# *RBGL*: R interface to boost graph library

VJ Carey `stvjc@channing.harvard.edu`

October 9, 2002

*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.

## 1 Working with a simple S4 graph class

The *RBGL* package includes an S4 class for representing directed graphs by vertex and edge lists. The class is called `graf` and an example object representing file dependencies is included, as shown in Figure 1. The figure is made with the `toDot` function, which will work on any system where the ATT graphViz `dot` utility is available.

```
> library(RBGL)

Loading required package: methods

> data(FileDep)
> print(FileDep)

graf object:
vertices:
dax_h yow_h boz_h zow_h foo_cpp foo_o bar_cpp bar_o libfoobar_a zig_cpp zig_o
zag_cpp zag_o libzigzag_a killerapp
edges: [ 19 ]

> x <- toDot(FileDep, outDotFile = "fd.dot")
> try(system("dot -Tps fd.dot > fd.ps"))
```

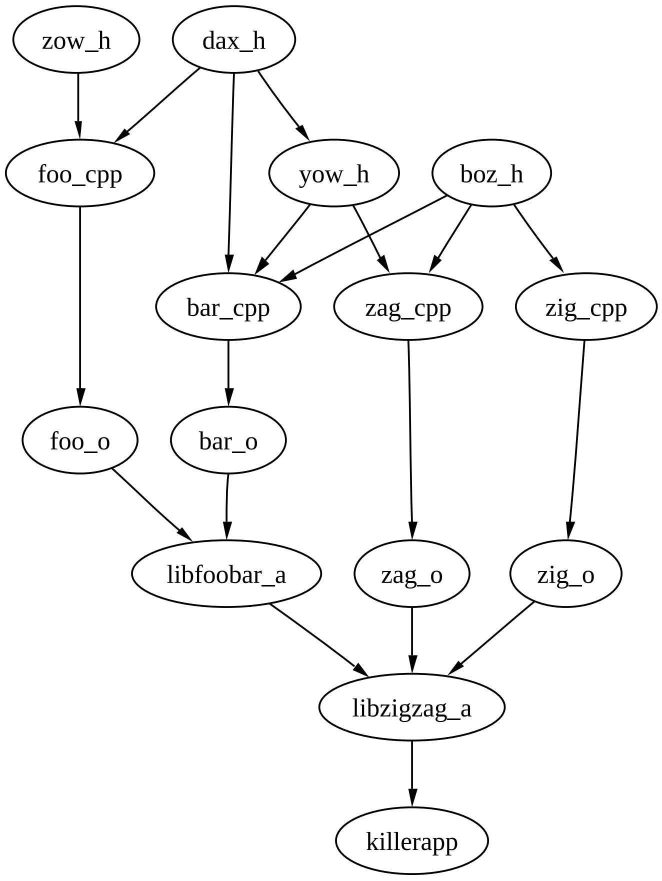

Figure 1: File dependency digraph example from Boost library.

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

```
> print(ts <- tsort(FileDep))
```

```
 [1]  3  2  9 10  0  1 11 12  6  7  4  5  8 13 14
```

```
> print(vertices(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
> FD2@edges[[20]] <- c("bar_cpp", "dax_h")
> print(tsort(FD2))
```

```
not a dag, returning zeroes
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

## 2 Dealing with edge attributes

Currently there is a crude mechanism for attaching edge costs.

```
> FD3 <- FileDep
> setEdgeCosts(FD3) <- runif(19)
> print(FD3)
```

```
graf object:
vertices:
dax_h yow_h boz_h zow_h foo_cpp foo_o bar_cpp bar_o libfoobar_a zig_cpp zig_o
zag_cpp zag_o libzigzag_a killerapp
edges: [ 19 ]

 edge costs range from  0.01465037  to  0.932525
```

```
> print(edgeCosts(FD3))
```

```
 [1] 0.34179487 0.39442251 0.10576390 0.01465037 0.59775850 0.14678464
 [7] 0.43600017 0.17418395 0.47225331 0.49640122 0.63827725 0.67390938
[13] 0.93252505 0.35623545 0.05537406 0.76364631 0.08856254 0.47978769
[19] 0.11458722
```

## 3 Quick look at RBGL with *sna* rgraph

The *sna* package on CRAN uses an S3 class called `graph`, based on an adjacency matrix representation. Adjacency matrices are converted to adjacency list representation using `Am2Al` in *RBGL*.

