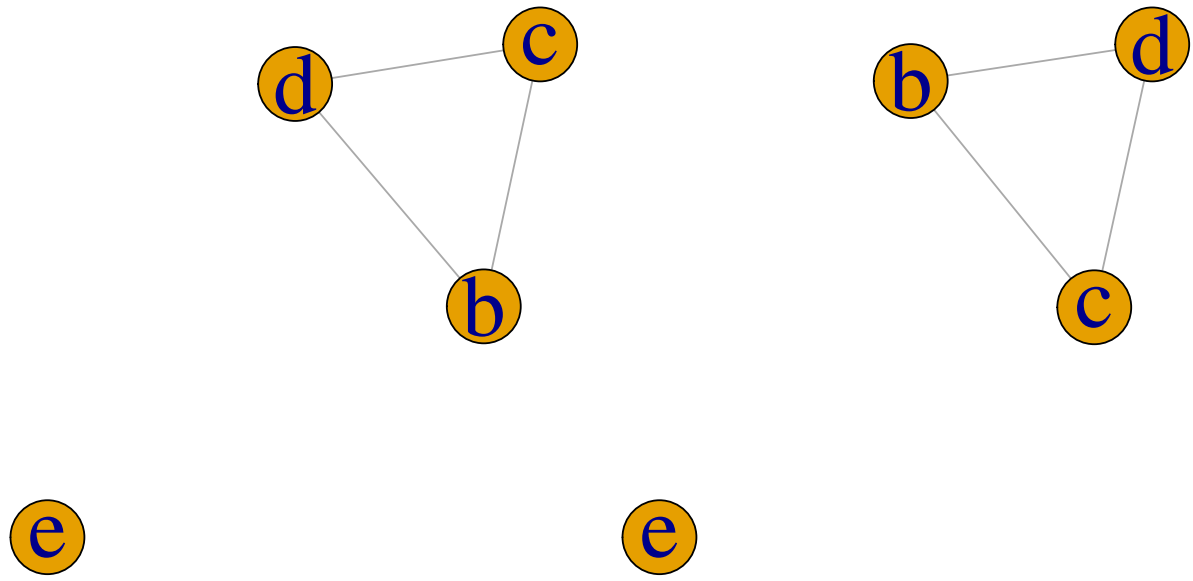
This shows that $\{b, c\}$ separates $\{a\}$ and $\{d\}$.

The graph $\mathcal{G}_0 = (V_0, E_0)$ is said to be a *subgraph* of $\mathcal{G} = (V, E)$ if $V_0 \subseteq V$ and $E_0 \subseteq E$. For $A \subseteq V$, let E_A denote the set of edges in E between vertices in A. Then $\mathcal{G}_A = (A, E_A)$ is the *subgraph induced by* A. For example

```
ug1 <- gRbase::subGraph(c("b", "c", "d", "e"), ug0)
```

```
ug12 <- igraph::subgraph(ug0, c("b", "c", "d", "e"))
```

```
par(mfrow=c(1,2), mar=c(0,0,0,0))
myplot(ug1); myplot(ug12)
```



The *boundary* $\text{bd}(\alpha) = \text{adj}(\alpha)$ is the set of vertices adjacent to α and for undirected graphs the boundary is equal to the set of *neighbours* $\text{ne}(\alpha)$. The *closure* $\text{cl}(\alpha)$ is $\text{bd}(\alpha) \cup \{\alpha\}$.

```
gRbase::adj(ug0, "c")

## $c
## [1] "b" "d"

gRbase::closure("c", ug0)

## [1] "c" "b" "d"
```

1.2.2 Directed Acyclic Graphs

A *directed graph* as a mathematical object is a pair $\mathcal{G} = (V, E)$ where V is a set of vertices and E is a set of directed edges, normally drawn as arrows. A directed graph is *acyclic* if it has no directed cycles, that is, cycles with the arrows pointing in the same direction all the way around. A *DAG* is a directed graph that is acyclic.

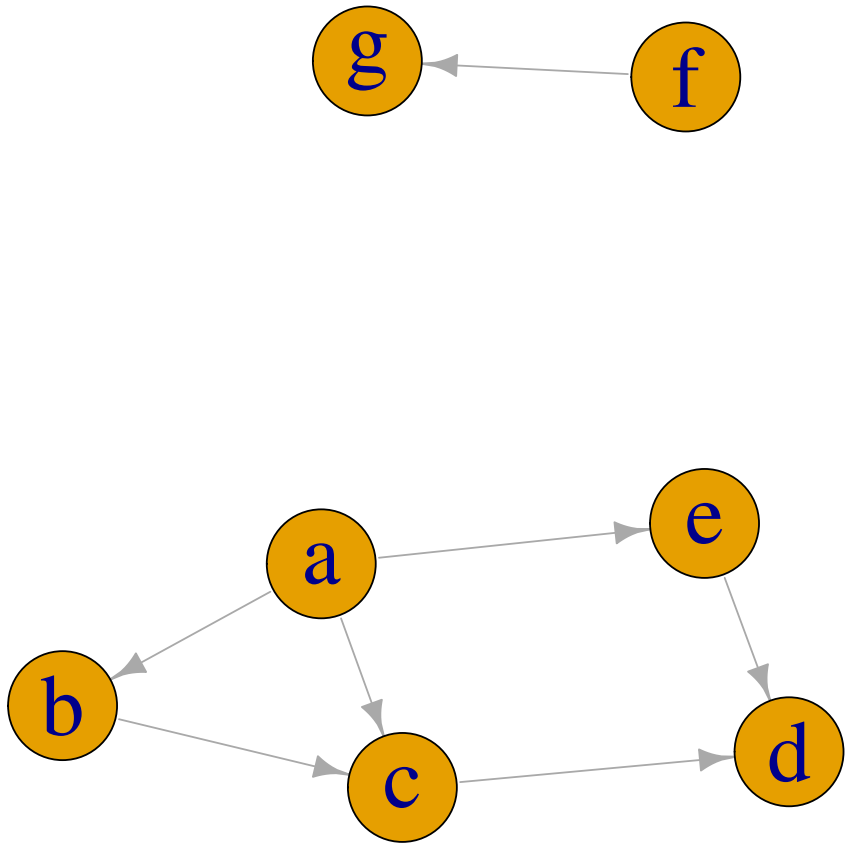
A DAG may be created using the *dag()* function. The graph can be specified by a list of formulas or by a list of vectors. The following statements are equivalent:

```
dag0 <- gRbase::dag(~a, ~b*a,  ~c*a*b, ~d*c*e, ~e*a, ~g*f)
dag0 <- gRbase::dag(~a + b*a + c*a*b + d*c*e + e*a + g*f)
dag0 <- gRbase::dag(~a + b|a + c|a*b + d|c*e + e|a + g|f)
dag0 <- gRbase::dag("a", c("b", "a"), c("c", "a", "b"), c("d", "c", "e"),
            c("e", "a"), c("g", "f"))
dag0

## IGRAPH afaf71d DN-- 7 7 --
## + attr: name (v/c)
## + edges from afaf71d (vertex names):
## [1] a->b a->c b->c c->d e->d a->e f->g
```

Note that `~a` means that `"a"` has no parents while `~d*b*c` means that `"d"` has parents `"b"` and `"c"`. Instead of "`*`", a "`:`" can be used in the specification. If the specified graph contains cycles then `dag()` returns `NULL`.

