RBGL: R interface to boost graph library

L. Long, VJ Carey, and R. Gentleman

March 22, 2019

Summary. The RBGL package is primarily an interface from R to the Boost Graph Library (BGL). It includes some graph algorithms built on top of those from BGL and some algorithms independent of BGL.

Contents

1 Basic notations/Preliminaries

1.1 Basics Notations

We use the following notation:

G: a graph, represented as G = (V, E); *V* = v1, v2, ..., vn: a set of vertices (or nodes); *E* = e1, e2, ..., em: a set of edges with ei = [vj, vk], with vj, vk are in V; *W* = w1, w2, ..., wm: a set of weights of the edges, i.e., wi is the weight on edge ei.

A *walk* is a sequence of vertices v1, v2, ..., vk such that for all i, [vi, vi+1] in *E*. A *path* is a walk without repeated vertices. A *cycle* is a path that begins and ends at the same vertice.

A *directed* graph is a graph with direction assigned to its edges, therefore, [vj, vk] != [vk, vj].

A *directed acyclic graph (DAG)* is a directed graph with no directed cycle.

An *in-degree* of vertex v is the total number of edges [u, v] in E; an *out-degree* of v is the total number of edges [v, u] in *E*.

A network *N* is a directed graph *G* with (a) a source *s* whose in-degree is 0, (b) a sink *t* whose out-degree is 0, and (c) a *capacity* for each edge in a network.

A *flow* in *N* assigns a value on each edge that doesn't exceed its capacity, all the internal vertices have the same incoming flow as the outgoing flow, *s* has outgoing flow only, *t* has incoming flow only.

1.2 Examples in use

We are going to use the following graphs repeatedly in the examples.

```
> con <- file(system.file("XML/bfsex.gxl", package="RBGL"))
> bf <- fromGXL(con)
> close(con)
```

```
> con <- file(system.file("XML/dfsex.gxl", package="RBGL"))
> df <- fromGXL(con)
> close(con)
```

```
> con <- file(system.file("XML/dijkex.gxl", package="RBGL"))
> dijk <- fromGXL(con)
> close(con)
```

```
> con <- file(system.file("XML/conn.gxl", package="RBGL"))
> coex <- fromGXL(con)
> close(con)

> con <- file(system.file("XML/conn2.gxl", package="RBGL"))
> coex2 <- fromGXL(con)
> close(con)

> con <- file(system.file("XML/conn2iso.gxl", package="RBGL"))
> coex2i <- fromGXL(con)
> close(con)

> con <- file(system.file("XML/kmstEx.gxl", package="RBGL"))
> km <- fromGXL(con)
> close(con)

> con <- file(system.file("XML/biconn.gxl", package="RBGL"))
> bicoex <- fromGXL(con)
> close(con)

> con <- file(system.file("XML/ospf.gxl", package="RBGL"))
> ospf <- fromGXL(con)
> close(con)

> con <- file(system.file("dot/joh.gxl", package="RBGL"))
> joh <- fromGXL(con)
> close(con)

> con <- file(system.file("XML/hcs.gxl", package="RBGL"))
> hcs <- fromGXL(con)
> close(con)

> con <- file(system.file("XML/snacliqueex.gxl", package="RBGL"))
> kclex <- fromGXL(con)
> close(con)

> con <- file(system.file("XML/snacoreex.gxl", package="RBGL"))
> kcoex <- fromGXL(con)
> close(con)
```

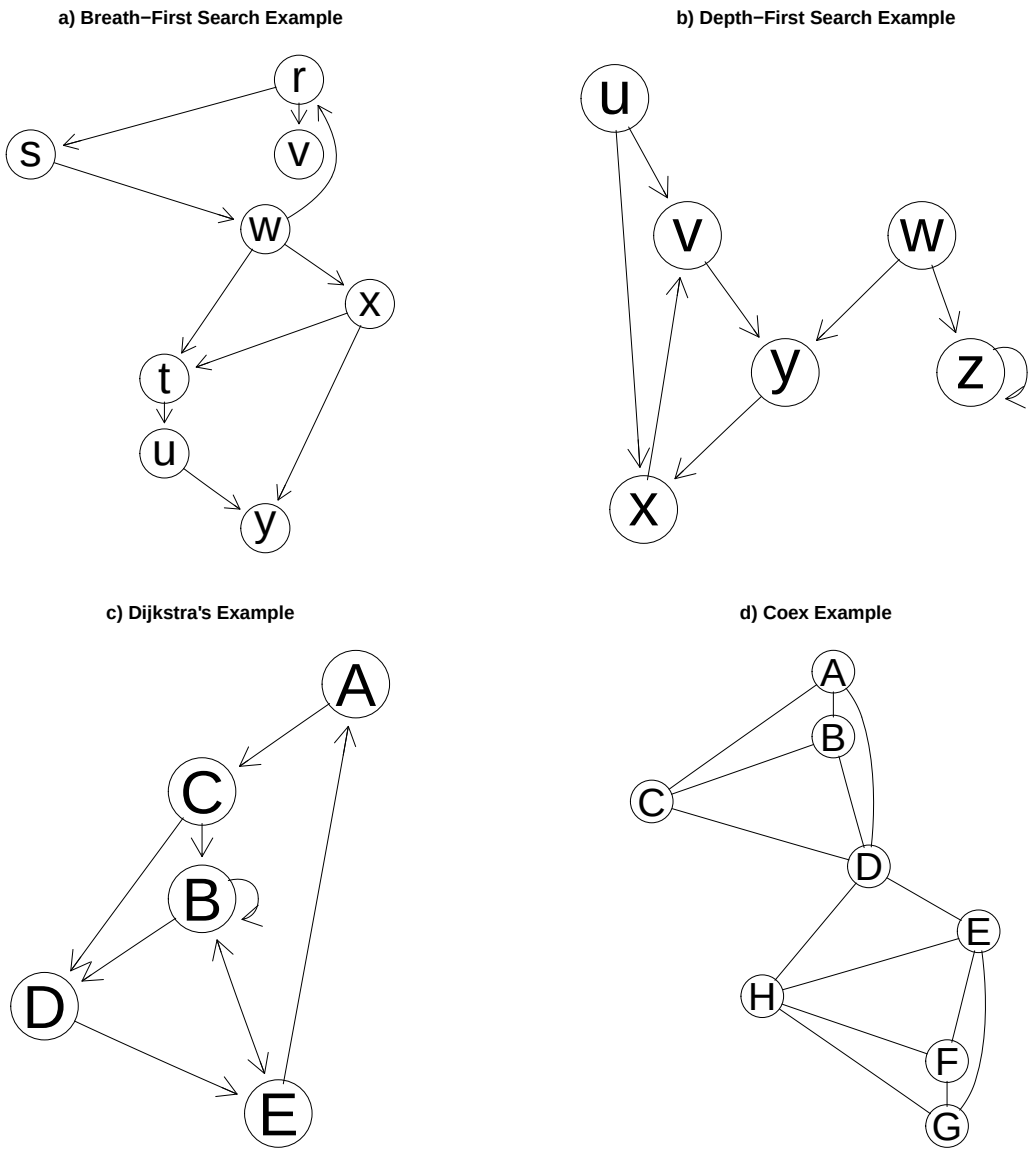

Figure 1: The example graphs (I).

