Graphs in the **gRbase** package

Søren Højsgaard

gRbase version 1.8-6.6 as of 2020-06-14

```
## Loading required package:  BiocGenerics
## Loading required package:  parallel
##
## Attaching package:  BiocGenerics
## The following objects are masked from package:parallel:
##
##    clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##    clusterExport, clusterMap, parApply, parCapply, parLapply,
##    parLapplyLB, parRapply, parSapply, parSapplyLB
## The following objects are masked from package:stats:
##
##    IQR, mad, sd, var, xtabs
## The following objects are masked from package:base:
##
##    Filter, Find, Map, Position, Reduce, anyDuplicated, append,
##    as.data.frame, basename, cbind, colnames, dirname, do.call,
##    duplicated, eval, evalq, get, grep, grepl, intersect, is.unsorted,
##    lapply, mapply, match, mget, order, paste, pmax, pmax.int, pmin,
##    pmin.int, rank, rbind, rownames, sapply, setdiff, sort, table,
##    tapply, union, unique, unsplit, which, which.max, which.min
## Loading required package:  grid
```

Contents

1 Introduction

The packages **graph**, **RBGL**, **Rgraphviz** and **igraph** are extremely useful tools for graph operations, manipulation and layout. The **gRbase** package adds some additional tools to these fine packages. The most important tools are:

1. Undirected and directed acyclic graphs can be specified using formulae or an adjacency list using the functions `ug()` and `dag()`.

   This gives graphs represented in one of the following forms:[1]

   - A `graphNEL` object (the default),
   - A dense adjacency matrix (a "standard" matrix in `R`).
   - A sparse adjacency matrix(a `dgCMatrix` from the `Matrix` package).

2. Some graph algorithms are implemented in **gRbase**. These can be applied to graphs represented as `graphNEL`s and matrices. The most important algorithms are:

   - `moralize()`, (moralize a directed acyclic graph)
   - `mcs()`,(maximum cardinality search for undirected graph)
   - `triangulate()`, (triangulate undirected graph)
   - `rip()`, (RIP ordering of cliques of triangulated undirected graph)
   - `jTree()` (Create junction tree from triangulated undirected graph).
   - `get_cliques()`, (get the (maximal) cliques of an undirected graph)
   - `minimal_triang()` (minimal triangulation of undirected graph)
   - `mpd()` (maximal prime subgraph decomposition of undirected graph)

   The general scheme is the following: For example, for maximum cardinality search there is a `mcs()` function and a `mcs.default()` method performs maximum cardinality search for graphs represented as `graphNEL`s and as sparse and dense matrices. The workhorse is the function `mcsMAT()` which takes a sparse or a dense matrix as input.

[1] There is a fourth form: **igraph** objects. These, however, will probably not be supported in the future.

2 Graphs

Undirected graphs can be created by the `ug()` function and directed acyclic graphs (DAGs) by the `dag()` function. The graphs can be specified either using formulae or a list of vectors; see examples below.

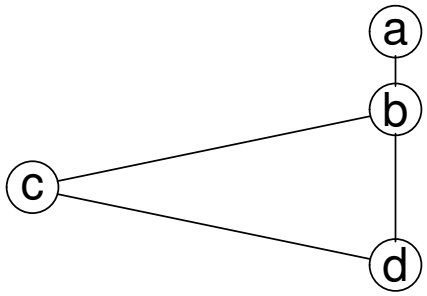
2.1 Undirected graphs

An undirected graph is created by the `ug()` function. The following specifications are equivalent (notice that ":" and "*" can be used interchangably):

```
uG11 <- ug( ~a:b + b:c:d)
uG12 <- ug( c("a", "b"), c("b", "c", "d") )
uG13 <- ug( list( c("a", "b"), c("b", "c", "d") ) )
```