Per default the *dag()* function returns an `igraph` object, but the option `result="matrix"` leads it to return an adjacency matrix instead.

```
myplot(dag0)
```



The nodes and edges of a DAG can be retrieved with the *nodes()* and *edges()* functions.

```
gRbase::nodes(dag0)

## [1] "a" "b" "c" "d" "e" "f" "g"

gRbase::edges(dag0) |> str()

## List of 7
```

```
##  $ a: chr [1:3] "b" "c" "e"
##  $ b: chr "c"
##  $ c: chr "d"
##  $ d: chr(0)
##  $ e: chr "d"
##  $ f: chr "g"
##  $ g: chr(0)
```

Thus *edges()* gives the children of each node. Alternatively a list of (ordered) pairs can be optained with *edgeList()*

```
edgeList(dag0) |> str()

## List of 7
##  $ : chr [1:2] "a" "b"
##  $ : chr [1:2] "a" "c"
##  $ : chr [1:2] "a" "e"
##  $ : chr [1:2] "b" "c"
##  $ : chr [1:2] "c" "d"
##  $ : chr [1:2] "e" "d"
##  $ : chr [1:2] "f" "g"
```

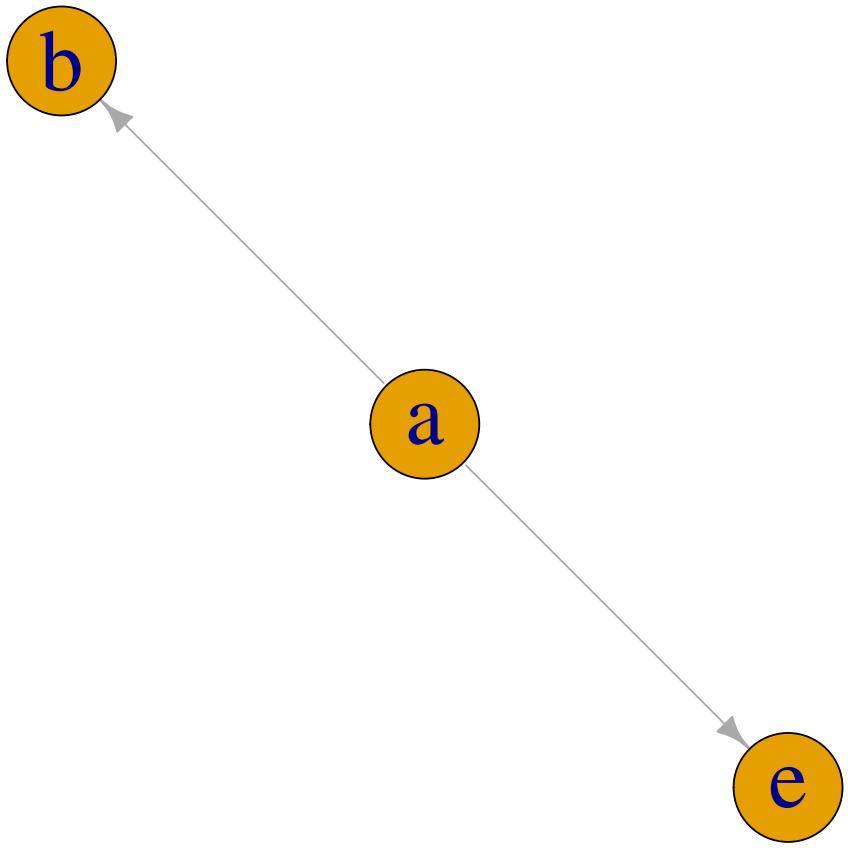
The *vpar()* function returns a list, with an element for each node together with its parents:

```
vpardag0 <- gRbase::vpar(dag0)
vpardag0 |> str()

## List of 7
##  $ a: chr "a"
##  $ b: chr [1:2] "b" "a"
##  $ c: chr [1:3] "c" "a" "b"
##  $ d: chr [1:3] "d" "c" "e"
##  $ e: chr [1:2] "e" "a"
##  $ f: chr "f"
##  $ g: chr [1:2] "g" "f"

vpardag0$c

## [1] "c" "a" "b"
```

A *path* (of length n) from α to β is a sequence of vertices $\alpha = \alpha_0, \ldots, \alpha_n = \beta$ such that $\alpha_{i-1} \to \alpha_i$ is an edge in the graph. If there is a path from α to β we write $\alpha \mapsto \beta$. The