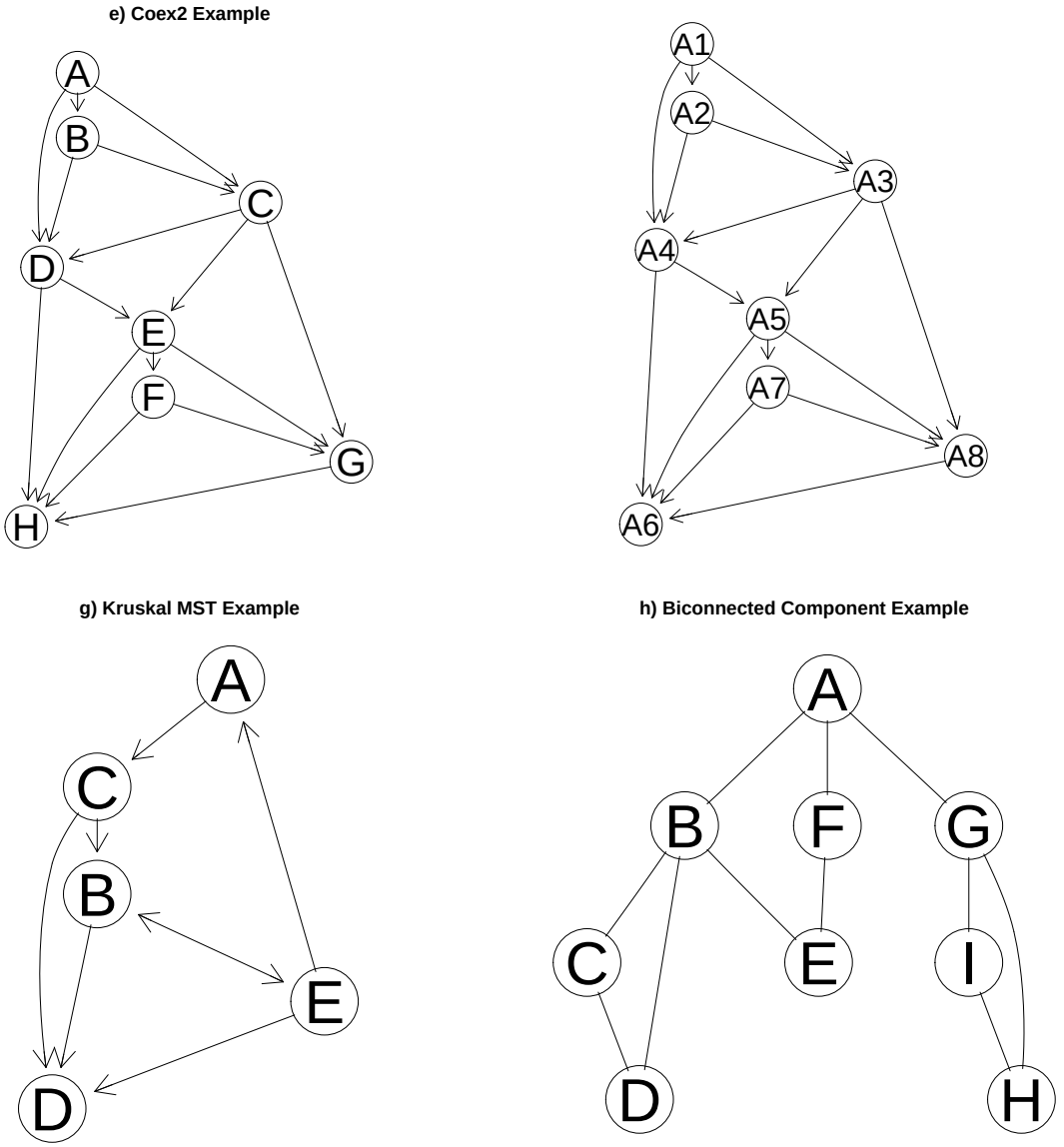

Figure 2: The example graphs (II).

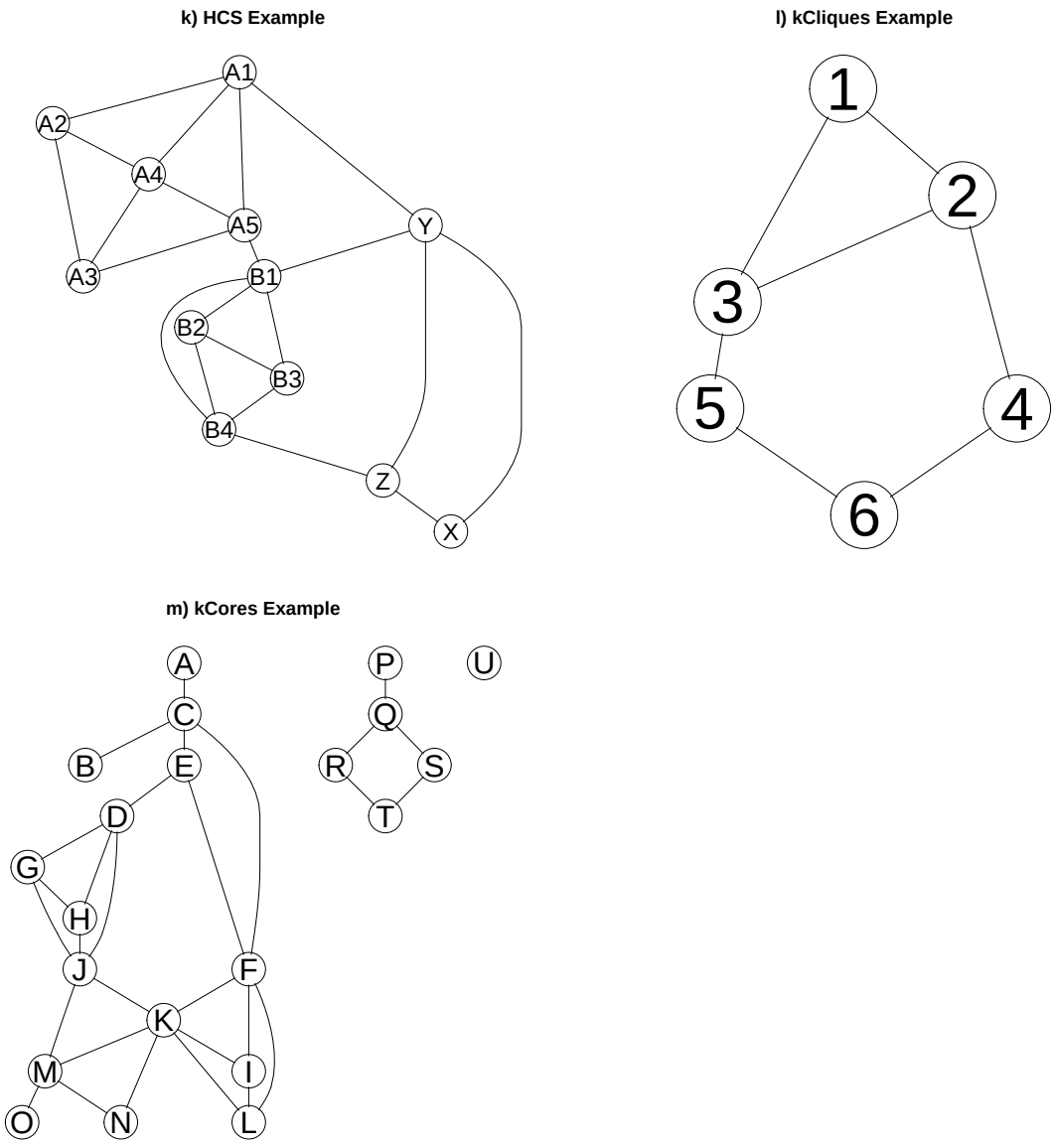

Figure 3: The example graphs (III).

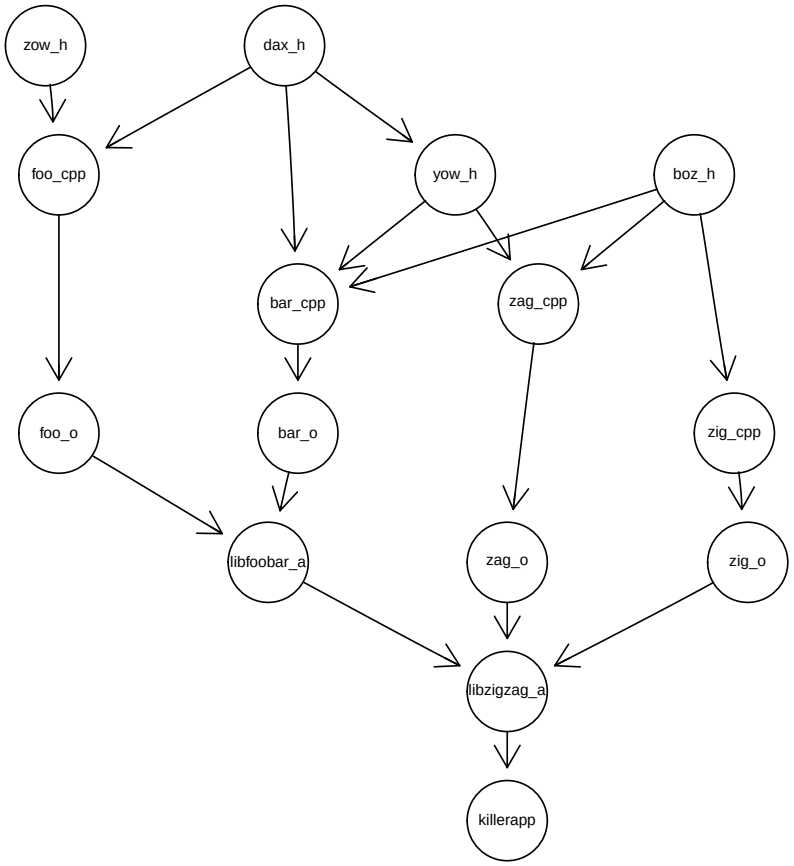

Figure 4: File dependency digraph example from Boost library.