Default is to return a `graphNEL` object (for which there is a plot method):

```
uG11

## A graphNEL graph with undirected edges
## Number of Nodes = 4
## Number of Edges = 4

nodes(uG11)

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

str( edges(uG11) )

## List of 4
##  $ a: chr "b"
##  $ b: chr [1:3] "c" "d" "a"
##  $ c: chr [1:2] "d" "b"
##  $ d: chr [1:2] "b" "c"

plot( uG11 )
```



Matrix representations are obtained with:

```
uG11m <- ug( ~a:b + b:c:d, result="matrix")
uG11M <- ug( ~a:b + b:c:d, result="dgCMatrix")
```

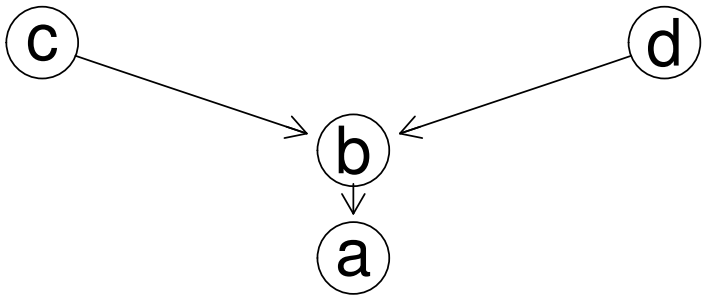
2.2 Directed acyclic graphs (DAGs)

A directed acyclic graph is created by the `dag()` function. The following specifications are equivalent (notice that ":" and "*" can be used interchangably):

```
daG11 <- dag( ~a:b + b:c:d)
daG12 <- dag( c("a", "b"), c("b", "c", "d") )
daG13 <- dag( list( c("a", "b"), c("b", "c", "d") ) )
```

The syntax rules are that `~a` (and `"a"`) means that "a" has no parents while `~c:a:b` (and `c("c","a","b")`) means that "c" has parents "a" and "b".

```
daG11

## A graphNEL graph with directed edges
## Number of Nodes = 4
## Number of Edges = 3

nodes( daG11 )

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

str( edges( daG11 ) )

## List of 4
##  $ a: chr(0)
##  $ b: chr "a"
##  $ c: chr "b"
##  $ d: chr "b"

plot( daG11 )
```



Matrix representations are obtained with:

```
daG11m <- dag( ~a:b + b:c:d, result="matrix")
daG11M <- dag( ~a:b + b:c:d, result="dgCMatrix")
```

2.3 Other types of graphs

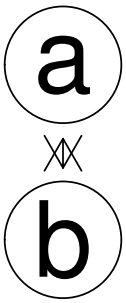
The `dag()` function allows for creation of directed graphs which are not DAGs. Consider

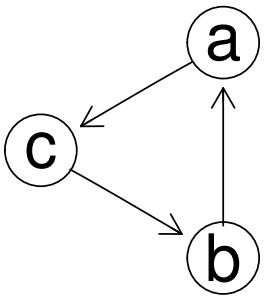
```
d1.bi <- dag(~a:b + b:a)
edgemode( d1.bi )

## [1] "directed"

str( edges(d1.bi) )

## List of 2
##  $ a: chr "b"
##  $ b: chr "a"
```

This graph is not DAG because there is an edge from `a` to `b` and from `b` to `a`; i.e., the edge is bidirected. Likewise we may create:

```
d2.cyc <- dag(~a:b + b:c + c:a)
```

```
par(mfrow=c(1,2)); plot(d1.bi); plot(d2.cyc)
```





Notice: Supplying `dag()` with `forceCheck=TRUE` forces `dag()` to check if the graph is acyclic:

```
dag(~a:b + b:c + c:a, forceCheck=TRUE)
```

2.4 Graph properties

2.5 Adjacency matrices

Graphs in the **graph** package (i.e. `graphNEL` objects) are represented as adjacency lists. However, there is a substantial overhead (in terms of computing time) for such objects. The graph algorithms in **gRbase** are mostly based on a representation as sparse adjacency matrices (which leads to faster code).