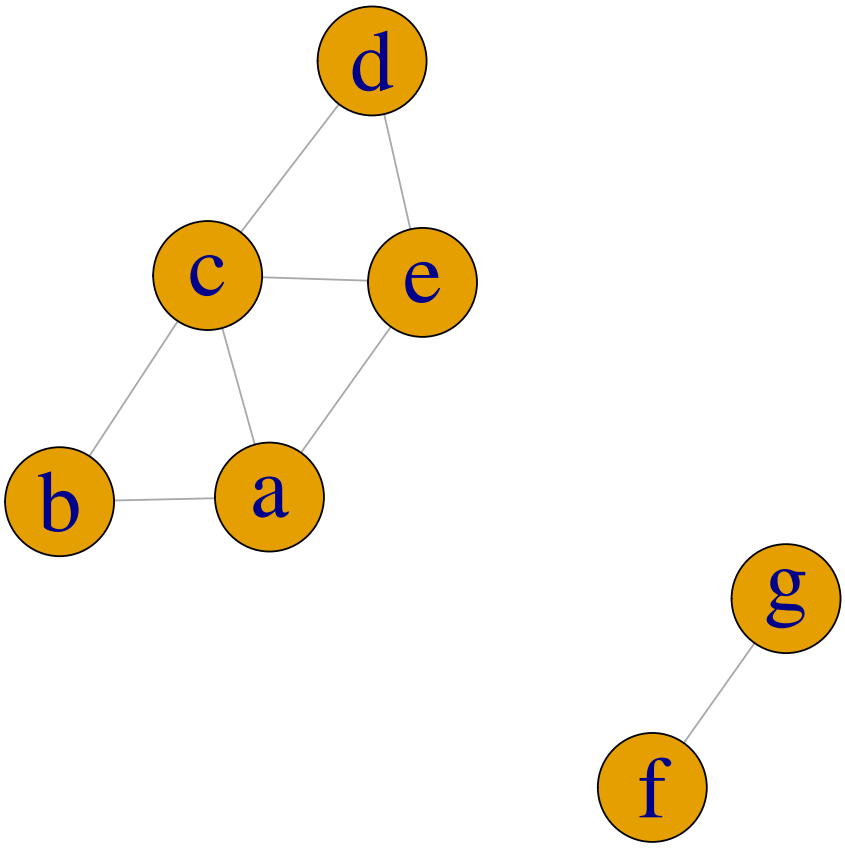
parents $\mathrm{pa}(\beta)$ of a node β are those nodes α for which $\alpha \to \beta$. The *children* $\mathrm{ch}(\alpha)$ of a node α are those nodes β for which $\alpha \to \beta$. The *ancestors* $\mathrm{an}(\beta)$ of a node β are the nodes α such that $\alpha \mapsto \beta$. The *ancestral set* $\mathrm{an}(A)$ of a set A is the union of A with its ancestors. The *ancestral graph* of a set A is the subgraph induced by the ancestral set of A.

```
gRbase::parents("d", dag0)

## [1] "c" "e"

gRbase::children("c", dag0)

## [1] "d"

gRbase::ancestralSet(c("b", "e"), dag0)

## [1] "a" "b" "e"

ag <- gRbase::ancestralGraph(c("b", "e"), dag0)
myplot(ag)
```



An important operation on DAGs is to (i) add edges between the parents of each node, and then (ii) replace all directed edges with undirected ones, thus returning an undirected graph. This operation is used in connection with independence interpretations of the DAG, see Sect. 1.3, and is known as *moralization*. This is implemented by the *moralize()* function:

```
dag0m <- gRbase::moralize(dag0)
myplot(dag0m)
```



1.2.3 Mixed Graphs

Although the primary focus of this book is on undirected graphs and DAGs, it is also useful to consider *mixed graphs*. These are graphs with at least two types of edges, for example directed and undirected, or directed and bidirected.

A sequence of vertices $v_1, v_2, \ldots v_k, v_{k+1}$ is called a *path* if for each $i = 1 \ldots k$, either $v_i - v_{i+1}$, $v_i \leftrightarrow v_{i+1}$ or $v_i \rightarrow v_{i+1}$. If $v_i - v_{i+1}$ for each i the path is called *undirected*, if $v_i \rightarrow v_{i+1}$ for each i it is called *directed*, and if $v_i \rightarrow v_{i+1}$ for at least one i it is called *semi-directed*. If $v_i = v_{k+1}$ it is called a *cycle*.

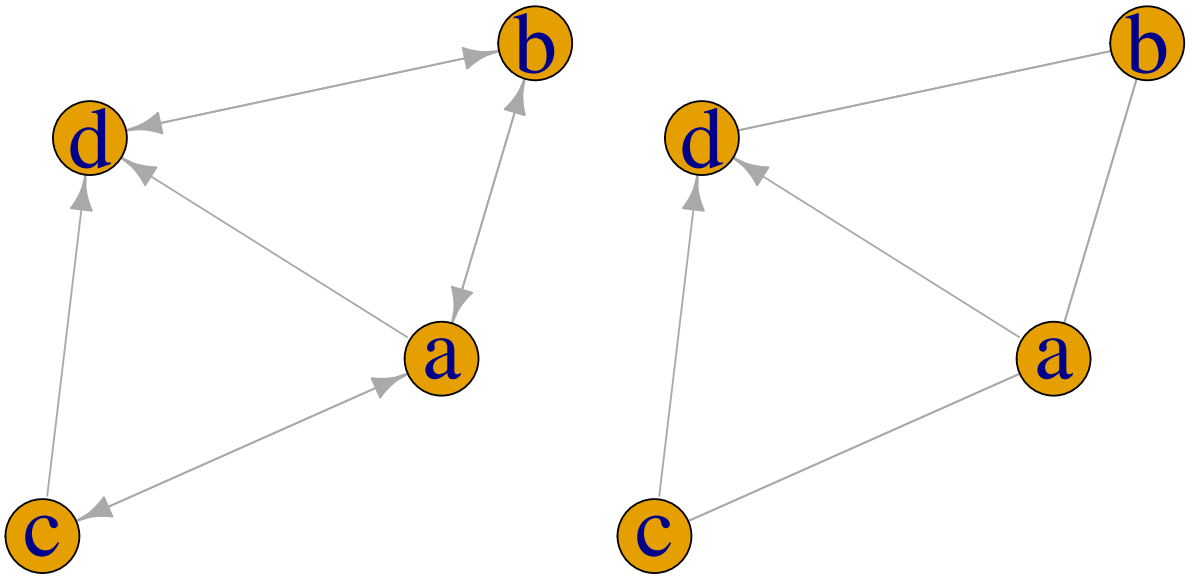
Mixed graphs are represented in the **igraph** package as directed graphs with multiple edges. In this sense they are not simple. A convenient way of defining them (in lieu of model formulae) is to use adjacency matrices. We can construct such a matrix as follows:

```
adjm <- matrix(c(0, 1, 1, 1,
                 1, 0, 0, 1,
                 1, 0, 0, 1,
                 0, 1, 0, 0), byrow=TRUE, nrow=4)
rownames(adjm) <- colnames(adjm) <- letters[1:4]
adjm

##   a b c d
## a 0 1 1 1
## b 1 0 0 1
## c 1 0 0 1
## d 0 1 0 0
```

Note that `igraph` interprets symmetric entries as double-headed arrows and thus does not distinguish between bidirected and undirected edges. However we can persuade `igraph` to display undirected instead of bidirected edges:

```
gG1 <- gG2 <- as(adjm, "igraph")
lay <- layout.fruchterman.reingold(gG1)
E(gG2)$arrow.mode <- c(2,0)[1+is.mutual(gG2)]
```

