

# *RBGL*: R interface to boost graph library

VJ Carey `stvjc@channing.harvard.edu`

October 20, 2003

*Summary.* A very preliminary implementation of an interface from R to the Boost Graph Library (BGL, an alternative to STL programming for mathematical graph objects) is presented. *This 2003 update employs the graph class of Bioconductor.*

## Contents

<b>1</b>	<b>Working with the Bioconductor graph class</b>	<b>1</b>
<b>2</b>	<b>Algorithms supported by RBGL</b>	<b>3</b>
2.1	Topological sort . . . . .	3
2.2	Kruskal's minimum spanning tree . . . . .	3
2.3	Depth first search . . . . .	5
2.4	Breadth first search . . . . .	6
2.5	Dijkstra's shortest paths . . . . .	6
2.6	Connected components . . . . .	8
2.7	Strongly connected components . . . . .	9
2.8	Edge connectivity and minimum disconnecting set . . . . .	9

## 1 Working with the Bioconductor graph class

An example object representing file dependencies is included, as shown in Figure 1.

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

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

Figure 1: File dependency digraph example from Boost library.

## 2 Algorithms supported by RBGL

### 2.1 Topological sort

The `tsort` function will return the indices of vertices in topological sort order:

```
> ts <- tsort(FileDep)
> print(nodes(FileDep)[ts + 1])

[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@edgeL[["bar_cpp"]]$edges <- c(8,1)
#tsort(FD2)
```

### 2.2 Kruskal's minimum spanning tree

Function `mstree.kruskal` just returns a list of edges, weights and nodes determining the minimum spanning tree (MST) by Kruskal's algorithm.

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

$edgeList
      [,1] [,2] [,3] [,4]
[1,]     1     4     5     2
[2,]     3     5     1     4

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

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

Figure 2: Kruskal MST example from Boost library.

Figure 3: DFS example from Boost library.

## 2.3 Depth first search

The `dfs` function returns a list of node indices by discovery and finish order.

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

```
$discovered
[1] 1 2 5 4 3 6
```

```
$finish
[1] 4 5 2 1 6 3
```

Here is the list of nodes in DFS discovery order.

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

## 2.4 Breadth first search

The `bfs` function returns a vector of node indices for a breadth-first search (BFS) starting at the node indexed by `init.ind`.

```
> bf <- fromGXL(file(system.file("XML/bfsex.gxl", package = "RBGL")))
> bf@edgemode <- "undirected"
> print(o <- bfs(bf, init.ind = 2))

[1] 2 6 1 3 7 5 4 8
```

The nodes in BFS order starting with the second node are

```
> print(nodes(bf)[o])

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

## 2.5 Dijkstra's shortest paths

```
> dd <- fromGXL(file(system.file("XML/dijkex.gxl", package = "RBGL")))
> print(dijkstra.sp(dd))
```

\$distances

A B C D E

0 6 1 4 5

\$penult

[1] 1 5 1 3 4

\$start

[1] 1

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

> dijkstra.sp(ospf, 6)
```

\$distances

RT1	RT2	RT3	RT4	RT5	RT6	RT7	RT8	RT9	RT10	RT11	RT12	N1	N2	N3	N4
7	7	6	7	6	0	8	8	11	7	10	11	10	10	7	8
N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	H1					
8	12	10	11	13	14	10	14	14	17	21					

\$penult

[1] 15 15 6 15 6 6 17 17 20 6 19 20 1 2 3 3 10 8 10 11 12 9 7 5 5

Figure 4: Network example from BGL.

```
[26] 7 12
```

```
$start
```

```
[1] 6
```

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

```
$length
```

```
RT1
```

```
7
```

```
$path
```

```
[1] "RT6" "RT3" "N3" "RT1"
```

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

```
> print(dijkstra.sp(dd))
```

```
$distances
```

```
A B C D E
```

```
0 6 1 4 5
```

```
$penult
```

```
[1] 1 5 1 3 4
```

```
$start
```

```
[1] 1
```

## 2.6 Connected components

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

```
> km@nodes <- c(km@nodes, "F", "G", "H")
```

```
> km@edgeL$F <- list(edges = numeric(0))
```

```
> km@edgeL$G <- list(edges = 8)
```

```
> km@edgeL$H <- list(edges = 7)
```

```
> km@edgemode <- "undirected"
```

```
> if (length(agrep("solaris", version[["platform"]])) == 0) print(connectedComp(km,
```

```
$"1"
```

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

```
$"2"
```

```
[1] "F"
```

```
$"3"
```

```
[1] "G" "H"
```



## 2.7 Strongly connected components

```
> km <- fromGXL(file(system.file("GXL/kmstEx.gxl", package = "graph")))
> km@nodes <- c(km@nodes, "F", "G", "H")
> km@edgeL$F <- list(edges = numeric(0))
> km@edgeL$G <- list(edges = 8)
> km@edgeL$H <- list(edges = 7)
> km@edgemode <- "directed"
> print(strongComp(km))
```

```
$"1"
```

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

```
$"2"
```

```
[1] "F"
```

```
$"3"
```

```
[1] "G" "H"
```

## 2.8 Edge connectivity and minimum disconnecting set

```
> coex <- fromGXL(file(system.file("XML/conn.gxl", package = "RBGL")))
> dcoex <- coex
> dcoex@edgemode <- "directed"
> udcoex <- ugraph(dcoex)
```

```
> if (length(agrep("solaris", version[["platform"]])) == 0) print(edgeConnectivity
```

```
$connectivity
```

```
[1] 2
```

```
$minDisconSet
```

```
$minDisconSet[[1]]
```

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

```
$minDisconSet[[2]]
```

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

Figure 5: Edge connectivity example.