A non-zero value at entry (i, j) in an adjacency matrix A for a graph means that there is an edge from i to j. If also (j, i) is non-zero then there is also an edge from j to i. In this case we may think of a bidirected edge between i and j or we may think of the edge as being undirected. We do not distinguish between undirected and bidirected edges in the **gRbase** package. Put differently, in **gRbase**, edges are either directed or undirected/bidirected. In contrast, with `graphNEL` objects one can work with three types of edges: undirected, directed and bidirected edges.

- `is_ug()` checks if the adjacency matrix is symmetric (If applied to a graphNEL, the adjacency matrix is created and checked for symmetry.)
- `is_tug()` checks if the graph is undirected and triangulated (also called chordal) by checking if the adjacency matrix is symmetric and the vertices can be given a perfect ordering using maximum cardinality seach.
- `is_dg()` checks if a graph is directed, i.e., that there are no undirected edges. This is done by computing the elementwise product of A and the transpose of A; if there are no non–zero entries in this product then the graph is directed.
- `is_dag()` will return `TRUE` if all edges are directed and if there are no cycles in the graph. (This is checked by checking if the vertices in the graph can be given a topological ordering which is based on identifying an undirected edge with a bidrected edge).

Notice a special case, namely if the graph has no edges at all (such that the adjacency matrix consists only of zeros). Such a graph is both undirected, triangulated, directed and directed acyclic.

Consider these examples (where `isDirected()` is a method from the **graph** package):

@

```
properties <- function(x){
  c(is_ug=is_ug(x), is_tug=is_tug(x), is_dg=is_dg(x), is_dag=is_dag(x),
    isD=graph::isDirected(x))
}
properties( uG11 )

##  is_ug is_tug  is_dg is_dag    isD
##   TRUE   TRUE  FALSE  FALSE  FALSE

properties( daG11 )

##  is_ug is_tug  is_dg is_dag    isD
##  FALSE  FALSE   TRUE   TRUE   TRUE

properties( d1.bi )

##  is_ug is_tug  is_dg is_dag    isD
##   TRUE   TRUE  FALSE  FALSE   TRUE

properties( d2.cyc )

##  is_ug is_tug  is_dg is_dag    isD
##  FALSE  FALSE   TRUE  FALSE   TRUE
```

2.6 Graph coercion

Graphs can be coerced between differente representations using `as()`; for example:

```
mat <- as(uG11, "matrix")
Mat <- as(mat, "dgCMatrix")
NEL <- as(Mat, "graphNEL")
```

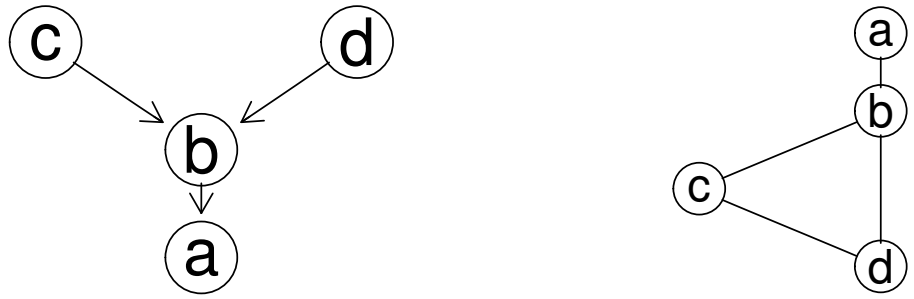
3 Advanced graph operations

3.1 Moralization

A moralized directed acyclic graph is obtained with

```
daG11.mor <- moralize(daG11)
```

```
par(mfrow=c(1,2)); plot(daG11); plot(daG11.mor)
```



We can work with a matrix representation too. Default is that the output representation is the same as the input representation, but this can be changed:

```
moralize( daG11m, result="dgCMatrix" )

## 4 x 4 sparse Matrix of class "dgCMatrix"
##   a b c d
## a . 1 . .
## b 1 . 1 1
## c . 1 . 1
## d . 1 1 .
```