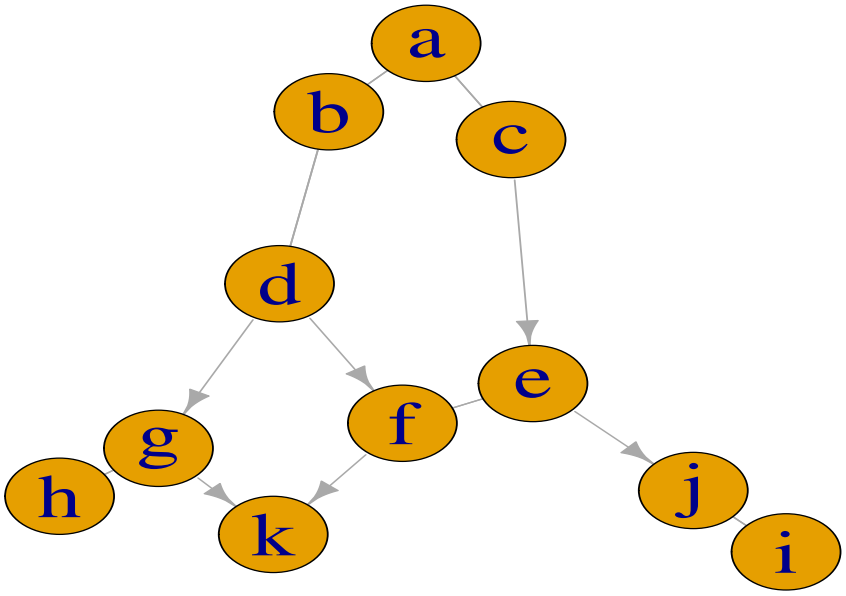
```
par(mfrow=c(1,2), mar=c(0,0,0,0))
myplot(gG1, layout=lay); myplot(gG2, layout=lay)
```



A *chain graph* is a mixed graph with no bidirected edges and no semi-directed cycles. Such graphs form a natural generalisation of undirected graphs and DAGs, as we shall see later. The following example is from Frydenberg [1990]:

```
d1 <- matrix(0, 11, 11)
d1[1,2] <- d1[2,1] <- d1[1,3] <- d1[3,1] <- d1[2,4] <- d1[4,2] <-
  d1[5,6] <- d1[6,5] <- 1
d1[9,10] <- d1[10,9] <- d1[7,8] <- d1[8,7] <- d1[3,5] <-
  d1[5,10] <- d1[4,6] <- d1[4,7] <- 1
d1[6,11] <- d1[7,11] <- 1
```

```
rownames(d1) <- colnames(d1) <- letters[1:11]
cG1 <- as(d1, "igraph")
E(cG1)$arrow.mode <- c(2,0)[1+is.mutual(cG1)]
myplot(cG1, layout=layout.fruchterman.reingold)
```



The *components* of a chain graph $\mathcal{G}$ are the connected components of the graph formed after removing all directed edges from $\mathcal{G}$. All edges within a component are undirected, and all edges between components are directed. Also, all arrows between any two components have the same direction. The graph constructed by identifying its nodes with the components of $\mathcal{G}$, and joining two nodes with an arrow whenever there is an arrow between the corresponding components in $\mathcal{G}$, is a DAG, the so-called *component DAG* of $\mathcal{G}$, written $\mathcal{G}_C$.

The *anterior set* of a vertex set $S \subseteq V$ is defined in terms of the component DAG. Write the set of components of $\mathcal{G}$ containing S as S_c. Then the anterior set of S in $\mathcal{G}$ is defined as the union of the components in the ancestral set of S_c in $\mathcal{G}_C$. The *anterior graph* of $S \subseteq V$ is the subgraph of $\mathcal{G}$ induced by the anterior set of S.

The *moralization* operation is also important for chain graphs. Similar to DAGs, unmarried parents of the same chain components are joined and directions are then removed.

1.3 Conditional Independence and Graphs

The concept of statistical independence is presumably familiar to all readers but that of *conditional independence* may be less so. Suppose that we have a collection of random variables $(X_v)_{v\in V}$ with a joint density. Let A, B and C be subsets of V and let $X_A = (X_v)_{v\in A}$ and similarly for X_B and X_C. Then the statement that X_A and X_B are conditionally independent given X_C, written $A \perp\!\!\!\perp B \,|\, C$, means that for each possible value of x_C of X_C, X_A and X_B are independent in the conditional distribution given $X_C = x_c$. So if we write $f()$ for a generic density or probability mass function, then one characterization of $A \perp\!\!\!\perp B \,|\, C$ is that

$$f(x_A, x_B \,|\, x_C) = f(x_A \,|\, x_C) f(x_B \,|\, x_C).$$

An equivalent characterization [Dawid, 1998] is that the joint density of (X_A, X_B, X_C) factorizes as

$$f(x_A, x_B, x_C) = g(x_A, x_C) h(x_B, x_C), \tag{1.1}$$

that is, as a product of two functions $g()$ and $h()$, where $g()$ does not depend on x_B and $h()$ does not depend on x_A. This is known as the *factorization criterion*.

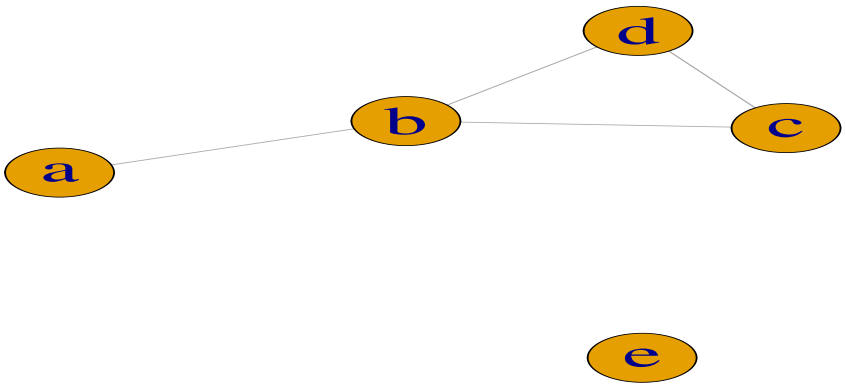
Parametric models for $(X_v)_{v\in V}$ may be thought of as specifying a set of joint densities (one for each admissible set of parameters). These may admit factorisations of the form just described, giving rise to conditional independence relations between the variables. Some models give rise to patterns of conditional independences that can be represented as an undirected graph. More specifically, let $\mathcal{G} = (V, E)$ be an undirected graph with cliques $C_1, \ldots C_k$. Consider a joint density $f()$ of the variables in V. If this admits a factorization of the form

$$f(x_V) = \prod_{i=1}^{k} g_i(x_{C_i})$$

for some functions $g_1() \ldots g_k()$ where $g_j()$ depends on x only through x_{C_j} then we say that $f()$ factorizes according to $\mathcal{G}$.