```
> library(sna)
```

Make a random graph with 5 vertices:

```
> set.seed(123)
> GG <- rgraph(5, 1)
> print(GG)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    1    0    1
[2,]    1    0    0    0    0
[3,]    0    0    0    1    1
[4,]    0    0    0    0    0
[5,]    0    1    1    0    0
```

Print the 'list form':

```
> print(GGl <- Am2Al(GG))
```

```
  [,1] [,2] [,3] [,4] [,5] [,6] [,7]
x    1    4    0    4    2    0    2
y    0    1    2    2    3    4    4
```

Build the `graf` analog, then try a topological sort:

```
> GGE <- list()
> for (i in 1:ncol(GGl)) GGE[[i]] <- GGl[, i]
> uan <- function(x) sort(unique(as.numeric(x)))
> GGV <- uan(GGl)
> sGG <- new("graf", edges = as.edgeStruct(GGE), vertices = GGV)
> toDot(sGG, outDotFile = "sGG.dot")
```

```
[1] "dot file written to sGG.dot  use 'dot -Tps [.dot] [.ps] to render"
```

```
> try(system("dot -Tps sGG.dot > sGGd.ps"))
> print(sGG)
```

```
graf object:
vertices:
0 1 2 3 4
edges: [ 7 ]
```

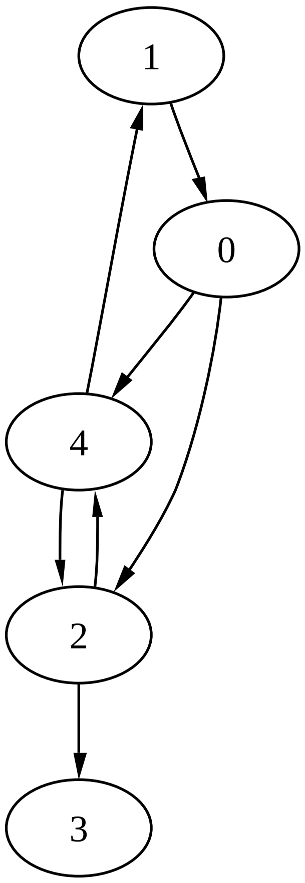

Figure 2: Random graph.

```
> print(tsort(sGG))

not a dag, returning zeroes
[1] 0 0 0 0 0
```

We can see why the topological sort fails:

Let's define a converter for *sna* graphs to objects of class `graf`:

```
> snag2graf <- function(x) {
+     m <- Am2Al(x)
+     o <- list()
+     for (i in 1:ncol(m)) o[[i]] <- m[, i]
+     v <- sort(unique(as.numeric(m)))
+     new("graf", edges = as.edgeStruct(o), vertices = v)
+ }
```

Now let's make a big random graph:

```
> rg <- rgraph(14, 1)
> gg <- snag2graf(rg)
> toDot(gg, outDotFile = "gg.dot")

[1] "dot file written to gg.dot  use 'dot -Tps [.dot] [.ps] to render"