2 Working with the Bioconductor graph class

An example object representing file dependencies is included, as shown in Figure 4.

```
> data(FileDep)
> FileDep

A graphNEL graph with directed edges
Number of Nodes = 15
Number of Edges = 19
```

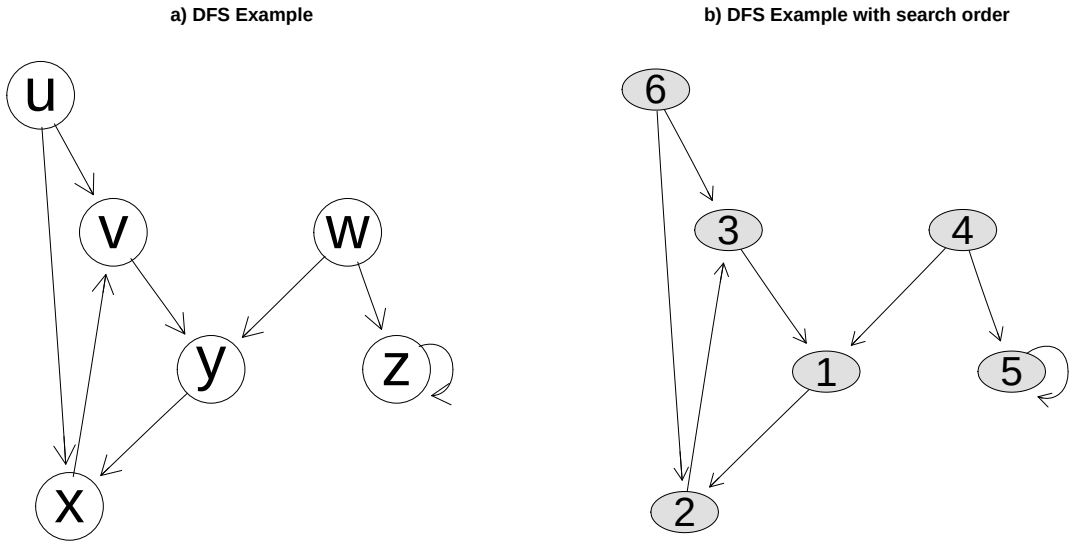

Figure 5: a) The graph for depth-first-search example. b) The graph for depth-first-search example, showing search orders.

3 Algorithms from BGL

3.1 Depth First Search

The `dfs` function returns two vectors of node names of discovery and finish order in a depth-first-search (DFS), starting at the given vertex.

```
> print(dfs.res <- dfs(df, "y"))

$discovered
[1] "y" "x" "v" "w" "z" "u"

$finish
[1] "v" "x" "y" "z" "w" "u"
```

In this example, DFS starts with y, reaches x and v; DFS restarts from w, reaches z; DFS restarts from u; at this point, all the vertices in the graph are visited once and only once. You could see the search order in the figure.

3.2 Breadth First Search

The `bfs` function returns a vector of node names of discovery order in a breadth-first search (BFS), starting at the given vertex.

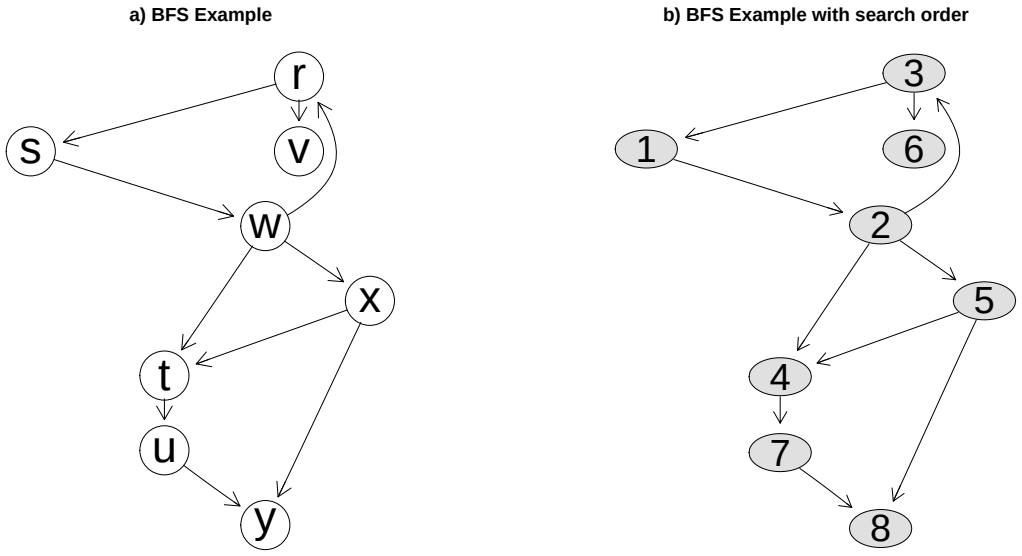

Figure 6: a) The graph for breadth-first-search example. b) The graph for breadth-first-search example, showing search orders.

```
> print(bfs.res <- bfs(bf,"s"))

[1] "s" "w" "r" "t" "x" "v" "u" "y"
```

In this example, BFS starts from vertex s, reaches w; from w BFS reaches r, t and x; from r BFS reaches v; from t BFS reaches u; from x BFS reaches y; at this time, BFS visits all the vertices in the graph once and only once.

3.3 Shortest paths

Edge weights play a major role in shortest-path problems. The weight of an edge in a graph could represent the relationship between the two vertices, such as distance, probability, etc.

TO-BE-FINALIZED: Our knowledge of such a relationship between two vertices is: (1) we know there is an edge and there is a measured value on it; (2) we know there is an edge but there is NO measured value on it; (3) we know there is NO edge; (4) we DO NOT know if there is an edge.

Corresponding edge weights are: case 1: measured value; case 2: `NA`; case 3: `Inf`; case 4: TO-BE-DETERMINED

When there is a loop of negative weight between two vertices, the distance between them is `-Inf`.

The shortest path problem is to find a path between two vertices where the sum of all the edge weights on this path is minimum.

There are two sets of algorithms available: (1) find shortest paths between a single vertex, say, *source s*, and all other vertices, i.e., *V-s*, available algorithms are: *Dijkstra's*, *Bellman Ford's* and *DAG*, and (2) find shortest paths between all pairs of vertices, available algorithms are: *Johnson's* and *Floyd Warshall's*.

Dijkstra's algorithm is for the single-source shortest-paths problem on graphs (directed or undirected) with non-negative weights on edges. If all the edges have the same weight, use depth-first-search instead.

```
> nodes(dijk)

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

> edgeWeights(dijk)

$A
C
1

$B
B D E
2 1 2

$C
B D
7 3

$D
E
1

$E
A B
1 1

> dijkstra.sp(dijk)

$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
```

The function `dijkstra.sp` finds the shortest paths from A, which is the first node in the node list - default source, to all other vertices in the graph: *B, C, D, E*, shown in the *distances* part. The *penult* shows TO-BE-FILLED-IN.