If all the densities under a model factorize according to $\mathcal{G}$, then $\mathcal{G}$ encodes the conditional independence structure of the model, through the following result (the *global Markov property*): whenever sets A and B are separated by a set C in the graph, then $A \perp\!\!\!\perp B \,|\, C$ under the model. Thus for example

```
myplot(ug0)
```




```
gRbase::separates("a", "d", "b", ug0)

## [1] TRUE
```

shows that under a model with this dependence graph, $a \perp\!\!\!\perp d \,|\, b$.

If we want to find out whether two variable sets are marginally independent, we ask whether they are separated by the empty set, which we specify using a character vector of length zero:

```
gRbase::separates("a", "d", character(0), ug0)

## [1] FALSE
```

Model families that admit suitable factorizations are described in later chapters in this book. These include: log-linear models for multivariate discrete data, graphical Gaussian models for multivariate Gaussian data, and mixed interaction models for mixed discrete and continuous data.

Other models give rise to patterns of conditional independences that can be represented by DAGs. These are models for which the variable set V may be ordered in such way that the joint density factorizes as follows

$$f(x_V) = \prod_{v \in V} f(x_v \,|\, x_{\mathrm{pa}(v)}) \tag{1.2}$$

for some variable sets $\{\mathrm{pa}(v)\}_{v \in V}$ such that the variables in $\mathrm{pa}(v)$ precede v in the ordering. Again the vertices of the graph represent the random variables, and we can identify the sets $\mathrm{pa}(v)$ with the parents of v in the DAG.

With DAGs, conditional independence is represented by a property called *d-separation*. That is, whenever sets A and B are d-separated by a set C in the graph, then $A \perp\!\!\!\perp B \,|\, C$ under the model. The notion of d-separation can be defined in various ways, but one characterisation is as follows: A and B are d-separated by a set C if and only if they are separated in the graph formed by moralizing the anterior graph of $A \cup B \cup C$.

So we can easily define a function to test this:

```
d_separates <- function(a, b, c, dag_) {
    ##ag <- ancestralGraph(union(union(a, b), c), dag_)
    ag <- ancestralGraph(c(a, b, c), dag_)
    separates(a, b, c, moralize(ag))
}
d_separates("c", "e", "a", dag0)

## [1] TRUE
```

So under `dag0` it holds that $c \perp\!\!\!\perp e \,|\, a$.

Still other models correspond to patterns of conditional independences that can be represented by a chain graph $\mathcal{G}$. There are several ways to relate Markov properties to chain graphs. Here we describe the so-called LWF Markov properties, associated with Lauritzen, Wermuth and Frydenberg.

For these there are two levels to the factorization requirements. Firstly, the joint density needs to factorize in a way similar to a DAG, i.e.

$$f(x_V) = \prod_{C \in \mathcal{C}} f(x_C \,|\, x_{\mathrm{pa}(C)})$$

where $\mathcal{C}$ is the set of components of $\mathcal{G}$. In addition, each conditional density $f(x_C \,|\, x_{\mathrm{pa}(C)})$ must factorize according to an undirected graph constructed in the following way. First form the subgraph of $\mathcal{G}$ induced by $C \cup \mathrm{pa}(C)$, drop directions, and then complete $\mathrm{pa}(C)$ (that is, add edges between all vertices in $\mathrm{pa}(C)$)).

For densities which factorize as above, conditional independence is related to a property called *c-separation*: that is, $A \perp\!\!\!\perp B \,|\, C$ whenever sets A and B are c-separated by C in the graph. The notion of c-separation in chain graphs is similar to that of d-separation in DAGs. A and B are c-separated by a set C if and only if they are separated in the graph formed by moralizing the anterior graph of $A \cup B \cup C$.

1.4 More About Graphs

1.4.1 Special Properties

A node in an undirected graph is *simplicial* if its boundary is complete.

```
gRbase::is.simplicial("b", ug0)

## [1] FALSE

gRbase::simplicialNodes(ug0)

## [1] "a" "c" "d" "e"
```

To obtain the *connected components* of a graph:

```
gRbase::connComp(ug0) |> str()

## List of 2
##  $ : chr [1:4] "a" "b" "c" "d"
##  $ : chr "e"

## Using igraph
igraph::components(ug0) |> str()

## List of 3
##  $ membership: Named num [1:5] 1 1 1 1 2
##   ..- attr(*, "names")= chr [1:5] "a" "b" "c" "d" ...
##  $ csize     : num [1:2] 4 1
##  $ no        : int 2
```

If a cycle $\alpha = \alpha_0, \alpha_1, \ldots, \alpha_n = \alpha$ has adjacent elements $\alpha_i \sim \alpha_j$ with $j \notin \{i-1, i+1\}$ then it is said to have a *chord*. If it has no chords it is said to be *chordless*. A graph with no chordless cycles of length ≥ 4 is called *triangulated* or *chordal*:

```
gRbase::is.triangulated(ug0)

## [1] TRUE
```

```
igraph::is_chordal(ug0)

## $chordal
## [1] TRUE
##
## $fillin
## NULL
##
## $newgraph
## NULL
```

Triangulated graphs are of special interest for graphical models as they admit closed-form maximum likelihood estimates and allow considerable computational simplification by decomposition.

A triple (A, B, D) of non–empty disjoint subsets of V is said to *decompose* $\mathcal{G}$ into $\mathcal{G}_{A\cup D}$ and $\mathcal{G}_{B\cup D}$ if $V = A \cup B \cup D$ where D is complete and separates A and B.

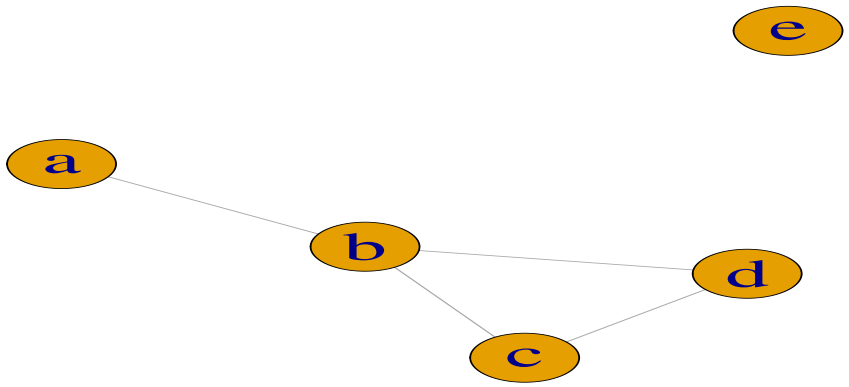
```
gRbase::is.decomposition("a", "d", c("b", "c"), ug0)

## [1] FALSE
```

Note that although $\{d\}$ is complete and separates $\{a\}$ and $\{b, c\}$ in `ug0`, the condition fails because $V \neq \{a, b, c, d\}$.

