graph, RBGL, Rgraphviz

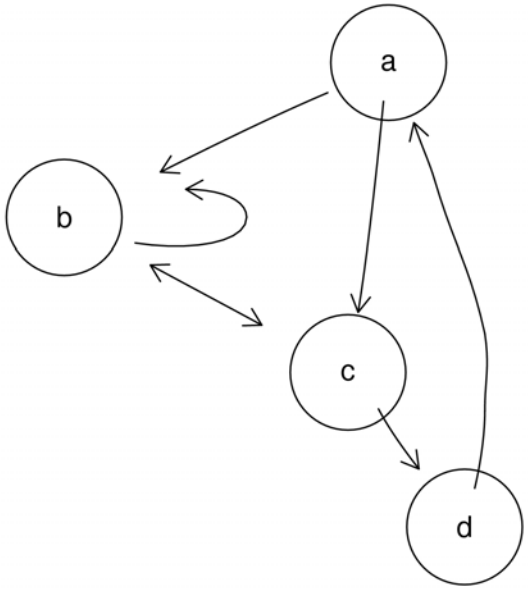
graph basic class definitions and functionality

RBGL interface to graph algorithms (e.g. shortest path, connectivity)

Rgraphviz rendering functionality
Different layout algorithms.
Node plotting, line type, color etc. can be controlled by the user.

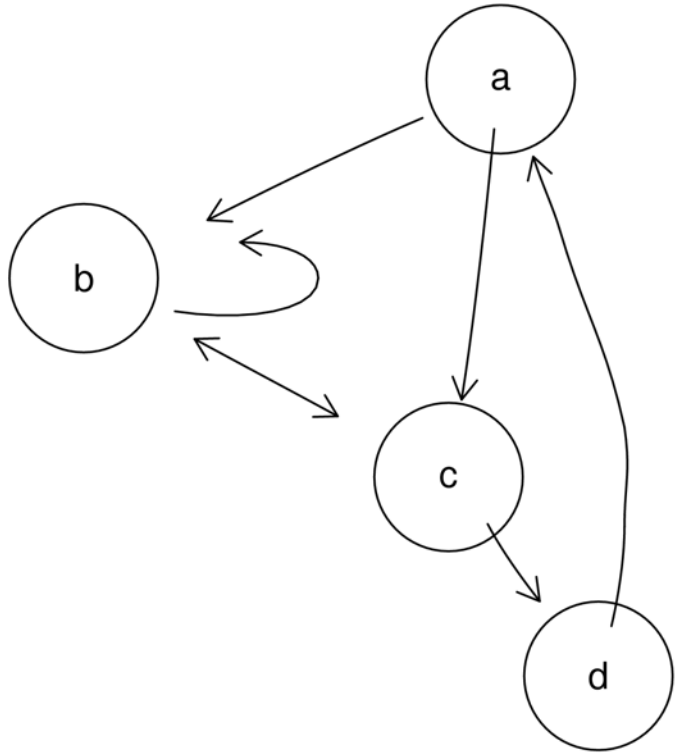
Creating our first graph

```
library(graph); library(Rgraphviz)

edges <- list(a=list(edges=2:3),
              b=list(edges=2:3),
              c=list(edges=c(2,4)),
              d=list(edges=1))

g <- new("graphNEL", nodes=letters[1:4], edgeL=edges,
         edgemode="directed")

plot(g)
```



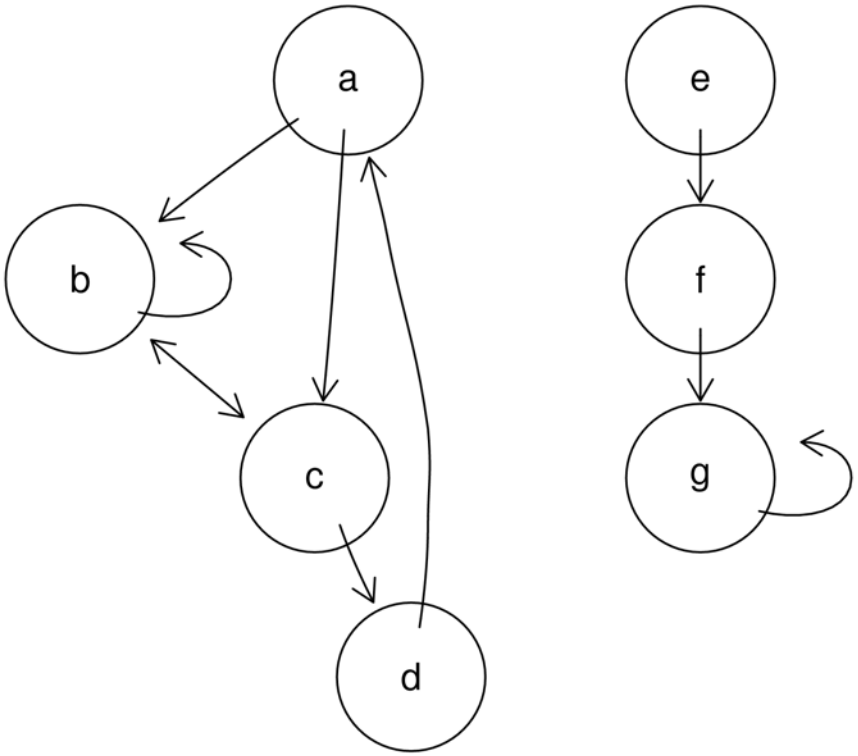
Querying nodes, edges, degree

```
> nodes(g)
[1] "a" "b" "c" "d"

> edges(g)
$a
[1] "b" "c"
$b
[1] "b" "c"
$c
[1] "b" "d"
$d
[1] "a"

> degree(g)
$inDegree
a b c d
1 3 2 1
$outDegree
a b c d
2 2 2 1
```