For instance, edge *A->C* exists and carries a weight of 1, so the shortest path from *A* to *C* is 1; the shortest path from *A* to *B* goes through *A->C->D->E->B* and has weight of 6 (1+3+1+1).

```
> nodes(ospf)[6]

[1] "RT3"

> dijkstra.sp(ospf,nodes(ospf)[6])

$distances
 RT1   N1   N3  RT2   N2  RT3  RT6   N4  RT4  RT5  RT7  N12  N13  N14 RT10   N6
   1    4    1    1    4    0    8    2    1    9   15   17   17   17   15   16
 N15  RT8   N7  RT9   N9  N11   N8 RT11 RT12  N10   H1
  24   16   20   19   19   22   18   18   19   21   29

$penult
 RT1   N1   N3  RT2   N2  RT3  RT6   N4  RT4  RT5  RT7  N12  N13  N14 RT10   N6
   3    1    6    3    4    6    6    6    3    9   10   10   10   10    7   15
 N15  RT8   N7  RT9   N9  N11   N8 RT11 RT12  N10   H1
  11   16   18   21   24   20   15   23   21   25   25

$start
RT3
  6

> sp.between(ospf, "RT6", "RT1")

$`RT6:RT1`
$`RT6:RT1`$length
[1] 7

$`RT6:RT1`$path_detail
[1] "RT6" "RT3" "N3"  "RT1"
```

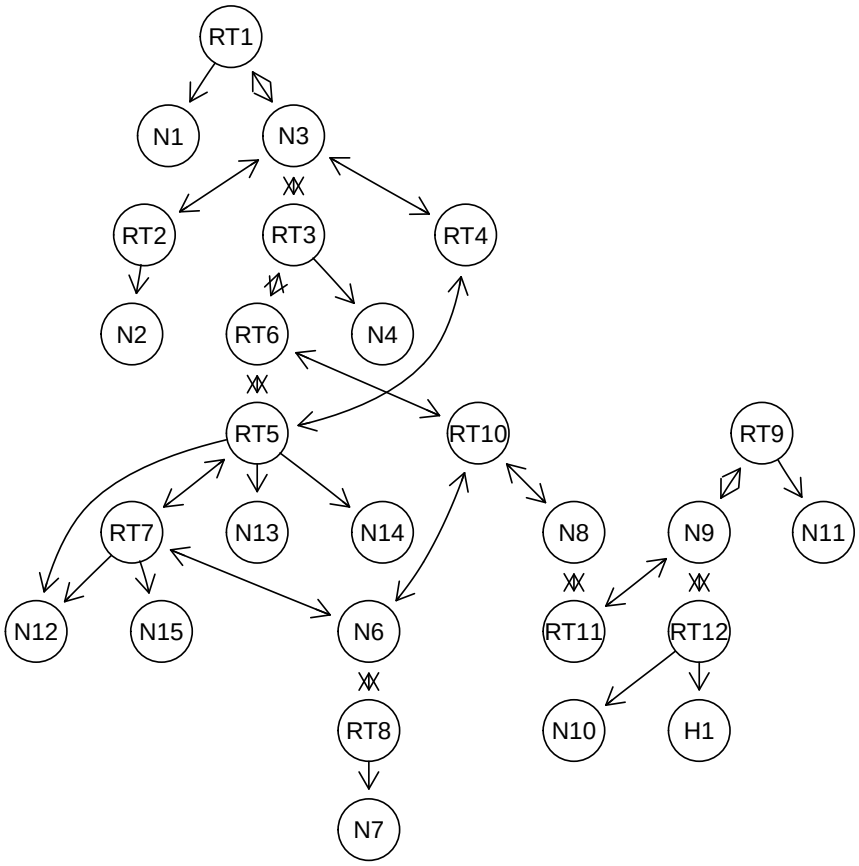

Figure 7: Network example from BGL.

```
$`RT6:RT1`$length_detail
$`RT6:RT1`$length_detail[[1]]
RT6->RT3  RT3->N3  N3->RT1
       6        1        0
```

The first part of this example finds the shortest paths from *start RT6* to all the other vertices in the graph, and the second part finds the shortest path between two vertices: *RT6* and *RT1*.

Bellman-Ford's algorithm is for the single-source shortest-paths problem on graphs (directed or undirected) with both positive and negative edge weights. The default source is the first entry in the list of nodes in the graph.

```
> dd <- coex2
> nodes(dd)

[1] "A" "B" "C" "D" "E" "G" "H" "F"
```

```
> bellman.ford.sp(dd)

$`all edges minimized`
[1] TRUE

$distance
A B C D E G H F
0 1 1 1 2 2 2 3

$penult
A B C D E G H F
1 1 1 1 3 3 4 5

$start
[1] "A"

> bellman.ford.sp(dd,nodes(dd)[2])

$`all edges minimized`
[1] TRUE

$distance
  A   B   C   D   E   G   H   F
Inf   0   1   1   2   2   2   3

$penult
A B C D E G H F
1 2 2 2 3 3 4 5

$start
[1] "B"
```

The first `bellman.ford.sp` returns the shortest paths from *start* A, which is the first vertex on the node list, to all other vertices. The second call shows the shortest paths from *start* B to all other vertices, since there is no path from B to A, the *distance* between them is `Inf`.

The *DAG algorithm* is for the single-source shortest-paths problem on a weighted, directed acyclic graph (DAG), which is more efficient for DAG than both Dijkstra's and Bellman-Ford's algorithms. When all the edges have the same weight, use depth-first-search instead.

```
> dd <- coex2
> dag.sp(dd)
```

```
$distance
A B C D E G H F
0 1 1 1 2 2 2 3

$penult
A B C D E G H F
1 1 1 1 3 3 4 5

$start
[1] "A"
```

```
> dag.sp(dd,nodes(dd)[2])

$distance
  A   B   C   D   E   G   H   F
Inf   0   1   1   2   2   2   3

$penult
A B C D E G H F
1 2 2 2 3 3 4 5

$start
[1] "B"
```

It's easy to see that *conn2.gxl* doesn't contain any cycle, so we could use function `dag.sp` on it. The first example finds the shortest paths from the *start A* to all other vertices. The second example finds the shortest paths from *start B* to all other vertices, since no path goes from B to A, the distance is *Inf*.

Johnson's algorithm finds the shortest path between every pair of vertices in a sparse graph. Its time complexity is $O(V\ E\ log\ V)$.

```
> zz <- joh
> edgeWeights(zz)

$a
 b  e  c
 3 -4  8

$b
d e
1 7

$e
```

```
d
6

$c
b
4

$d
 c  a
-5  2
```

```
> johnson.all.pairs.sp(zz)

  a  b  e  c d
a 0  1 -4 -3 2
b 3  0 -1 -4 1
e 8  5  0  1 6
c 7  4  3  0 5
d 2 -1 -2 -5 0
```

This example uses a graph with negative edge weights.

The shortest paths between all pairs of vertices are presented in the matrix, entry [i, j] gives the distance from vertex i to vertex j. For example, the shortest path from vertex c to vertex d is of length 5; the shortest path from vertex a to vertex e is of length -4, since edge a->e is available and of distance -4; the shortest distance from a to c is -3.

Floyd-Warshall's algorithm finds the shortest path between every pair of vertices in a dense graph.

```
> floyd.warshall.all.pairs.sp(coex)

  A B C D E H F G
A 0 1 1 1 2 2 3 3
B 1 0 1 1 2 2 3 3
C 1 1 0 1 2 2 3 3
D 1 1 1 0 1 1 2 2
E 2 2 2 1 0 1 1 1
H 2 2 2 1 1 0 1 1
F 3 3 3 2 1 1 0 1
G 3 3 3 2 1 1 1 0
```

All edge distances are assumed to be 1, if not given. Since the graph is undirected, the distance matrix is symmetric, for example, distance from C to G is the same as that from G to C.

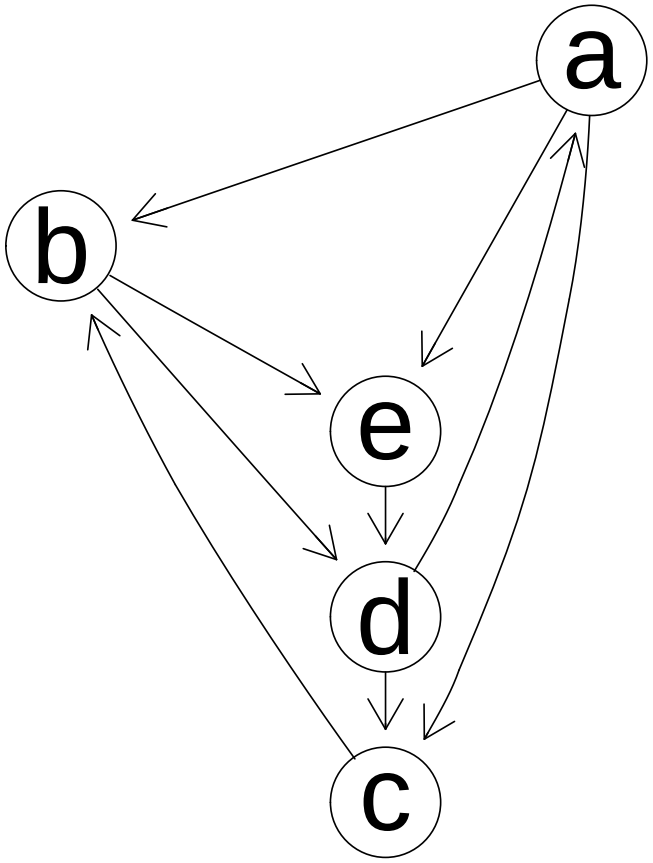

Figure 8: Example Johnson-all-pairs-shortest-paths example

3.4 Minimum spanning tree

Minimum-spanning-tree (MST) problem is to find a subset of edges that connect all the vertices, contains no cycles and have the minimum weight sum.