A graph is *decomposable* if it is complete or if it can be decomposed into decomposable subgraphs. A graph is decomposable if and only if it is triangulated.

An ordering of the nodes in a graph is called a *perfect ordering* if $\mathrm{bd}(i) \cap \{1, \ldots, i-1\}$ is complete for all i. Such an ordering exists if and only if the graph is triangulated. If the graph is triangulated, then a perfect ordering can be obtained with the *maximum cardinality search* (or mcs) algorithm. The *mcs()* function will produce such an ordering if the graph is triangulated; otherwise it will return `NULL`.

```
myplot(ug0)
```




```
gRbase::mcs(ug0)

## [1] "a" "b" "c" "d" "e"
```

```
igraph::max_cardinality(ug0)

## $alpha
## [1] 5 4 2 3 1
##
## $alpham1
## + 5/5 vertices, named, from 3c38e2b:
## [1] e c d b a

igraph::max_cardinality(ug0)$alpham1 |> attr("names")

## [1] "e" "c" "d" "b" "a"
```

Sometimes it is convenient to have some control over the ordering given to the variables:

```
gRbase::mcs(ug0, root=c("d", "c", "a"))

## [1] "d" "c" "b" "a" "e"
```

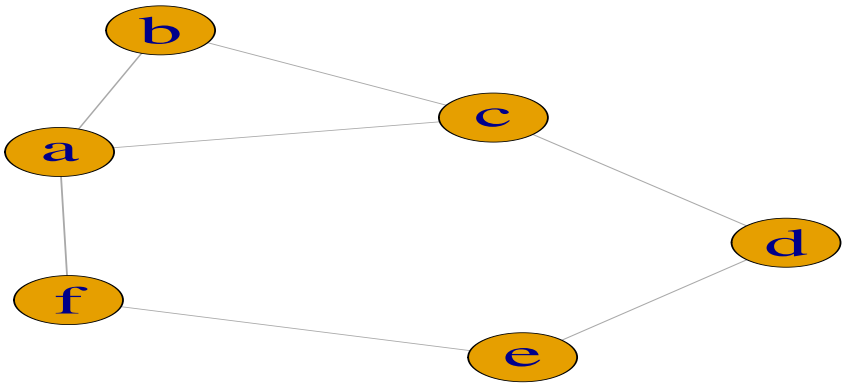
Here *mcs()* tries to follow the ordering given and succeeds for the first two variables but then fails afterwards.

The cliques of a triangulated undirected graph can be ordered as $(C_1, \ldots, C_Q)$ to have the *running intersection property* (also called a *RIP ordering*). The running intersection property is that $C_j \cap (C_1 \cup \ldots \cup C_{j-1}) \subset C_i$ for some $i < j$ for $j = 2, \ldots, Q$. We define the sets $S_j = C_j \cap (C_1 \cup \ldots \cup C_{j-1})$ and $R_j = C_j \setminus S_j$ with $S_1 = \emptyset$. The sets S_j are called *separators* as they separate R_j from $(C_1 \cup \ldots \cup C_{j-1}) \setminus S_j$. Any clique C_i where $S_j \subset C_i$ with $i < j$ is a possible parent of C_i. The *rip()* function returns such an ordering if the graph is triangulated (otherwise, it returns `list()`):

```
gRbase::rip(ug0)

## cliques
##   1 : a b
##   2 : b c d
##   3 : e
## separators
##   1 :
##   2 : b
##   3 :
## parents
##   1 : 0
##   2 : 1
##   3 : 0
```

If a graph is not triangulated it can be made so by adding extra edges, so called *fill-ins*, using *triangulate()*:

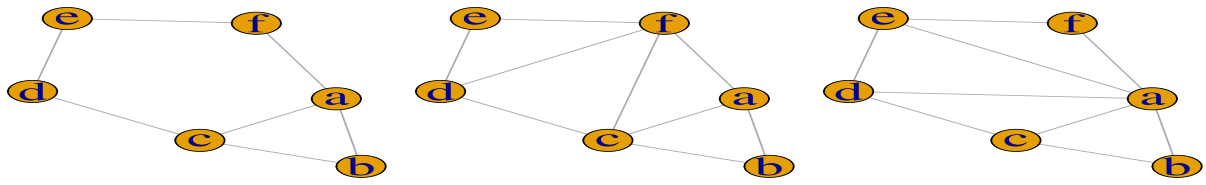
```
ug2 <- gRbase::ug(~a:b:c + c:d + d:e + a:e)
ug2 <- gRbase::ug(~a:b:c + c:d + d:e + e:f + a:f)

gRbase::is.triangulated(ug2)

## [1] FALSE

igraph::is_chordal(ug2)  |> str()

## List of 3
##  $ chordal : logi FALSE
##  $ fillin  : NULL
##  $ newgraph: NULL

myplot(ug2)
```




```
ug3 <- gRbase::triangulate(ug2)
gRbase::is.triangulated(ug3)

## [1] TRUE
```

```
zzz <- igraph::is_chordal(ug2, fillin=TRUE, newgraph=TRUE)
V(ug2)[zzz$fillin]

## + 4/6 vertices, named, from 3a6c240:
## [1] d a e a

ug32 <- zzz$newgraph
```

```
par(mfrow=c(1,3), mar=c(0,0,0,0))
lay <- layout.fruchterman.reingold(ug2)
myplot(ug2, layout=lay);
myplot(ug3, layout=lay);
myplot(ug32, layout=lay)
```



The *Markov blanket* of a vertex v in a DAG may be defined as the minimal set that d-separates v from the remaining variables. It is easily derived as the set of neighbours to v in the moral graph of $\mathcal{G}$. For example, the Markov blanket of vertex `e` in `dag0` is