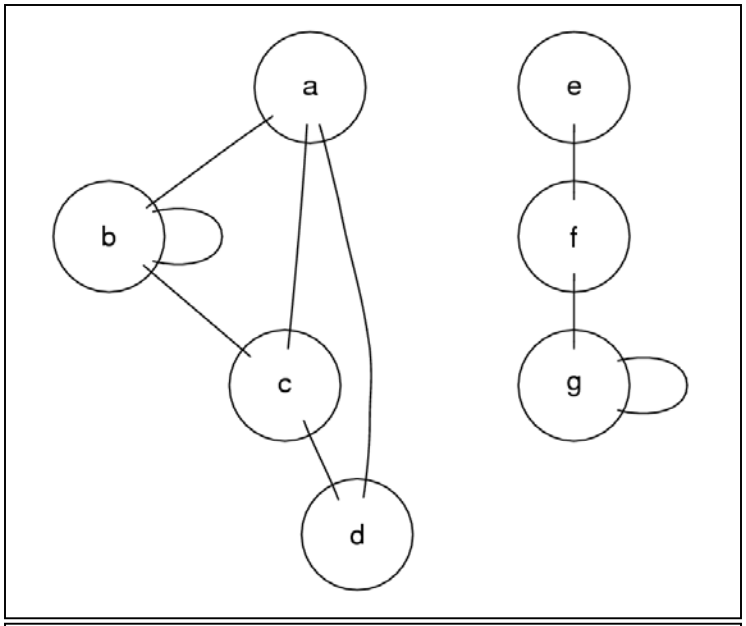
Adjacent and accessible nodes

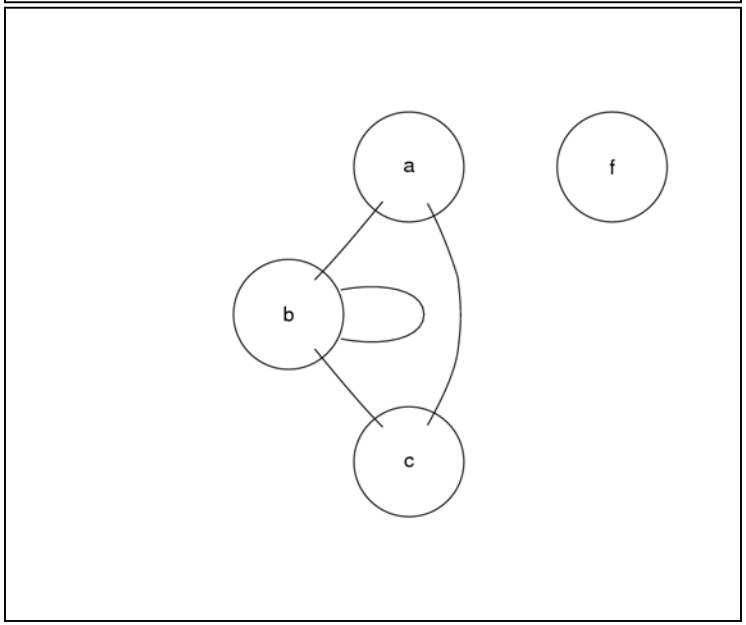
```
> adj(g, c("b", "c"))
$b
[1] "b" "c"
$c
[1] "b" "d"

> acc(g, c("b", "c"))
$b
a c d
3 1 2

$c
a b d
2 1 1
```



Undirected graphs, subgraphs, boundary graph

```
> ug <- ugraph(g)
> plot(ug)


> sg <- subGraph(c("a", "b",
      "c", "f"), ug)
> plot(sg)
> boundary(sg, ug)
> $a
>[1] "d"
> $b
> character(0)
> $c
>[1] "d"
> $f
>[1] "e" "g"
```





Weighted graphs

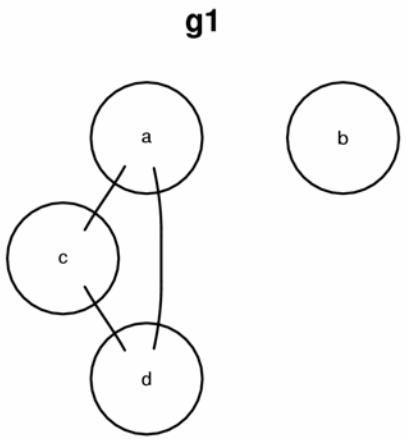
```
> edges <- list(a=list(edges=2:3, weights=1:2),
+               b=list(edges=2:3, weights=c(0.5, 1)),
+               c=list(edges=c(2,4), weights=c(2:1)),
+               d=list(edges=1, weights=3))

> g <- new("graphNEL", nodes=letters[1:4],
edgeL=edges, edgemode="directed")

> edgeWeights(g)
$a
2 3
1 2
$b
  2   3
0.5 1.0
$c                    $d
2 4                   1
2 1                   3
```