There are two algorithms available: *Kruskal's algorithm* and *Prim's algorithm*. Both are for undirected graphs with weighted edges, and both return a list of edges, weights and nodes determining MST.

The `mstree.kruskal` function finds the MST by Kruskal's algorithm.

```
> mstree.kruskal(km)

$edgeList
     [,1] [,2] [,3] [,4]
from "A"  "B"  "E"  "E"
to   "C"  "D"  "B"  "A"

$weights
       [,1] [,2] [,3] [,4]
weight    1    1    1    1

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

This graph is treated as undirected graph with corresponding weights. MST consists of 4 edges, *A->C, D->E, E->A, B->D*, each is of weight 1.

The `mstree.prim` function finds the MST by Prim's algorithm.

```
> mstree.prim(coex2)

$edgeList
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
from "A"  "A"  "A"  "A"  "D"  "C"  "D"  "E"
to   "A"  "B"  "C"  "D"  "E"  "G"  "H"  "F"

$weights
       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
weight    0    1    1    1    1    1    1    1

$nodes
[1] "A" "B" "C" "D" "E" "G" "H" "F"
```

The graph is treated as undirected graph with default weight 1. MST consists of 7 edges, *A->B, A->C, A->D, C->E, D->H, E->F, C->G*.

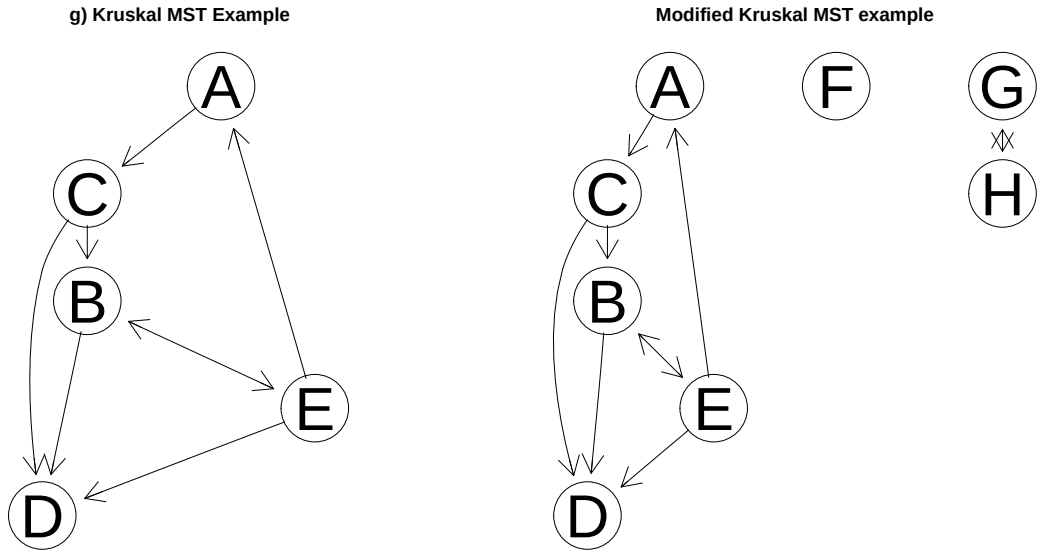

Figure 9: Kruskal MST examples.

3.5 Connected components

There are several algorithms available for this group of problems.

A *connected component* of an undirected graph is a subgraph that for any two vertices in this subgraph, u and v, there's a path from u to v.

The `connectedComp` function computes the connected components in an undirected graph.

```
> km1 <- km
> km1 <- graph::addNode(c("F","G","H"), km1)
> km1 <- addEdge("G", "H", km1, 1)
> km1 <- addEdge("H", "G", km1, 1)
> connectedComp(ugraph(km1))

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

$`2`
[1] "F"

$`3`
[1] "G" "H"
```

The original graph has one connected component. After we add three vertices, *F, G,*

H and an edge *G-H*, make the graph *undirected*, the modified graph has three connected components now.

A *strongly connected component* of a directed graph is a connected subgraph that for every pair of vertices in this subgraph, u and v, there are both a path from u to v and a path from v to u.

The `strongComp` function computes the strongly connected components in a directed graph.

```
> km2 <- km
> km2 <- graph::addNode(c("F","G","H"), km2)
> km2 <- addEdge("G", "H", km2, 1)
> km2 <- addEdge("H", "G", km2, 1)
> strongComp(km2)

$`1`
[1] "D"

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

$`3`
[1] "F"

$`4`
[1] "G" "H"
```

After adding three vertices, *F, G, H* and an edge *G-H*, there are three strong components in the graph now.

A *biconnected* graph is a connected graph that removal of any single vertex doesn't disconnect it. If the removal of a vertex increases the number of components in a graph, this vertex is call an *articulation point*.

The `biConnComp` function computes the biconnected components in an undirected graph. The `articulationPoints` function finds all the articulation points in an undirected graph.

```
> biConnComp(bicoex)

[[1]]
[1] "B" "C" "D"

[[2]]
[1] "A" "B" "F" "E"
```

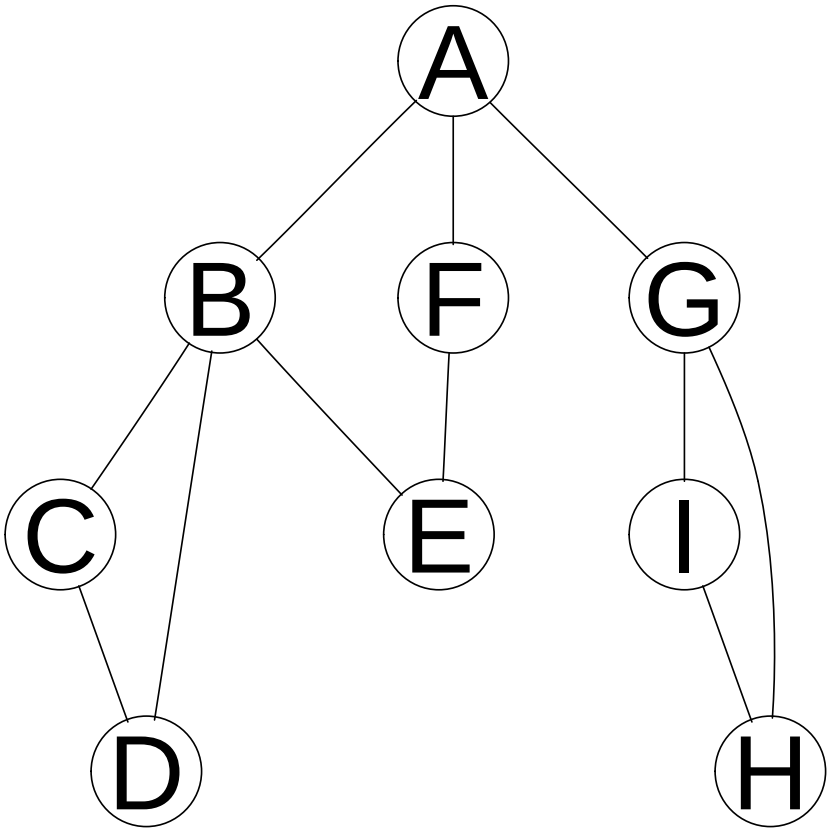

Figure 10: Biconnected components example from Boost library.