3.2 Topological sort - is a directed graph a DAG?

A topological ordering of a directed graph is a linear ordering of its vertices such that, for every edge $(u \to v)$, u comes before v in the ordering. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering.

```
topo_sort(daG11)

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

topo_sort(daG11m)
```

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

topo_sort(daG11M)

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

The following graph has a cycle and hence a topological ordering can not be made:

```
topo_sort(dag(~a:b + b:c + c:a))

## character(0)
```

Likewise for an undirected graph (recall that we regard an undirected edge as a bidirected edge):

```
topo_sort( ug( ~a:b ) )

## character(0)
```

3.3 Getting the cliques of an undirected graph

In graph theory, a clique is often a complete subset of a graph. A maximal clique is a clique which can not be enlarged. In statistics (and that is the convention we follow here) a clique is usually understood to be a maximal clique. Finding the cliques of a general graph is an NP–complete problem. Finding the cliques of triangulated graph is linear in the number of cliques.

```
str( get_cliques(uG11) )

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

str( get_cliques(uG11m) )

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

str( get_cliques(uG11M) )

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

For `graphNEL` objects one may also use the `maxClique()` function in **RBGL**, but `get_cliques()` applies also to matrices and it is substantially faster.

3.4 Perfect ordering and maximum cardinality search

An undirected graph is triangulated (or chordal) if it has no cycles of length ≥ 4 without a chord. This is equivalent to that the vertices can be given a perfect ordering. A perfect ordering (if it exists) can be obtained with Maximum Cardinality Search. If `character(0)` is returned the graph is not triangulated. Otherwise a perfect ordering of the nodes is returned.

```
mcs(uG11)

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

mcs(uG11m)

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

mcs(uG11M)

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

In some applications it is convenient to retain control over the ordering (if it exists). For example:

```
mcs(uG11, root=c("a","c"))

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

The desired ordering (specified by `root`) is followed as far as possible (here only to the first variable `"a"`). Notice the output when applying `mcs()` to a directed graph:[2]

```
mcs( daG11 )

## character(0)

mcs( as(daG11, "matrix") )

## character(0)
```

[2]Perhaps better to signal an error.

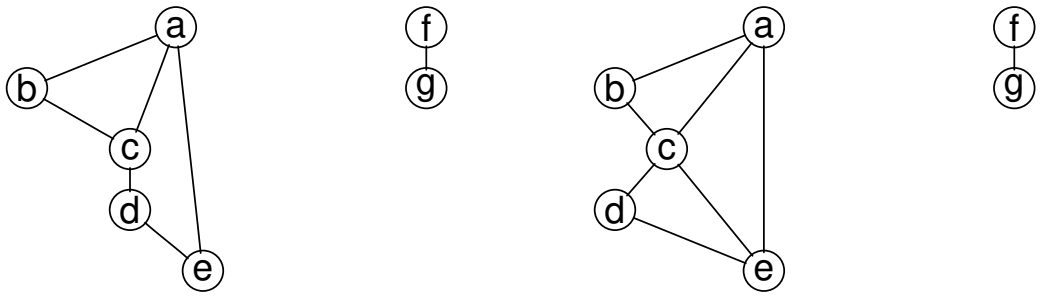
3.5 Triangulation

Any undirected graph can be triangulated by adding edges to the graph, so called fill-ins:

```
uG <- ug( ~a:b:c + c:d + d:e + a:e + f:g )
mcs(uG)

## character(0)

(tuG  <- triangulate(uG))

## A graphNEL graph with undirected edges
## Number of Nodes = 7
## Number of Edges = 8

mcs(tuG)

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

```
(tuG  <- triangulate(uG))

## A graphNEL graph with undirected edges
## Number of Nodes = 7
## Number of Edges = 8
```

```
par(mfrow=c(1,2)); plot(uG); plot(tuG)
```