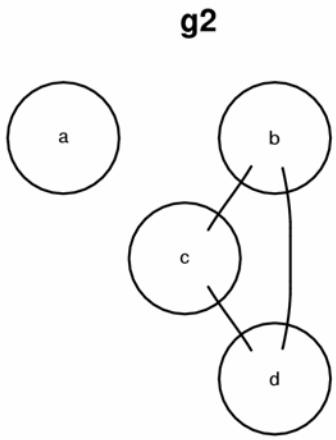
Graph manipulation

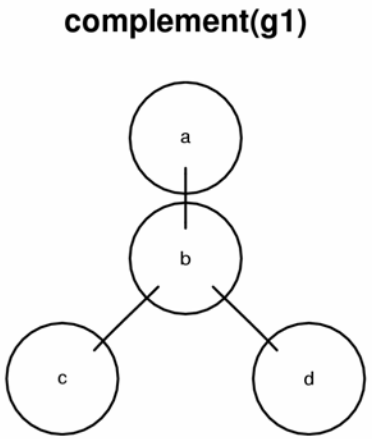
```
> g1 <- addNode("e", g)

> g2 <- removeNode("d", g)

> ## addEdge(from, to, graph, weights)
> g3 <- addEdge("e", "a", g1, pi/2)

> ## removeEdge(from, to, graph)
> g4 <- removeEdge("e", "a", g3)

> identical(g4, g1)
[1] TRUE
```

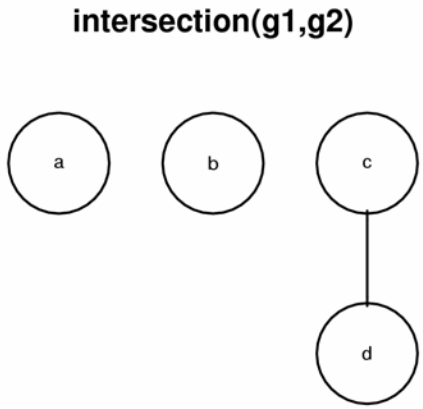
Graph algebra

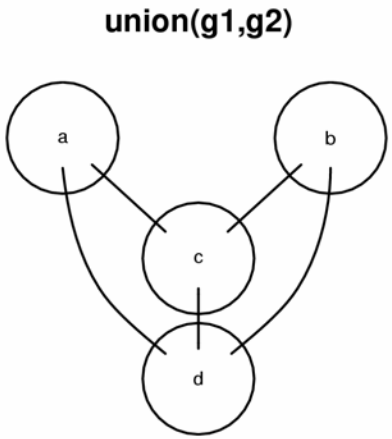
```
set.seed(4713)
V <- letters[1:4]
g1 <- randomGraph(V,
       1, .55)
g2 <- randomGraph(V,
      1, .55)
```





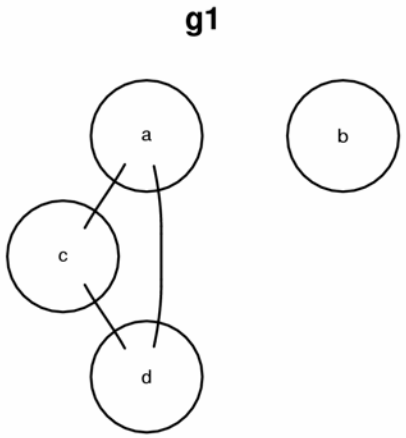


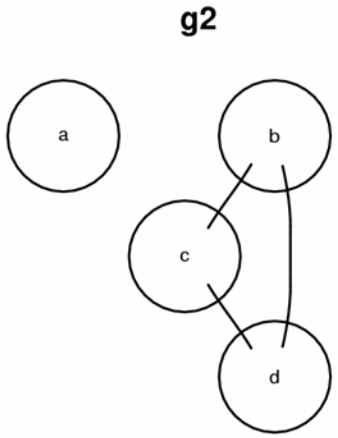




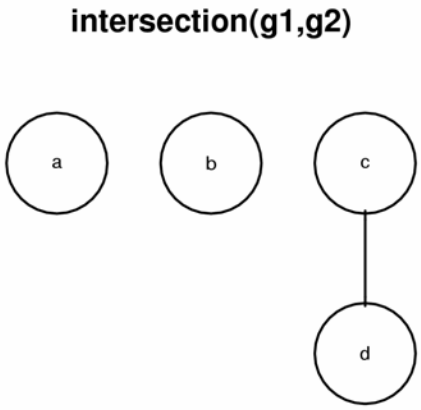
Graph algebra

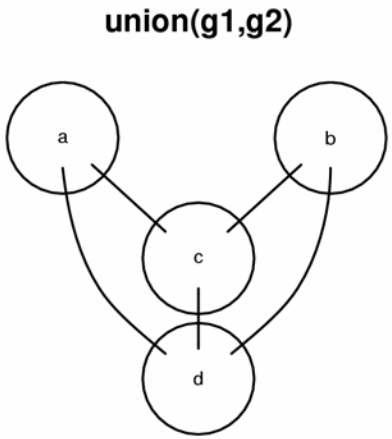
```
set.seed(4713)
V <- letters[1:4]
g1 <- randomGraph(V,
       1, .55)
g2 <- randomGraph(V,
      1, .55)
```





union and intersection are defined for graphs with common node sets





Random graphs

Random edge graph: `randomEGraph(V, p, edges)`

`V:` nodes

either `p:` probability per edge

or `edges:` number of edges

Random graph with latent factor: `randomGraph(V, M, p, weights=TRUE)`

`V:` nodes

`M:` latent factor

`p:` probability

For each node, generate a logical vector of length `length(M)`, with P(TRUE)=`p`. Edges are between nodes that share >= 1 elements. Weights can be generated according to number of shared elements.