```
[[3]]
[1] "G" "H" "I"

[[4]]
[1] "A" "G"
```

```
> articulationPoints(bicoex)
```

```
[1] "B" "G" "A"
```

There are 4 biconnected components in the example: one with vertices *B, C, D* and edges *B-C, C-D, B-D* labeled 0, one with vertices *A, B, E, F* and edges *A-B, B-E, E-F, A-F* labeled 1, one with vertices *G, H, I* and edges *G-I, G-H, I-H* labeled 2, and one with vertices *A, G* and edges *A-G* labeled 3.

There are 3 articulation points in the example: *A, B, G*. It's easy to see removing any one of them will result in more connected components.

When you *add* edges to an undirected graph and want to get updated information on the connected components, you could use the following functions: `init.incremental.components` function to initialize the process; after adding edges to the graph, use `incremental.components` function to update the information on the connected components; use `same.component` function to find out if two vertices are in the same connected component.

Currently, only one incremental graph is allowed at any given time. To start on a new graph, you need to call `init.incremental.components` first.

```
> jcoex <- join(coex, hcs)
> x <- init.incremental.components(jcoex)
> incremental.components(jcoex)

[[1]]
no. of connected components
                          2

[[2]]
[1] "B" "A" "C" "D" "E" "H" "F" "G"

[[3]]
 [1] "A2" "A1" "A4" "A5" "Y"  "A3" "B1" "B2" "B3" "B4" "Z"  "X"

> same.component(jcoex, "A", "F")

[1] TRUE

> same.component(jcoex, "A", "A1")

[1] FALSE

> jcoex <- addEdge("A", "A1", jcoex)
> x <- init.incremental.components(jcoex)
> incremental.components(jcoex)

[[1]]
no. of connected components
                          1

[[2]]
 [1] "B"  "A"  "C"  "D"  "E"  "H"  "F"  "G"  "A1" "A2" "A4" "A5" "Y"  "A3" "B1"
[16] "B2" "B3" "B4" "Z"  "X"

> same.component(jcoex, "A", "A1")
```

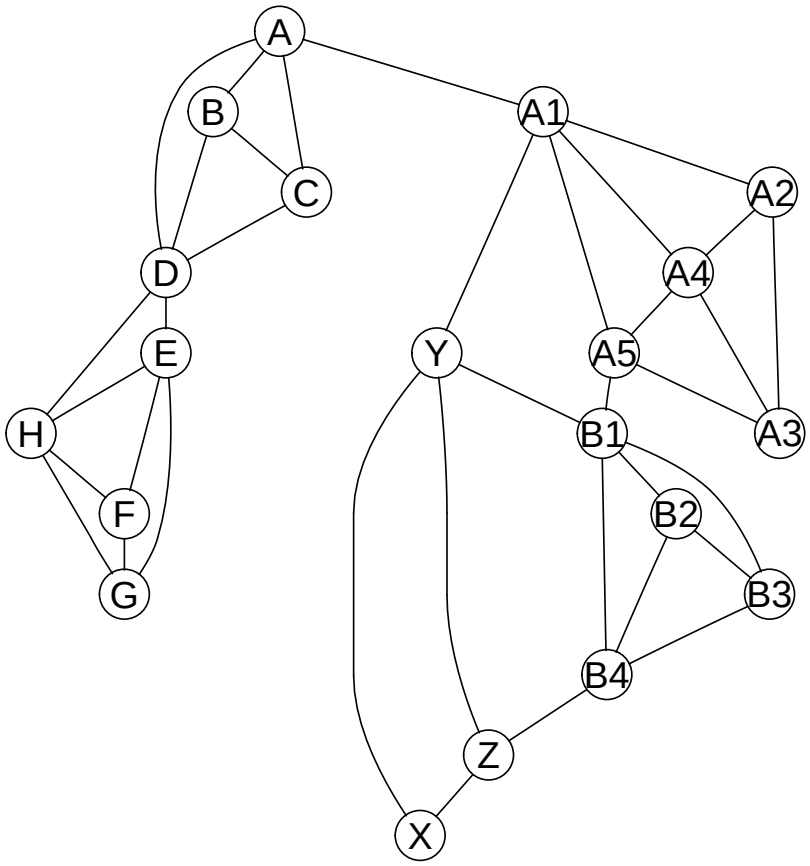

Figure 11: Example on incremental components: a graph connecting coex and hcs.

```
[1] TRUE
```

In the first part of this example, we join two separate graphs together, the resulting graph contains two connected components. Vertices *A* and *F* are in the same connected component, while vertices *A* and *A1* are not in the same connected component.

In the second part of the example, we add an edge connecting *A* and *X*, which effectively connects the two subgraphs, we have only one connected component left, which consists of all the vertices from the two original graphs, *A* and *A1* are in the same connected component now.

3.6 Maximum Flow

The functions, `edmonds.karp.max.flow` and `push.relabel.max.flow` are available to find the maximum flow between source and sink.

```
> edgeWeights(dijk)

$A
C
1

$B
B D E
2 1 2

$C
B D
7 3

$D
E
1

$E
A B
1 1

> edmonds.karp.max.flow(dijk, "B", "D")

$maxflow
[1] 2

$edges
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
from "A"  "B"  "B"  "B"  "C"  "C"  "D"  "E"  "E"
to   "C"  "B"  "D"  "E"  "B"  "D"  "E"  "A"  "B"

$flows
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
flow    1    0    1    1    0    1    0    1    0

> push.relabel.max.flow(dijk, "C", "B")

$maxflow
[1] 8

$edges
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
from "A"  "B"  "B"  "B"  "C"  "C"  "D"  "E"  "E"
to   "C"  "B"  "D"  "E"  "B"  "D"  "E"  "A"  "B"

$flows
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
flow    0    0    0    0    7    1    1    0    1
```

Call to `edmonds.karp.max.flow` finds the maximum flow of 2 from B to D: one part of flow 1 is B -> D directly, another part of flow 1 is B -> E -> A -> C -> D.

Call to `push.relabel.max.flow` find the maximum flow of 8 from C to B: one part of flow 7 is C -> B directly, another part of flow 1 is C -> D -> E -> B.

You can see the flow on each edge in the output, and each is no more than the capacity of the edge.

3.7 Sparse Matrix Ordering

There are three functions available in this category: `cuthill.mckee.ordering`, `minDegreeOrdering` and `sloan.ordering`.

Cuthill-McKee's algorithm tries to reduce the bandwidth of a graph by renumbering its vertices. The outputs are the vertices in the new ordering and reverse ordering.

Minimum degree Ordering is one approach that tries to reduce fill-ins in matrix reordering, which turns a system of equations $A\ x = b$ to $(P\ A\ PT)(P\ x) = P\ b$.

Sloan Ordering tries to reduce the profile and wavefront of a graph by renumbering its vertices.

```
> dijk1 <- ugraph(dijk)
> cuthill.mckee.ordering(dijk1)
```

```
$`reverse cuthill.mckee.ordering`
[1] "A" "B" "E" "C" "D"

$`original bandwidth`
[1] 4

$`new bandwidth`
[1] 3
```

```
> minDegreeOrdering(dijk1)

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

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

```
> sloan.ordering(dijk1)

$sloan.ordering
[1] "A" "E" "C" "B" "D"

$bandwidth
[1] 3

$profile
[1] 17

$maxWavefront
[1] 4

$aver.wavefront
[1] 2.6

$rms.wavefront
[1] 2.792848
```

TODO: EXPLAIN THESE OUTPUT.

3.8 Edge connectivity and minimum disconnecting set

For a single connected undirected graph, function *edgeConnectivity* calculates the minimum number of edges that have to be removed to create two disconnected components. No edge weight is taken into account and the output is the edges that need to be removed.

This is very similar to the *minCut* algorithm, which takes the edge weights into account when removing edges and outputs the vertices on the two disconnected components.

```
> edgeConnectivity(coex)

$connectivity
[1] 2

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

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

Mimimum of two edges must be removed to create two disconnected components: edges *D-E* and *D-H*.

3.9 Topological sort

The `tsort` function will return the names of vertices from a DAG in topological sort order.

```
> tsort(FileDep)

 [1] "zow_h"       "boz_h"       "zig_cpp"     "zig_o"       "dax_h"
 [6] "yow_h"       "zag_cpp"     "zag_o"       "bar_cpp"     "bar_o"
[11] "foo_cpp"     "foo_o"       "libfoobar_a" "libzigzag_a" "killerapp"
```

Note that if the input graph is not a DAG, BGL `topological_sort` will check this and throw 'not a dag'. This is crudely captured in the interface (a message is written to the console and zeroes are returned).

```
> FD2 <- FileDep
> # now introduce a cycle
> FD2 <- addEdge(from="bar_o", to="dax_h", FD2)
> tsort(FD2)

character(0)
```

3.10 Isomorphism

The `isomorphism` function determines if two graphs are isomorphism, i.e., determines if there is a one-to-one mapping *f* of vertices from one graph *g1* to the other *g2* such that edge *u -> v* is in *E(g1)* iff edge *f(u) -> f(v)* exists and is in *E(g2)*.