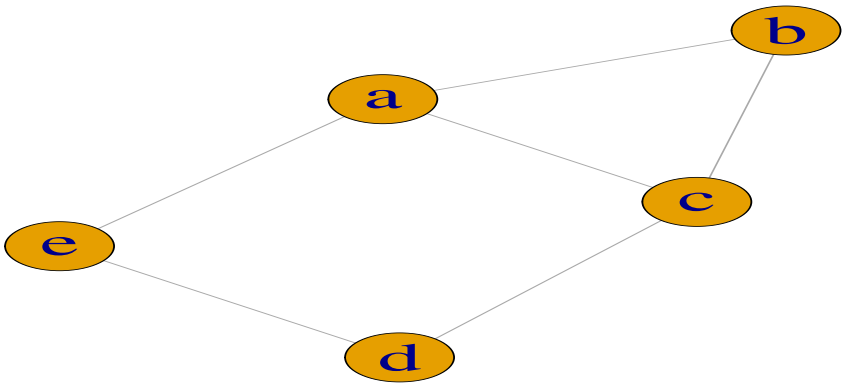
```
adj(moralize(dag0), "e")
```

It is easily seen that the Markov blanket of v is the union of v's parents, v's children, and the parents of v's children.

1.4.2 The igraph package

It is possible to create `igraph` objects using the *graph.formula()* function:

```
ug4 <- graph.formula(a -- b:c, c--b:d, e -- a:d)
ug4

## IGRAPH 34035f5 UN-- 5 6 --
## + attr: name (v/c)
## + edges from 34035f5 (vertex names):
## [1] a--b a--c a--e b--c c--d d--e

myplot(ug4)
```



The same graph may be created from scratch as follows:

```
ug4.2 <- graph.empty(n=5, directed=FALSE)
V(ug4.2)$name <- V(ug4.2)$label <- letters[1:5]
ug4.2 <- add.edges(ug4.2, 1+c(0,1, 0,2, 0,4, 1,2, 2,3, 3,4))
ug4.2

## IGRAPH 2c58479 UN-- 5 6 --
## + attr: label (v/c), name (v/c)
## + edges from 2c58479 (vertex names):
## [1] a--b a--c a--e b--c c--d d--e
```

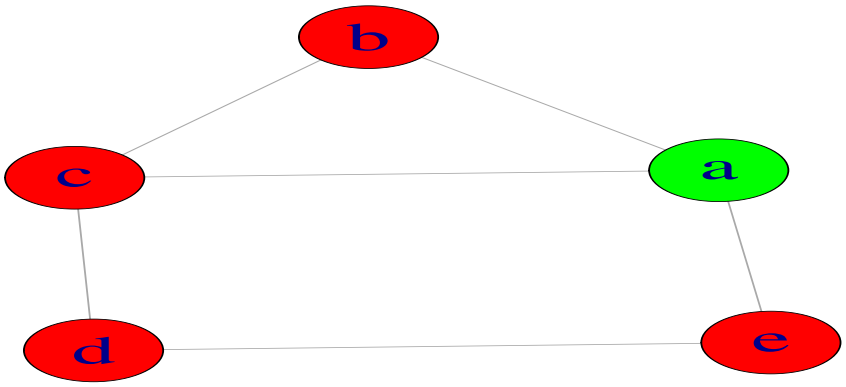
The graph is displayed using the *plot()* function, with a layout determined using the `graphopt` method. A variety of layout algorithms are available: type `?layout` for an overview. Note that per default the nodes are labelled $0, 1, \ldots$ and so forth. We show how to modify this shortly.

As mentioned previously we have created a custom function *myplot()* which creates somewhat more readable plots:

```
myplot(ug4, layout=layout.graphopt)
```

Objects in **igraph** graphs are defined in terms of node and edge lists. In addition, they have *attributes*: these belong to the vertices, the edges or to the graph itself. The following example sets a graph attribute, *layout*, and two vertex attributes, *label* and *color*. These are used when the graph is plotted. The *name* attribute contains the node labels.

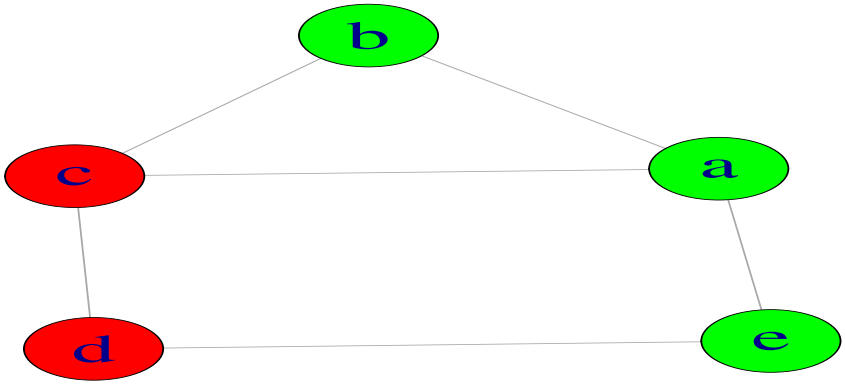
```
ug4$layout   <- layout.graphopt(ug4)
V(ug4)$label <- V(ug4)$name
V(ug4)$color <- "red"
V(ug4)[1]$color <- "green"
V(ug4)$size <- 40
V(ug4)$label.cex <- 3
plot(ug4)
```



Note the use of array indices to access the attributes of the individual vertices. Currently, the indices are zero-based, so that `V(ug4)[1]` refers to the second node (B). (This may change). Edges attributes are accessed similarly, using a container structure `E(ug4)`: also here the indices are zero-based (currently).

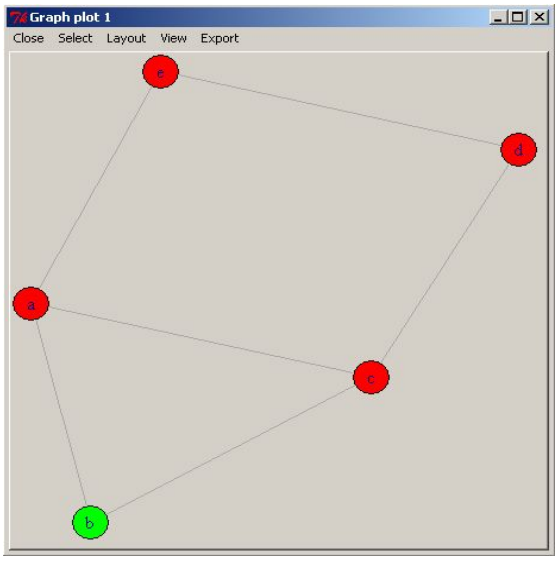
It is easy to extend `igraph` objects by defining new attributes. In the following example we define a new vertex attribute, `discrete`, and use this to color the vertices.

```
ug5 <- set.vertex.attribute(ug4, "discrete", value=c(T, T, F, F, T))
V(ug5)[discrete]$color <- "green"
V(ug5)[!discrete]$color <- "red"
plot(ug5)
```