Random graph with predefined degree distribution:

`randomNodeGraph(nodeDegree)`

`nodeDegree:` named integer vector

sum(nodeDegree)%%2==0

Graph representations

node-edge list: `graphNEL`

- **list of nodes**
- **list of out-edges for each node**

from-to matrix

adjacency matrix

adjacency matrix (sparse) `graphAM` (to come)

node list + edge list: `pNode, pEdge` (Rgraphviz)

- **list of nodes**
- **list of edges (node pairs, possibly ordered)**

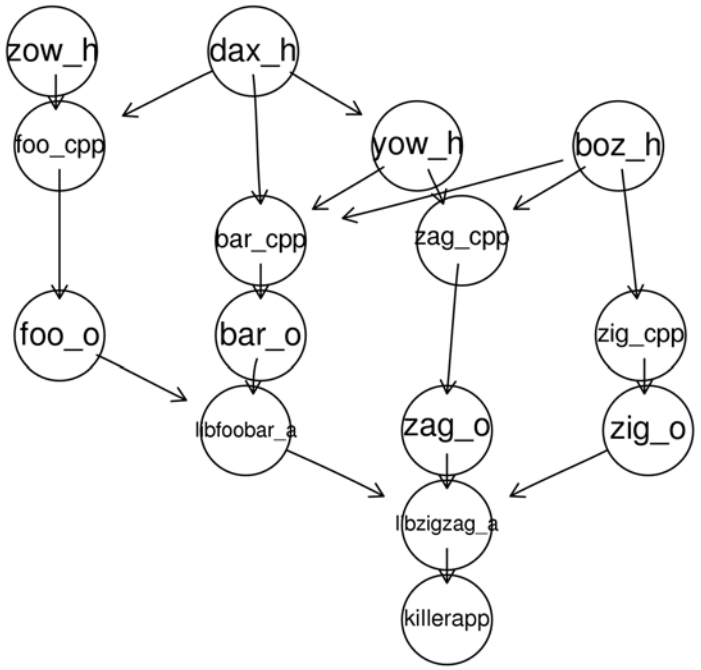
`Ragraph:` representation of a laid out graph

Graph representations: from-to-matrix

```
> ft
     [,1] [,2]
[1,]    1    2
[2,]    2    3
[3,]    3    1
[4,]    4    4

> ftM2adjM(ft)
  1 2 3 4
1 0 1 0 0
2 0 0 1 0
3 1 0 0 0
4 0 0 0 1
```

RBGL: interface to the ‘Boost Graph Library’

```
> library(RBGL)
> data(FileDep)

> ts <- tsort(FileDep)
> nodes(FileDep)[ts+1]
[1]  “zow_h"    "boz_h"
[3]  "zig_cpp" "zig_o”
[5]  "dax_h“    "yow_h"
[7]  "zag_cpp" "zag_o"
[9]  "bar_cpp" "bar_o"
[11] "foo_cpp" "foo_o"
[13] "libfoobar_a“
           "libzigzag_a"
[15] "killerapp"
>
```



topological sort

linear ordering of the edges such that:

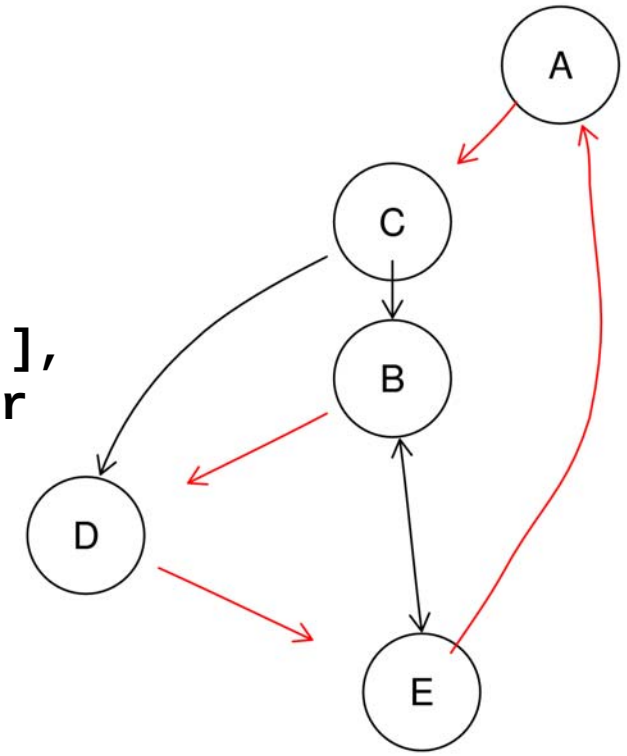
***if edge (u,v) appears in the graph, then u comes before v in the ordering*.**

The graph must be a directed acyclic graph (DAG).