```
> isomorphism(dijk, coex2)

$isomorphism
[1] FALSE

> isomorphism(coex2i, coex2)

$isomorphism
[1] TRUE
```

The function handles both directed and undirected graphs. There are more vertices in graph *conn2* than *dijkstra*, so it's impossible to find a one-to-one mapping between them. One the other hand, graph *conn2i* is basically the same graph as *conn2* except the vertices have different names, so they are isomorphism.

3.11 Vertex Coloring

The `sequential.vertex.coloring` function assigns colors, as numbers 0, 1, 2, ..., to vertices in a graph so that two vertices connected by an edge are of different colors. It does not guarantee that minimum number of colors is used, and the result depends on the input ordering of the vertices in the graph.

```
> sequential.vertex.coloring(coex)

$`no. of colors needed`
[1] 4

$`colors of nodes`
A B C D E H F G
0 1 2 3 0 1 2 3
```

We need 4 colors for the vertices of this graph, one color scheme is to give color 0 to vertices *A, E*, color 1 to vertices *B, H*, color 2 to vertices *C, F* and color 3 to vertices *D, G*.

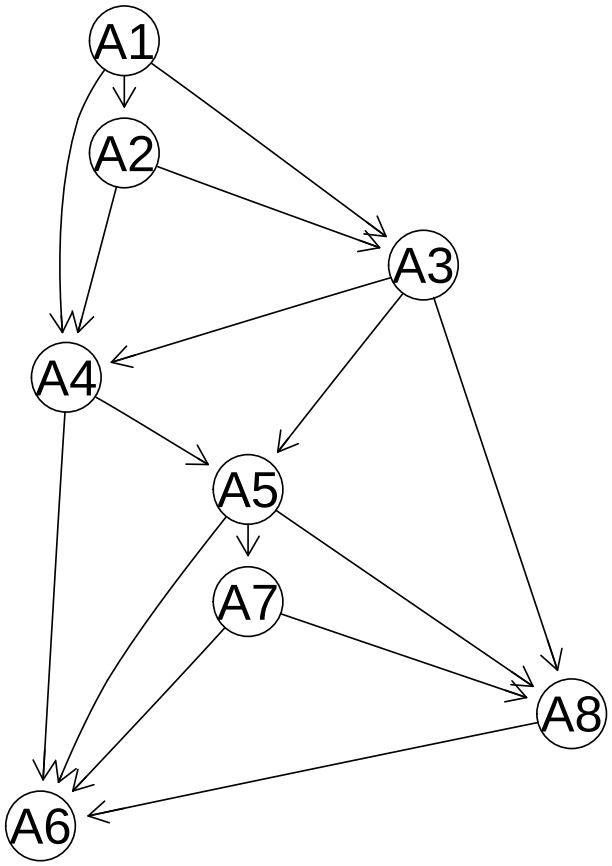

Figure 12: Example conn2i

3.12 Wavefront, Profiles

TODO: EXPLAIN THESE TERMS

The following functions are available: `ith.wavefront`, `maxWavefront`, `aver.wavefront` and `rms.wavefront`.

```
> ss <- 1
> ith.wavefront(dijk, ss)

$ith.wavefront
[1] 3

> maxWavefront(dijk)

$maxWavefront
[1] 4

> aver.wavefront(dijk)

$aver.wavefront
[1] 2.6

> rms.wavefront(dijk)

$rms.wavefront
[1] 2.792848
```

TODO: EXPLAIN THESE RESULTS

3.13 Betweenness Centrality and Clustering

Betweenness centrality of a vertex (or an edge) measures its importance in a graph, i.e., among all the shortest paths between every pair of vertices in the graph, how many of them have to go through this vertex (or edge). *Relative* betweenness centrality is calculated by scaling the *absolute* betweenness centrality by factor *2/((n-1)(n-2))*, where n is the number of vertices in the graph.

The `brandes.betweenness.centrality` function implements Brandes' algorithm in calculating betweenness centrality.

The `betweenness.centrality.clustering` function implements clustering in a graph based on edge betweenness centrality.

TODO: EXPLAIN MORE

```
> brandes.betweenness.centrality(coex)
> betweenness.centrality.clustering(coex, 0.1, TRUE)
```

This function has been temporarily withdrawn due to an assertion failure in the LLVM 9.0.0 C++ standard library. Users who are not using LLVM and wish to use it can reinstate it by:

1. Replace `src/bbc.cpp` with `inst/bbc.cpp`
2. Remove the `stop` command in line `R/interfaces.cpp`

4 Algorithms built on RBGL

4.1 Min-Cut

Given an undirected graph G=(V, E) of a single connected component, a *cut* is a partition of the set of vertices into two non-empty subsets S and V-S, a *cost* is the weight sum of edges that are incident on one vertex in S and one vertex in V-S. The min-cut problem is to find a cut (S, V-S) of minimum cost.

For simplicity, subset *S* is the smaller of the two.

```
> minCut(coex)

$mincut
[1] 2

$S
[1] "A" "B" "C" "D"

$`V-S`
[1] "E" "H" "F" "G"
```

Currently all edge weights are assumed to be 1, minimum cut is of weight 2, it will partition the graph into two subsets: subset *A, B, C, D* and subset *E, H, F, G*.

4.2 highlyConnSG

A graph *G* with *n* vertices is highly connected if its connectivity *k(G) > n/2*. Function *highlyConnSG* partitions a graph into a set of highly connected subgraphs, by using minimum-cut algorithm repeatedly. To improve performance, it takes special care of singletons, low degree vertices and merges clusters.

```
> highlyConnSG(coex)
```

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

$clusters[[2]]
[1] "E" "H" "F" "G"
```

```
> highlyConnSG(hcs)
```

```
$clusters
$clusters[[1]]
[1] "A1" "A2" "A4" "A5" "A3"

$clusters[[2]]
[1] "B1" "B2" "B3" "B4"

$clusters[[3]]
[1] "Y" "Z" "X"
```

In graph *conn*, two highly-connected-subgraphs are found: subgraph with vertices *A, B, C, D* and subgraph with vertices *E, H, F, G*.

In graph *hcs*, 3 highly-connected-subgraphs are found: subgraph with vertices *A1, A2, A3, A4, A5*, subgraph with vertices *B1, B2, B3, B4* and subgraph with vertices *X, Y, Z*.

5 Algorithms independent from RBGL

5.1 maxClique

A *clique* is a complete subgraph, i.e., there is an edge between every pair of vertices.

Maximum Clique problem is to find the largest clique in a graph. This problem is NP-complete, which means it cannot be solved by any known polynomial algorithm.

Function *maxClique* implements the algorithm from *Finding all cliques of an undirected graph*, by C. Bron and J. Kerbosch (CACM, Sept 1973, Vol 16, No. 9.), which finds all the cliques in a graph.

```
> maxClique(coex)
```

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

$maxCliques[[2]]
```

```
[1] "D" "E" "H"

$maxCliques[[3]]
[1] "F" "E" "H" "G"

> maxClique(hcs)

$maxCliques
$maxCliques[[1]]
[1] "B1" "B2" "B3" "B4"

$maxCliques[[2]]
[1] "B1" "Y"

$maxCliques[[3]]
[1] "B1" "A5"

$maxCliques[[4]]
[1] "A2" "A4" "A3"

$maxCliques[[5]]
[1] "A2" "A4" "A1"

$maxCliques[[6]]
[1] "A4" "A5" "A3"

$maxCliques[[7]]
[1] "A4" "A5" "A1"

$maxCliques[[8]]
[1] "A1" "Y"

$maxCliques[[9]]
[1] "Z" "Y" "X"

$maxCliques[[10]]
[1] "Z"  "B4"
```

In graph *conn*, 3 cliques are found: clique with vertices *D, B, C, A*, clique with vertices *D, E, H* and clique with vertices *F, E, H, H*.

In graph *hcs*, 10 cliques are found. For instance, vertices *A2, A4, A3* form a clique, vertices *B1, Y* form a clique.

5.2 is.triangulated

A graph is *triangulated* if all cycles of length 4 or more have a chord. The `is.triangulated` function returns TRUE or FALSE, accordingly.

We implemented the following algorithm from *Combinatorial Optimization: algorithms and complexity* (p. 403) by C. H. Papadimitriou, K. Steiglitz: G is chordal iff either G is an empty graph, or there is a v in V such that (i) the neighborhood of v, i.e., v and its adjacent vertices, forms a clique, and (ii) recursively, G-v is chordal.