> try(system("dot -Tps gg.dot > ggd.ps"))
```

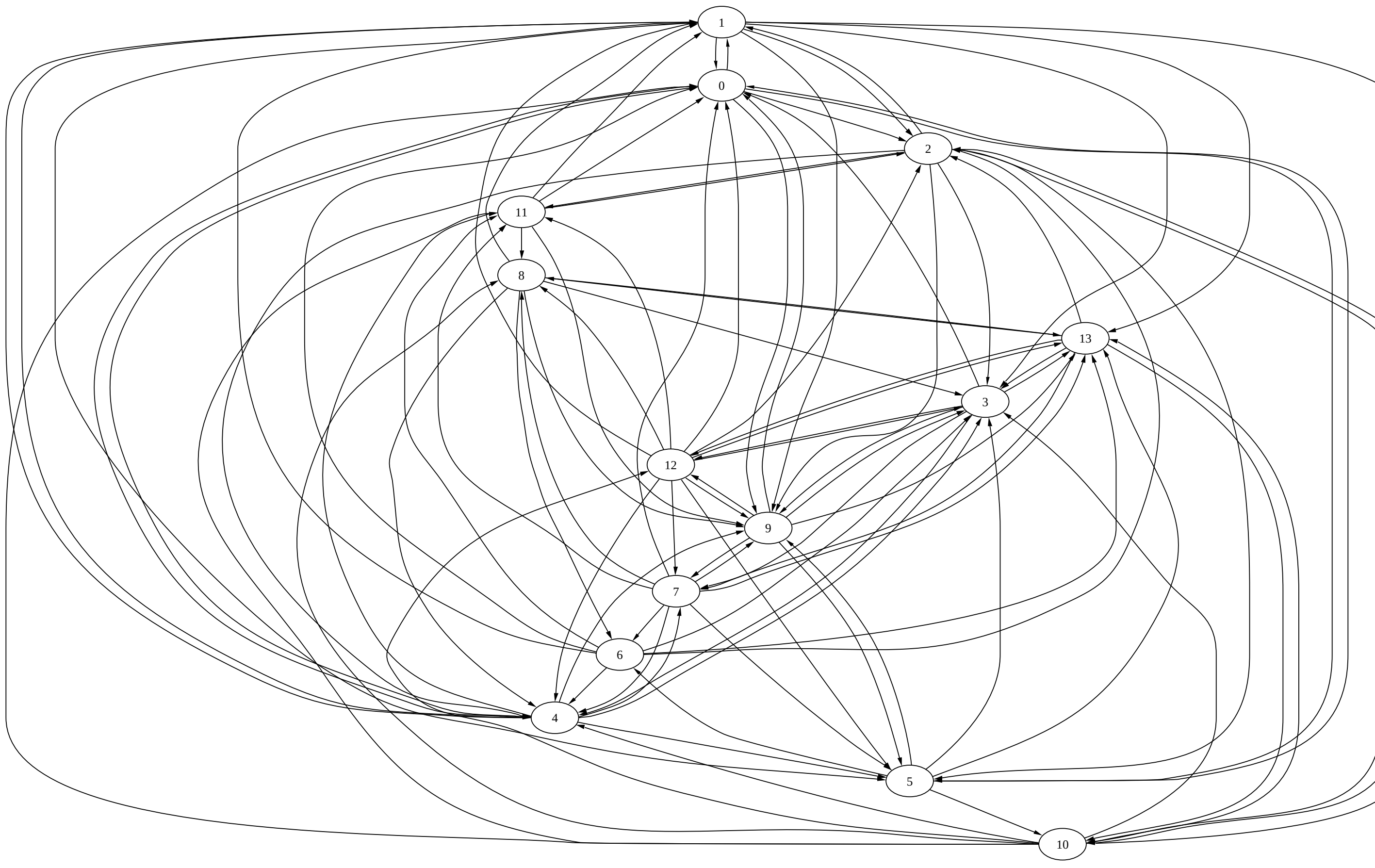

Figure 3: Large random graph from sna.

## 4 Rendering application: looking at GO

The GO data package is an example of a DAG. We should be able to usefully visualize aspects of this using our graf class.

We'll begin with the visualization of the cellular component hierarchy:

```
> library(GO)
> cc <- ls(env = GOCCPARENTS)
> print(cc[1])
```

```
[1] "GO:0000015"
```

We see that the environment consists of alphanumeric GO accession numbers. Each accession number is bound to a vector of its parents in the DAG:

```
> print(get(cc[1], env = GOCCPARENTS))
```

```
[1] "GO:0005829"
```

```
> nacc <- length(cc)
> allccp <- list()
> for (i in 1:nacc) allccp[[i]] <- get(cc[i], env = GOCCPARENTS)
```

We now have a vector of accession numbers and a conforming list of vectors of parents. We can construct a (possibly redundant) list of edges in the graph as follows:

```
> alled <- list()
> k <- 1
> for (i in 1:nacc) {
+     for (j in 1:length(allccp[[i]])) {
+         alled[[k]] <- c(cc[i], allccp[[i]][j])
+         k <- k + 1
+     }
+ }
```

We will obtain the unique set of edges by some character manipulations, then build the associated graph for rendering.

Now we get the edges and nodes of the associated tree:

```
> upairs <- function(x) {
+     xp <- lapply(x, function(x) paste(x, collapse = ";"))
+     uxp <- unique(unlist(xp))
+     strsplit(uxp, ";")
+ }
> GOEdges <- upairs(alled)
> GONodes <- unique(cc)
> gog <- makeGraf(V = GONodes, E = GOEdges)
> toDot(gog, "gog.dot")
```

```
[1] "dot file written to gog.dot  use 'dot -Tps [.dot] [.ps] to render"
```

After running the unix command

```
dot -Gsize=8,8 -Grotate=90 -Tps -Nfontsize=1
          -Nfixedsize=true -Nheight=.2 -Nwidth=.2
          -Gratio=fill gog.dot > gog.ps
```

We can see the basic topology of the tree.

Wrapping this all together, we can generate graf objects from any GO environment:

```
> GO2graf <- function(env) {
+     cc <- ls(env = env)
+     nacc <- length(cc)
+     allccp <- list()
+     for (i in 1:nacc) allccp[[i]] <- get(cc[i], env = env)
+     alled <- list()
+     k <- 1
+     for (i in 1:nacc) {
+         for (j in 1:length(allccp[[i]])) {
+             alled[[k]] <- c(cc[i], allccp[[i]][j])
+             k <- k + 1
+         }
+     }
+     upairs <- function(x) {
+         xp <- lapply(x, function(x) paste(x, collapse = ";"))
+         uxp <- unique(unlist(xp))
+         strsplit(uxp, ";")
+     }
+     GOEdges <- upairs(alled)
+     GONodes <- unique(cc)
+     makeGraf(V = GONodes, E = GOEdges)
+ }
```

That finishes the function. Now apply it to two more environments:

```
> mfg <- GO2graf(GOMFPARENTS)
> bpg <- GO2graf(GOBPPARENTS)
```

And render:

```
> toDot(mfg, "mf.dot")
```

```
[1] "dot file written to mf.dot  use 'dot -Tps [.dot] [.ps] to render"
```

```
> toDot(bpg, "bp.dot")
```

```
[1] "dot file written to bp.dot  use 'dot -Tps [.dot] [.ps] to render"
```

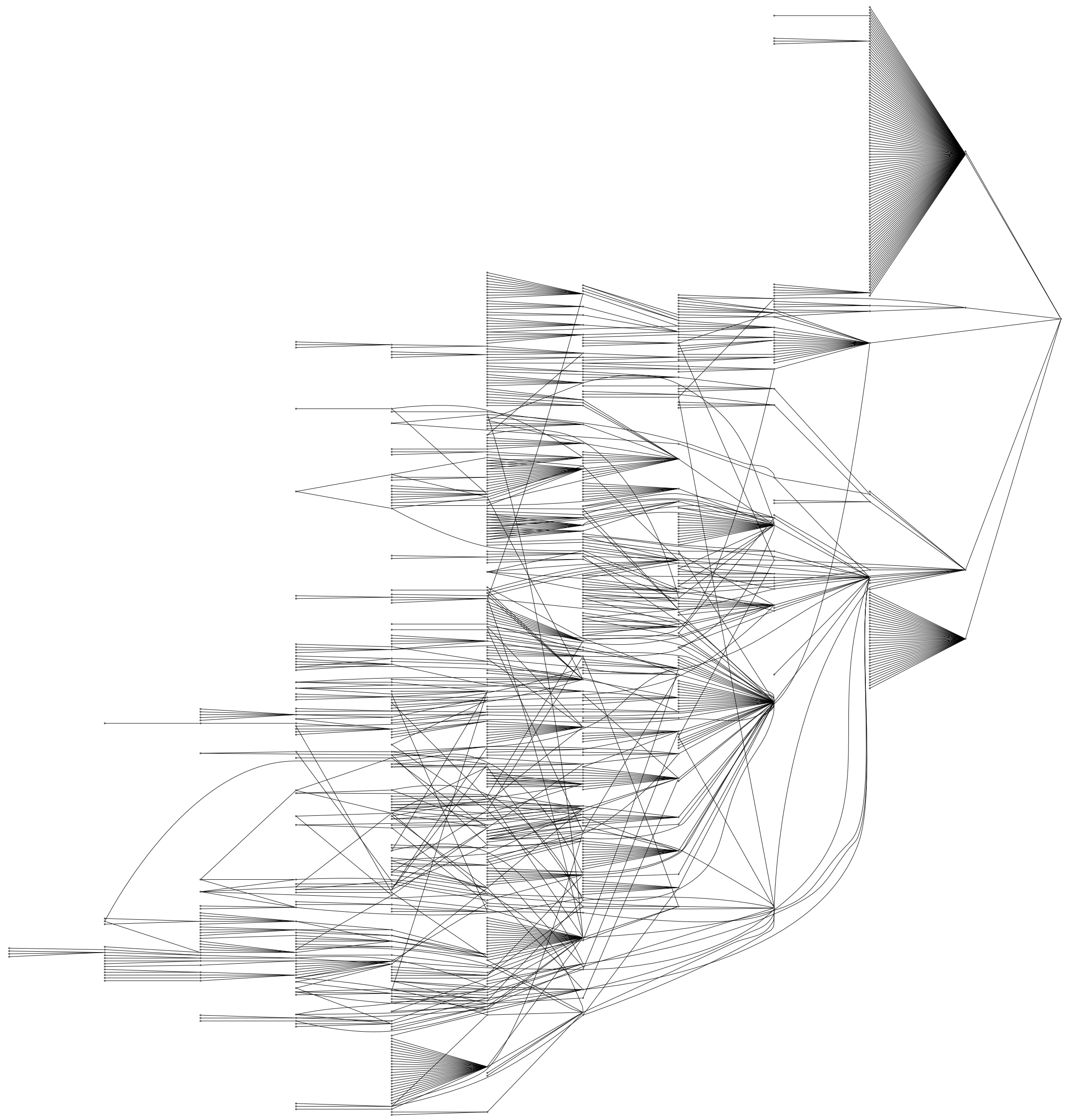

Figure 4: Cellular component, based on 4 Sep GO data package

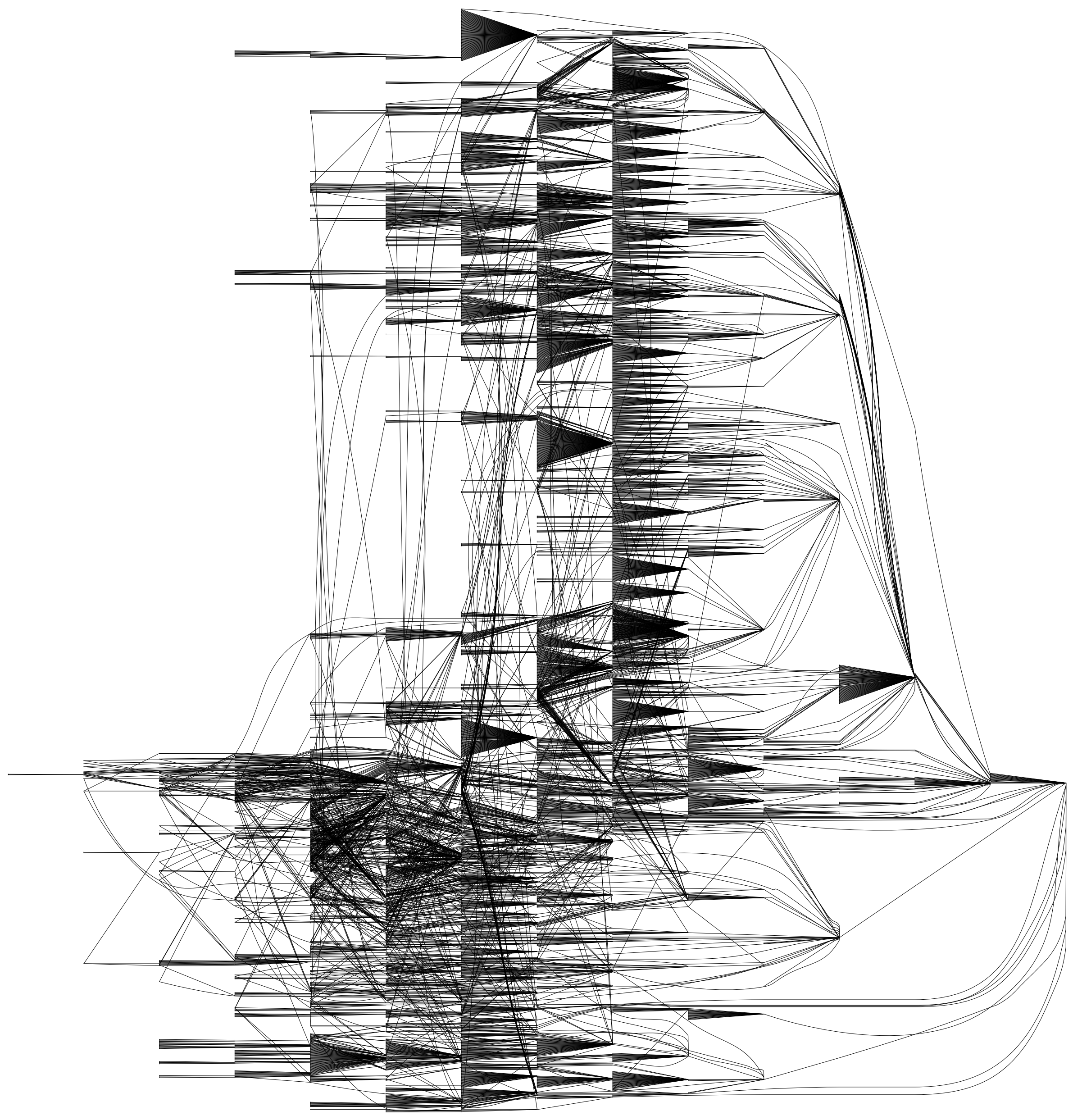

Figure 5: Molecular function, based on 4 Sep GO data package

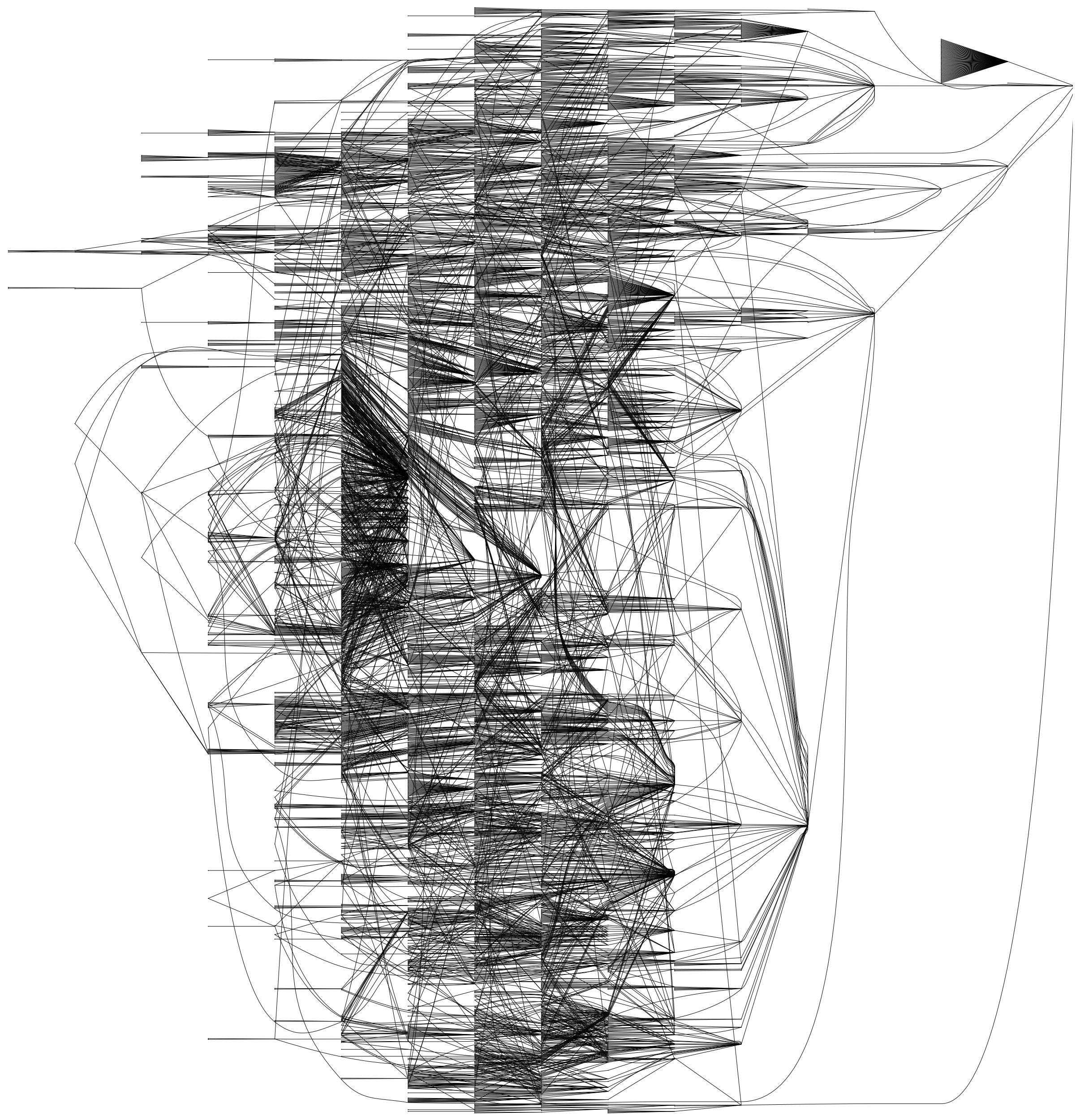

Figure 6: Biological process based on 4 Sep GO data package