The implementation consists mainly of a call to depth-first search

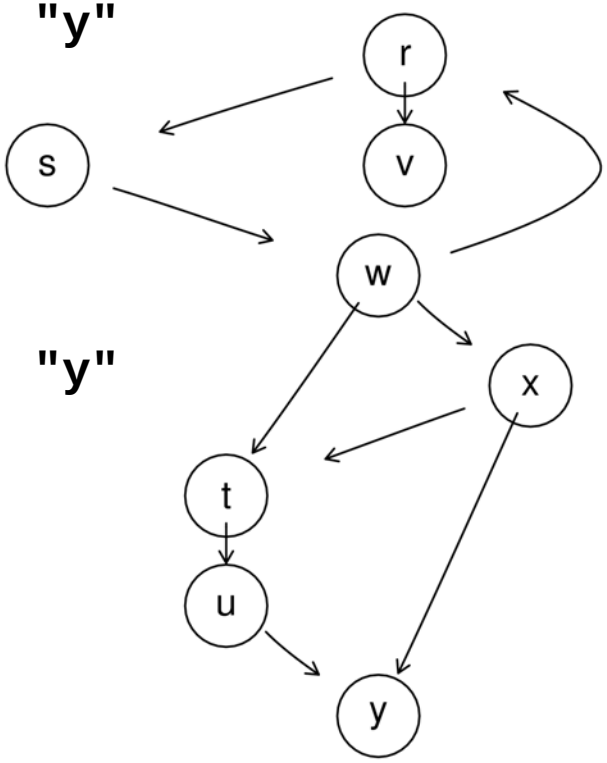
minimal spanning tree

```
km <-
fromGXL(file(system.file("GXL/kmstEx
.gxl", package = "graph")))

ms <- mstree.kruskal(km)

e <- buildEdgeList(km)
n <- buildNodeList(km)

for(i in 1:ncol(ms$edgeList))

e[[paste(ms$nodes[ms$edgeList[,i]],
       collapse="~")]]@attrs$color
              <- "red"

z <- agopen(nodes=n, edges=e,
edgeMode="directed", name="")

plot(z)
```



breadth first search

```
> br <- bfs(dd, "r")
> nodes(dd)[br]
[1] "r" "s" "v" "w" "t" "x" "u" "y"
>
> bs <- bfs(dd, "s")
> nodes(dd)[bs]
[1] "s" "w" "r" "t" "x" "v" "u" "y"
```