A useful interactive drawing facility is provided with the *tkplot()* function. This causes a pop-up window to appear in which the graph can be manually edited. One use of this is to edit the layout of the graph: the new coordinates can be extracted and re-used by the *plot()* function. For example

```
> tkplot(ug4)
2
```



The *tkplot()* function returns a window id (here 2). While the popup window is open, the current layout can be obtained by passing the window id to the *tkplot.getcoords()* function, as for example

```
xy <- tkplot.getcoords(2)
plot(g, layout=xy)
```

It is straightforward to reuse layout information with `igraph` objects. The layout functions when applied to graphs return a matrix of (x, y) coordinates:

```
layout.fruchterman.reingold(ug4)

##       [,1]    [,2]
## [1,] 2.314  1.1431
## [2,] 3.467  1.1399
## [3,] 2.808  0.1934
## [4,] 1.695 -0.4303
## [5,] 1.164  0.5885
```

Most layout algorithms use a random generator to choose an initial configuration. Hence if we set the layout attribute to be a layout function, repeated calls to plot will use different layouts. For example, after

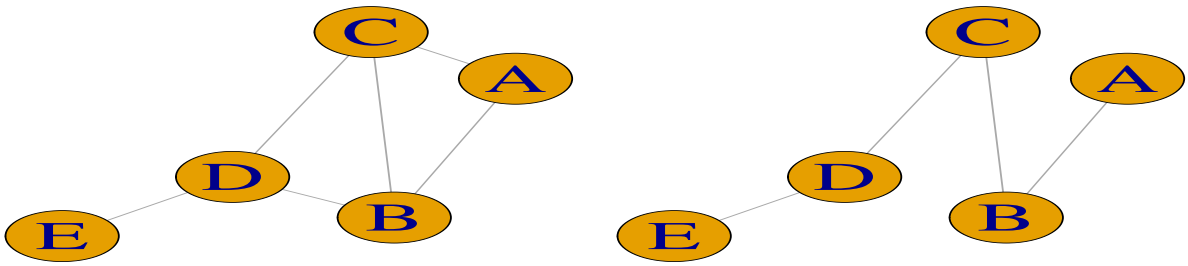
```
ug4$layout <- layout.fruchterman.reingold
```

repeated invocations of `plot(ug4)` will use different layouts. In contrast, after

```
ug4$layout <- layout.fruchterman.reingold(ug4)
```

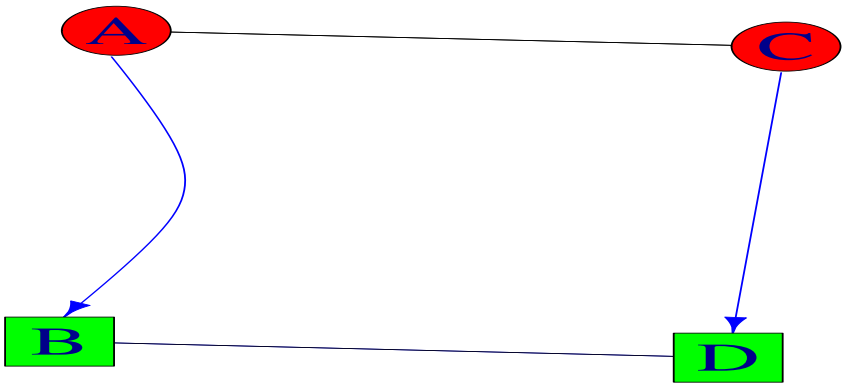
the layout will be fixed. The following code fragment illustrates how two graphs with the same vertex set may be plotted using the same layout.

```
ug5 <- gRbase::ug(~A*B*C + B*C*D + D*E)
ug6 <- gRbase::ug(~A*B + B*C + C*D + D*E)
lay.fr <- layout.fruchterman.reingold(ug5)
ug6$layout       <- ug5$layout       <- lay.fr
V(ug5)$size      <- V(ug6)$size      <- 50
V(ug5)$label.cex <- V(ug6)$label.cex <- 3
par(mfrow=c(1,2), mar=c(0,0,0,0))
plot(ug5); plot(ug6)
```



An overview of attributes used in plotting can be obtained by typing `?igraph.plotting`. A final example illustrates how more complex graphs can be displayed:

```
em1 <- matrix(c(0, 1, 1, 0,
                0, 0, 0, 1,
                1, 0, 0, 1,
                0, 1, 0, 0), nrow=4, byrow=TRUE)
iG  <- graph.adjacency(em1)
V(iG)$shape <- c("circle", "square", "circle", "square")
V(iG)$color <- rep(c("red", "green"), 2)
V(iG)$label <- c("A", "B", "C", "D")
E(iG)$arrow.mode <- c(2,0)[1 + is.mutual(iG)]
E(iG)$color  <- rep(c("blue", "black"), 3)
E(iG)$curved <- c(T, F, F, F, F, F)
iG$layout    <- layout.graphopt(iG)
myplot(iG)
```



1.4.3 Operations on Graphs in Different Representations

The **gRbase** package has a function *querygraph()* which provides a common interface to the graph operations for undirected graphs and DAGs illustrated above. Moreover, *querygraph()* works on graphs represented as `igraph` objects and adjacency matrices. The general syntax is

```
args(querygraph)

## function (object, op, set = NULL, set2 = NULL, set3 = NULL)
## NULL
```

For example, we obtain:

```
ug_ <- gRbase::ug(~a:b + b:c:d + e)
gRbase::separates("a", "d", c("b", "c"), ug_)

## [1] TRUE

gRbase::querygraph(ug_, "separates", "a", "d", c("b", "c"))

## [1] TRUE

gRbase::qgraph(ug_, "separates", "a", "d", c("b", "c"))

## [1] TRUE
```

Bibliography

Søren Højsgaard, David Edwards, and Steffen Lauritzen. *Graphical models with R*. Springer, 2012.

M. Frydenberg. The chain graph Markov property. *Scandinavian Journal of Statistics*, 17: 333–353, 1990.

A. P. Dawid. Conditional independence. In Samuel Kotz, Campbell B. Read, and David L. Banks, editors, *Encyclopedia of Statistical Sciences, Update Volume 2*, pages 146–155. Wiley-Interscience, New York, 1998.

Index