```
> is.triangulated(coex)

[1] TRUE

> is.triangulated(hcs)

[1] FALSE
```

5.3 separates

Function *separates* determines if a subset of vertices separates two other subsets of vertices, and returns TRUE or FALSE, accordingly.

```
> separates("B", "A", "E", km)

[1] TRUE

> separates("B", "A", "C", km)

[1] FALSE
```

5.4 kCores

A *k-core* in a graph is a subgraph where each vertex is adjacent to at least `k` other vertices in the same subgraph.

Function *kCores* finds all the k-cores in a graph. It returns the core numbers for all the nodes in the graph. When the given graph is directed, you can choose whether in-degree, out-degree or both should be considered.

The k-core of a graph is not a necessarily connected subgraph. If i > j, the i-core of a graph contains the j-core of the same graph.

The implementation is based on the algorithm by V. Batagelj and M. Zaversnik, 2002.

```
> kCores(kcoex)
```

```
A C B E F D G H J K I L M N O P Q R S T U
1 2 1 2 3 3 3 3 3 3 3 3 2 2 1 1 2 2 2 2 0

> kcoex2 <- coex2
> kCores(kcoex2)

A B C D E G H F
3 3 3 3 3 3 3 3

> kCores(kcoex2, "in")

A B C D E G H F
0 0 0 0 0 0 0 0

> kCores(kcoex2, "out")

A B C D E G H F
0 0 0 0 0 0 0 0

> g1 <- addEdge("C", "B", kcoex2)
> kCores(g1, "in")

A B C D E G H F
0 1 1 1 1 1 1 1

> g2 <- addEdge("C", "A", kcoex2)
> kCores(g2, "out")

A B C D E G H F
1 1 1 0 0 0 0 0
```

The example on directed graph, "conn2", turns out to be a waterfall-like graph. If we order the nodes as: A, B, C, D, E, F, H, G, all the edges go in the same direction, i.e., i -> j, i < j.

Let's consider in-degree-only case: A has no in-edge so it is 0-core; after you eliminate A, no in-edge to B, so B is 0-core; continue this, we could see that there's no subset of nodes that each and every single node has 1 in-degree. Therefore, they are all of 0-core.

For out-degree-only case: G has no out-edge, so it's 0-core; after eliminating G, F has no out-edge, so F is 0-core; continue this process, we could see that there's no subset of nodes that each and every single node has 1 out-edge. Therefore, they are all of 0-core.

If we add edge(s) to break the waterfall-like property, C->B, C->A, separately, we could see the changes in the core numbers that are consistant with the analysis above.

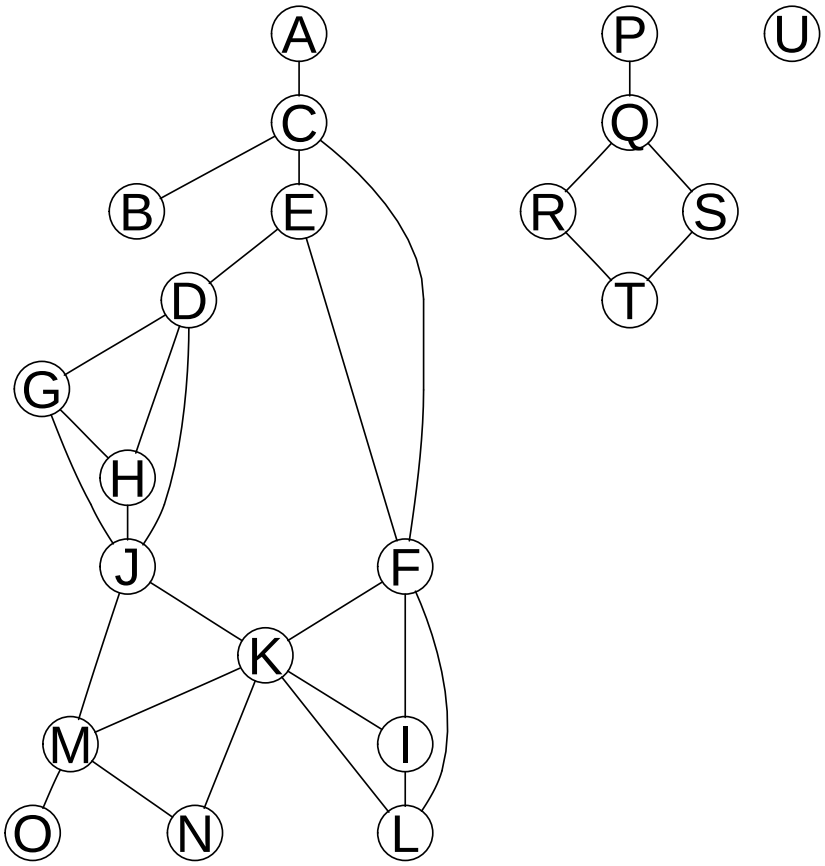

Figure 13: K-cores Example.

5.5 kCliques

In social network analysis, a k-cliques is a maximum subgraph that the shortest distance between any two nodes is no more than k.

Function *kCliques* finds all the k-cliques in an undirected graph (k = 1, ..., N, where N is the length of the longest shortest-path). It returns all the k-cliques.

Let D be a matrix, D[i][j] is the shortest path from node i to node j. Algorithm is outlined as following. o. use Johnson's algorithm to fill D; let N = max(D[i][j]) for all i, j; o. each edge is a 1-clique by itself; o. for k = 2, ..., N, try to expand each (k-1)-clique to k-clique: o. consider a (k-1)-clique the current k-clique KC; o. repeat the following: if for all nodes j in KC, D[v][j] <= k, add node v to KC; o. eliminate duplicates; o. the whole graph is N-clique.

```
> kCliques(kclex)

$`1-cliques`
$`1-cliques`[[1]]
[1] "1" "2" "3"

$`1-cliques`[[2]]
[1] "2" "4"

$`1-cliques`[[3]]
[1] "3" "5"

$`1-cliques`[[4]]
[1] "4" "6"

$`1-cliques`[[5]]
[1] "5" "6"


$`2-cliques`
$`2-cliques`[[1]]
[1] "1" "2" "3" "4" "5"

$`2-cliques`[[2]]
[1] "2" "3" "4" "5" "6"


$`3-cliques`
$`3-cliques`[[1]]
[1] "1" "2" "3" "4" "5" "6"
```

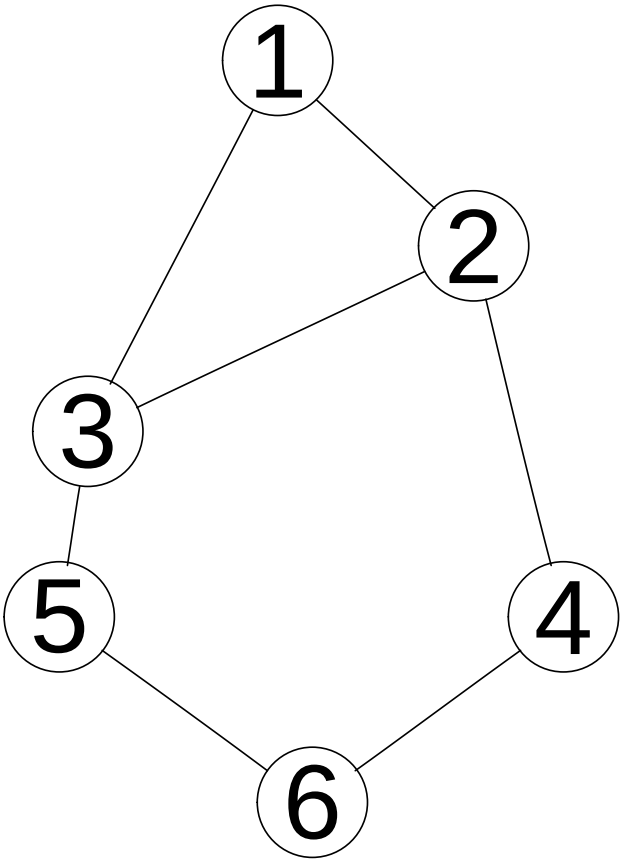

Figure 14: K-cliques Example.