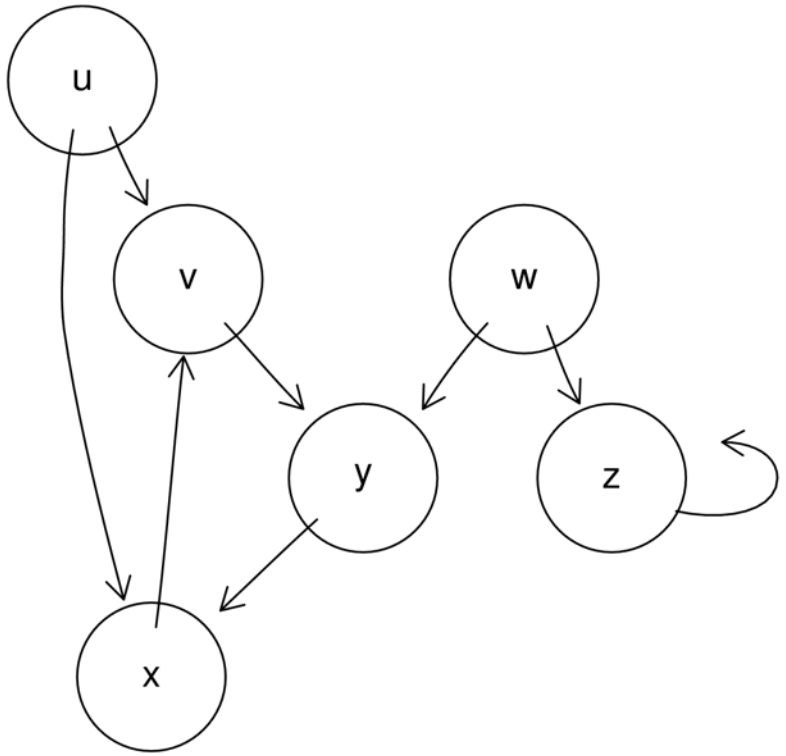


depth first search

```
> df <- dfs(dd2, "u", FALSE)
```

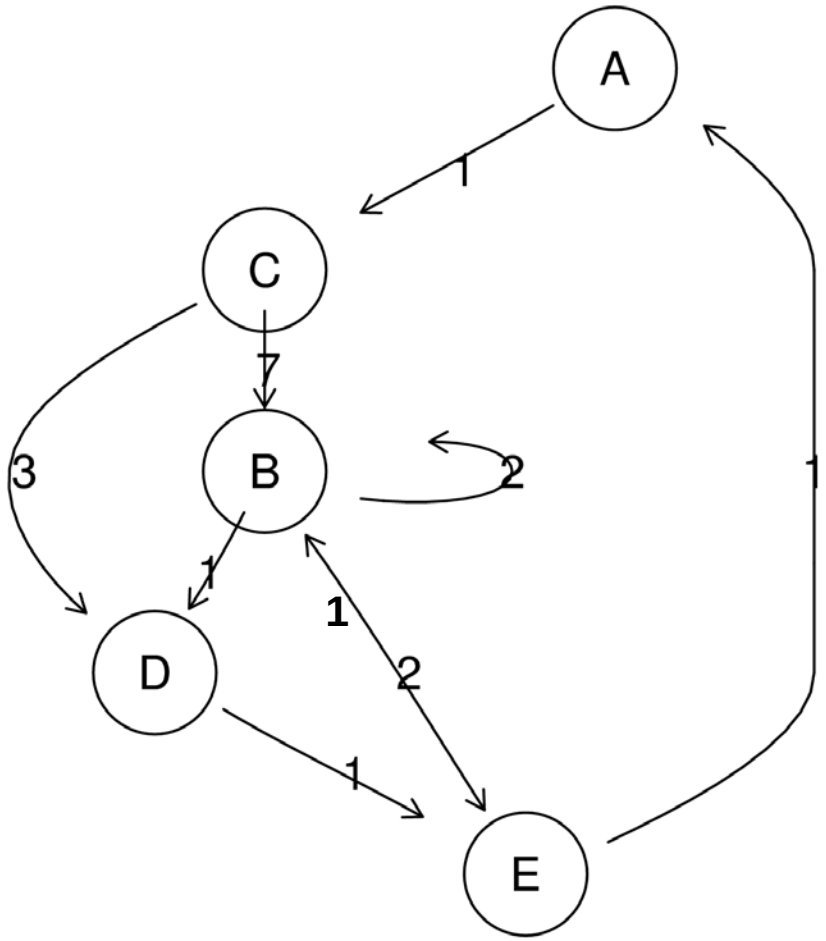
```
> nodes(dd)[df$discovered]
[1] "u" "v" "y" "x" "w" "z"
```

```
> nodes(dd)[df$finish]
[1] "x" "y" "v" "u" "z" "w"
```



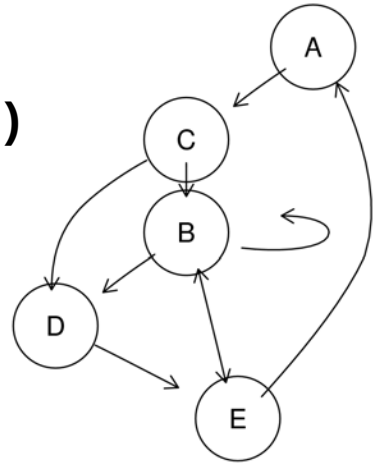
shortest path

```
> sp.between(g, "E", "C")

$"E:C"

$"E:C"$path
[1] "E" "A" "C"

$"E:C"$length
[1] 2

$"E:C"$pweights
E->A A->C
   1    1

> dijkstra.sp(g)
$distances
A B C D E
0 6 1 4 5
$penult
A B C D E
1 5 1 3 4
$start
A
1
```

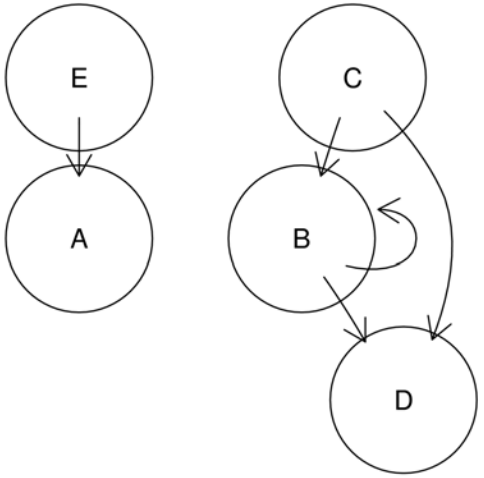


connected components

```
> g1 <- removeEdge('A', 'C', g)
> g1 <- removeEdge('D', 'E', g1)
> g1 <- removeEdge('B', 'E', g1)
> g1 <- removeEdge('E', 'B', g1)

> connectedComp(g)

$"1"
[1] "A" "B" "C" "D" "E"
```



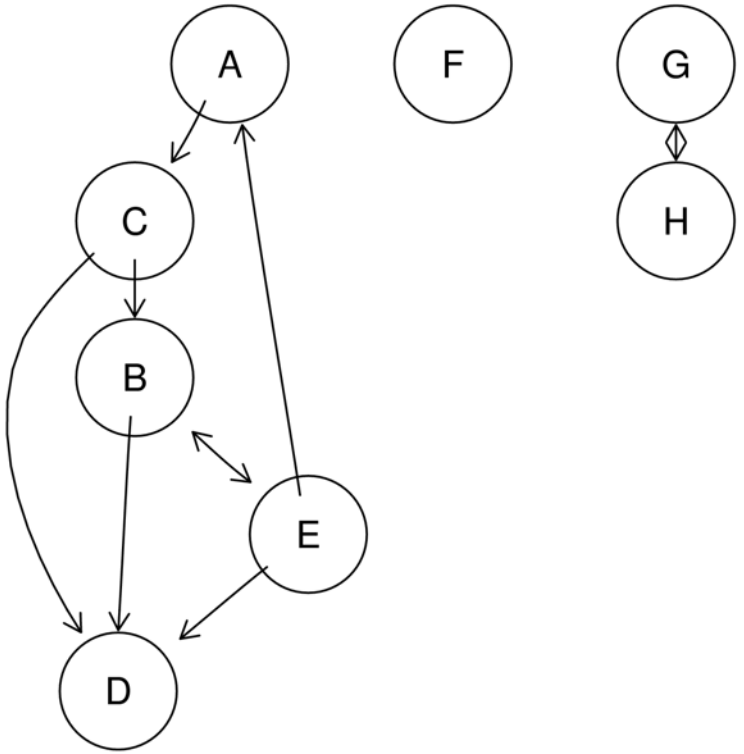

```
> connectedComp(g1)

$"1"
[1] "A" "E"
$"2"
[1] "B" "C" "D"
```