3.6 RIP ordering / junction tree

A RIP–ordering of the cliques of a triangulated graph can be obtained as:

```
rp <- rip(tuG); rp

## cliques
##   1 : c a b
##   2 : c a e
##   3 : c d e
##   4 : f g
## separators
##   1 :
##   2 : c a
##   3 : c e
##   4 :
## parents
##   1 : 0
##   2 : 1
##   3 : 2
##   4 : 0
```

```
plot(rp )
```

There is more information in a RIP-object:

```
names(rp)

## [1] "nodes"      "cliques"    "separators" "parents"    "children"
## [6] "host"       "nLevels"    "childList"

rp$nodes

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

rp$host

## [1] 2 1 3 3 3 4 4
```

```
rp$children

## [1]  2  3 NA NA

str(rp$separators)

## List of 4
##  $ : chr(0)
##  $ : chr [1:2] "c" "a"
##  $ : chr [1:2] "c" "e"
##  $ : chr(0)
```

The `host` component tells for each node, a clique in which the node can be found

The function `jTree` takes an undirected graph as input; triangulates it if it is not already so and then finds a RIP–ordering.

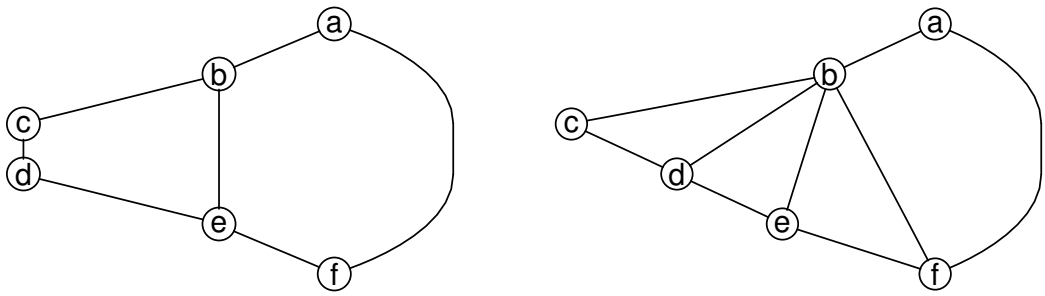
3.7 Minimal triangulation and maximum prime subgraph decomposition

A triangulation is minimal if no fill-ins can be removed without breaking the property that the graph is triangulated. (A related concept is the minimum triangulation, which is the the graph with the smallest number of fill-ins. The minimum triangulation is unique, but finding the minimum triangulation is NP-hard.)

For example, this graph has two 4–cycles:

```
g1 <- ug(~a:b + b:c + c:d + d:e + e:f + a:f + b:e)
g1mt <- minimal_triang(g1) # A minimal triangulation
```

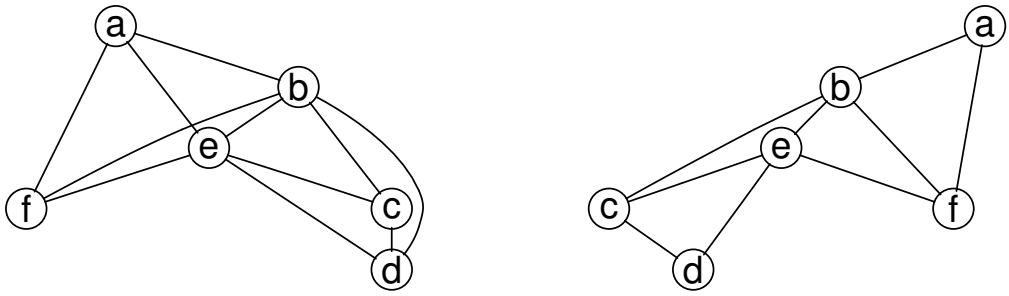
```
par(mfrow = c(1,2)); plot(g1); plot(g1mt)
```



The following graph is also a triangulation of `g1`, and from this a minimal triangulation can be obtained:

```
g2 <- ug(~a:b:e:f + b:c:d:e)
g1mt2 <- minimal_triang(g1, tobject=g2)
```