strongly connected components

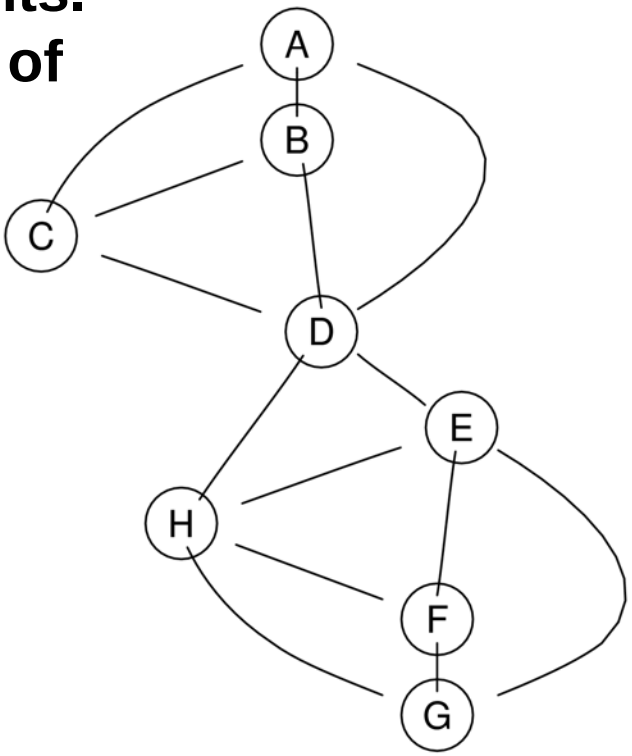
applies only to directed graphs

```
> strongComp(km)
$"1"
[1] "D"
$"2"
[1] "A" "B" "C" "E"
$"3"
[1] "F"
$"4"
[1] "G" "H"

> connectedComp(ugraph(km))
$"1"
[1] "A" "B" "C" "D" "E"
$"2"
[1] "F"
$"3"
[1] "G" "H"
```



connectivity

Let g have single connected component.
Edge connectivity of g: minimum number of edges in g that can be cut to produce a graph with two components.
Minimum disconnecting set: the set of edges in this cut.




```
> edgeConnectivity(g)
$connectivity
[1] 2

$minDisconSet
$minDisconSet[[1]]
[1] "D" "E"

$minDisconSet[[2]]
[1] "D" "H"
```

Rgraphviz: the different layout engines

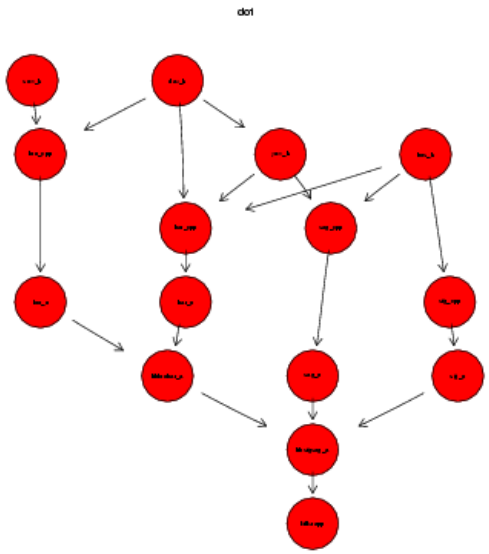
dot: directed graphs. Works best on DAGs and other graphs that can be drawn as hierarchies.

neato: undirected graphs using 'spring' models

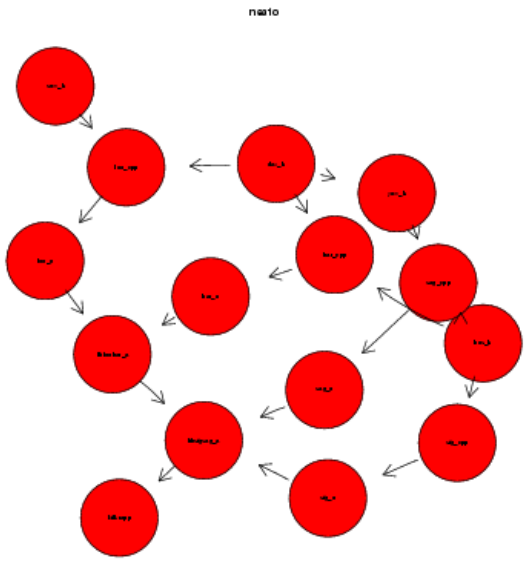
twopi: radial layout. One node ('root') chosen as the center. Remaining nodes on a sequence of concentric circles about the origin, with radial distance proportional to graph distance. Root can be specified or chosen heuristically.