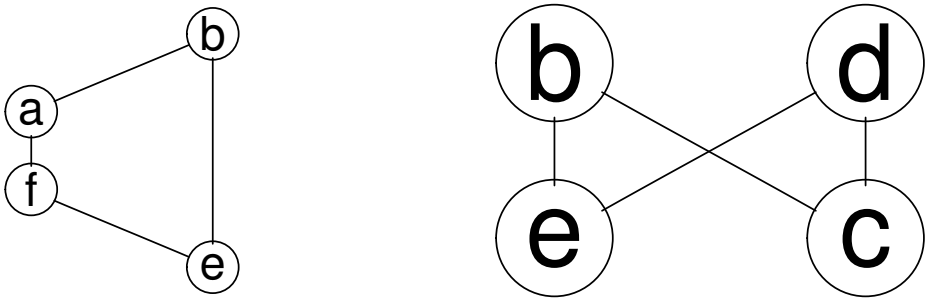
```
par(mfrow = c(1,2)); plot(g2); plot(g1mt2)
```



The junction tree of a maximum prime subgraph decomposition is obtained with:

```
mm <- mpd(g1); mm

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

```
par(mfrow = c(1,2))
plot(subGraph(mm$cliques[[1]], g1))
plot(subGraph(mm$cliques[[2]], g1))
```



4 Time and space considerations

4.1 Time

It is worth noticing that working with graphs representated as `graphNEL` objects can be somewhat slower than working with graphs represented as adjacency matrices. Consider finding the cliques of an undirected graph represented as a `graphNEL` object or as a matrix:

```
if(require(microbenchmark)){
    microbenchmark(
        RBGL::maxClique(uG11),
        get_cliques(uG11),
        get_cliques(uG11m),
        get_cliques(uG11M),
        times=10) }

## Loading required package:  microbenchmark

## Unit: microseconds
##                    expr     min      lq    mean  median      uq     max neval
##  RBGL::maxClique(uG11)  106.10  112.97  144.71  128.96  172.86  226.81    10
##      get_cliques(uG11) 1323.96 1366.87 1499.62 1441.66 1533.29 1976.08    10
##     get_cliques(uG11m)   20.57   22.03   26.59   25.39   27.24   46.58    10
##     get_cliques(uG11M)   27.77   30.56   37.72   35.60   41.49   59.20    10
##  cld
##   a
##    b
##   a
##   a
```

4.2 Space

The `graphNEL` representation is – at least – in principle more economic in terms of space requirements than the adjacency matrix representation (because the adjacency matrix representation uses a 0 to represent a "missing edge". The sparse matrix representation is clearly only superior to the standard matrix representation if the graph is sparse:

```
V <- 1:300
M <- 1:10
## Sparse graph
##
g1 <- randomGraph(V, M, 0.05)
length(edgeList(g1))

## [1] 1194
```

```
s <- c(NEL=object.size(g1),
       dense=object.size(as(g1, "matrix")),
       sparse=object.size(as(g1, "dgCMatrix")))
s / max(s)

##     NEL   dense  sparse
## 1.00000 0.66642 0.06134

## More dense graph
##
g1 <- randomGraph(V, M, 0.5)
length(edgeList(g1))

## [1] 42060

s <- c(NEL=object.size(g1),
       dense=object.size(as(g1, "matrix")),
       sparse=object.size(as(g1, "dgCMatrix")))
s / max(s)

##    NEL  dense sparse
## 1.0000 0.0216 0.0299
```

5 Graph queries

The **graph** and **RBGL** packages implement various graph operations for `graphNEL` objects. See the documentation for these packages. The **gRbase** implements a few additional functions, see Section 1. An additional function in **gRbase** for graph operations is `querygraph()`. This function is intended as a wrapper for the various graph operations available in **gRbase**, **graph** and **RBGL**. There are two main virtues of `querygraph()`: 1) `querygraph()` operates on any of the three graph representations described above[3] and 2) `querygraph()` provides a unified interface to the graph operations. The general syntax is

```
args(querygraph)

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

[3] Actually not quite yet, but it will be so in the future.