Rgraphviz: the different layout engines

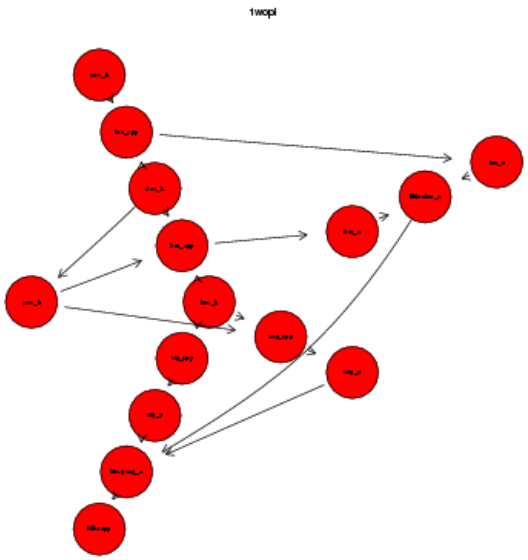
dot



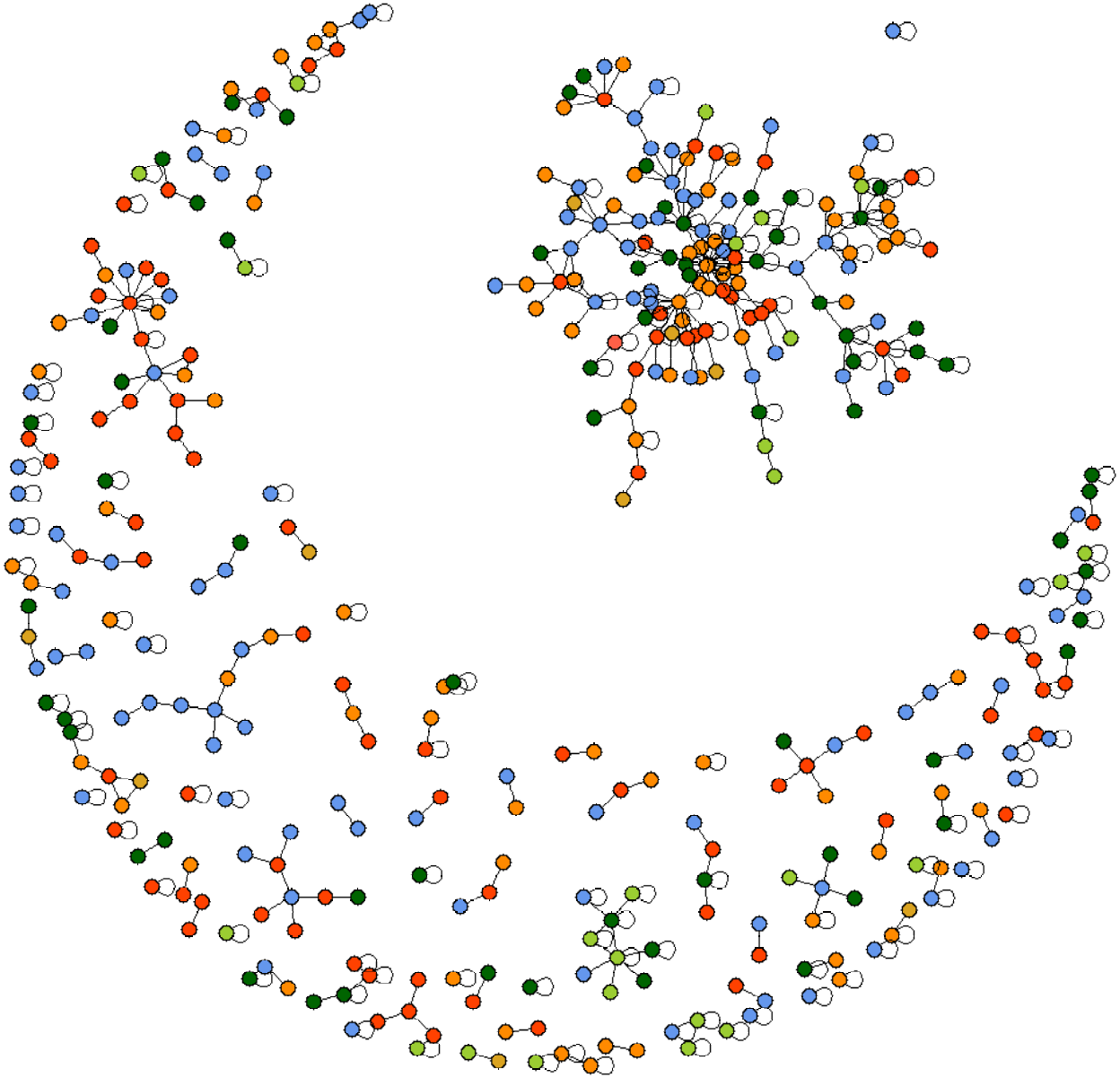
neato



twopi



domain combination graph



GXL: graph exchange language

```
<gxl>
 <graph edgemode="directed" id="G">
  <node id="A"/>
  <node id="B"/>
  <node id="C"/>
  …
  <edge id="e1" from="A" to="C">
   <attr name="weights">
    <int>1</int>
   </attr>
  </edge>
  <edge id="e2" from="B" to="D">
   <attr name="weights">
    <int>1</int>
   </attr>
  </edge>
  …
</graph>
</gxl>
```

from graph/GXL/kmstEx.gxl

GXL (www.gupro.de/GXL) is "an XML sublanguage designed to be a standard exchange format for graphs". The graph package provides tools for im- and exporting graphs as GXL