RBGL: R interface to boost graph library

VJ Carey `stvjc@channing.harvard.edu`

September 4, 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.

```
R> library(RBGL)

Loading required package: methods

R> data(FileDep)
R> 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 ]

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

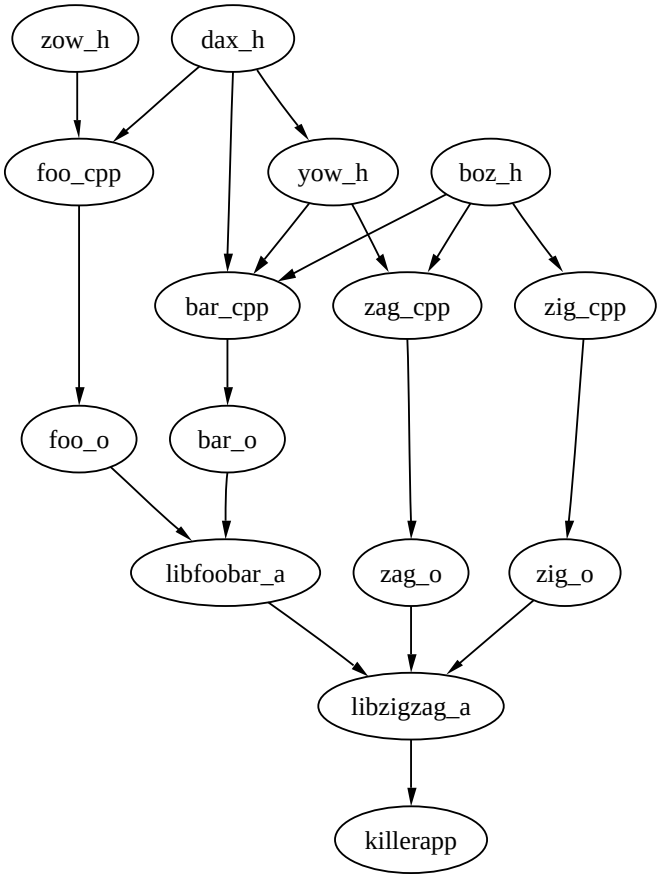

Figure 1: File dependency digraph example from Boost library.

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

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

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

R> 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):

```
R> FD2 <- FileDep
R> FD2@edges[[20]] <- c("bar_cpp", "dax_h")
R> 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.

```
R> FD3 <- FileDep
R> setEdgeCosts(FD3) <- runif(19)
R> 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  0.009236642  to  0.9688583

R> print(edgeCosts(FD3))

 [1] 0.019031147 0.846534026 0.271073071 0.752639341 0.928497212 0.477670186
 [7] 0.178178137 0.290539062 0.605881053 0.968858309 0.009236642 0.828361163
[13] 0.419870809 0.302664886 0.203459006 0.345007322 0.464891253 0.695514675
[19] 0.603390695
```

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*.

```
R> library(sna)
```

Make a random graph with 5 vertices:

```
R> set.seed(123)
R> GG <- rgraph(5, 1)
R> 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':

```
R> 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:

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

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

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

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

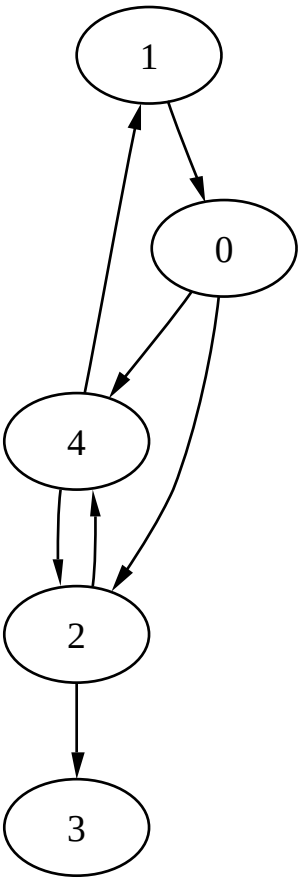

Figure 2: Random graph.

```
R> 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`:

```
R> 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:

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

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

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

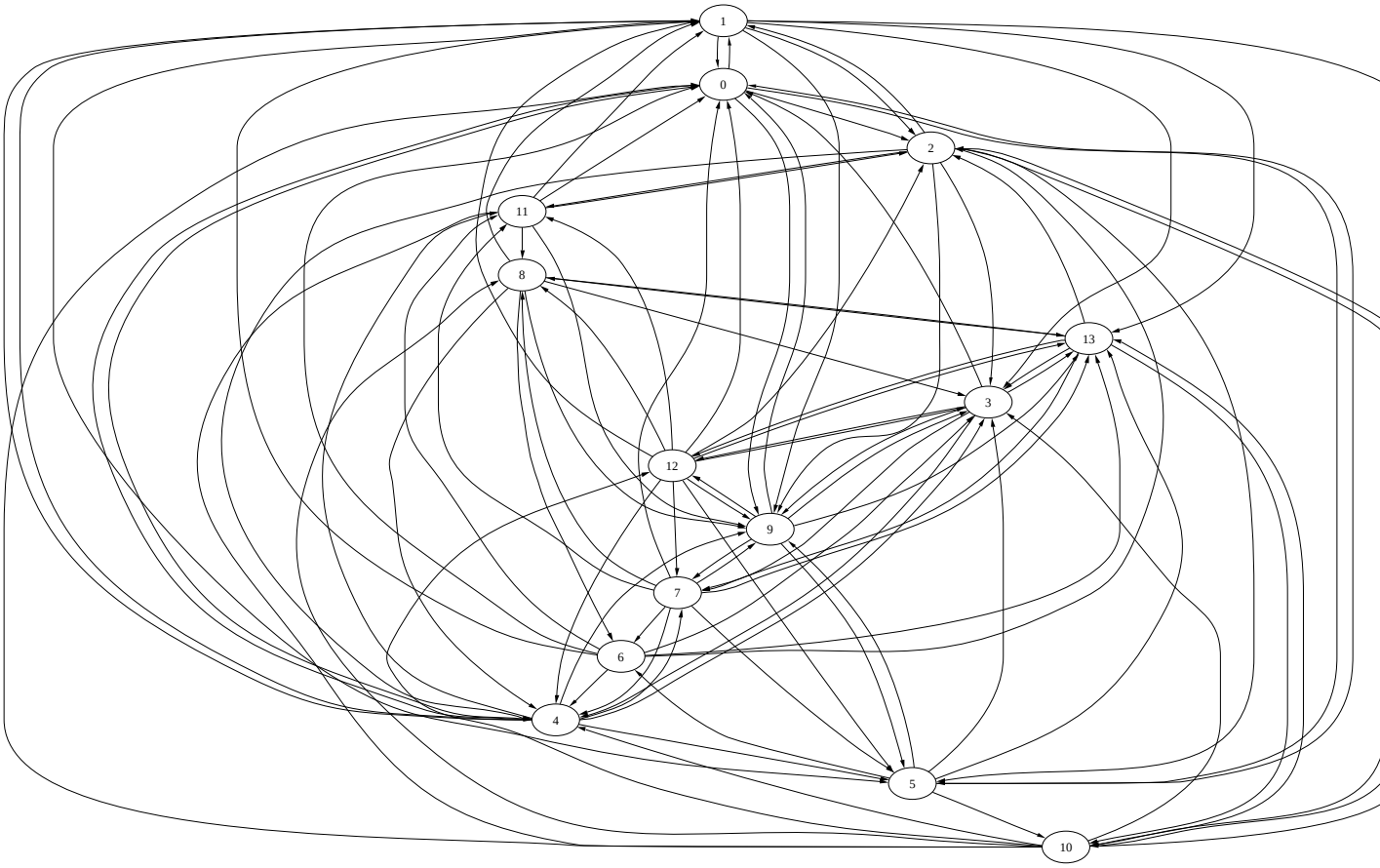

Figure 3: Large random graph from sna.

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:

```
R> library(GO)
R> cc <- ls(env = GOCCPARENTS)
R> 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:

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

[1] "GO:0005829"

R> nacc <- length(cc)
R> allccp <- list()
R> 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:

```
R> alled <- list()
R> k <- 1
R> 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:

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

```
R> 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:

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

And render:

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

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

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

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

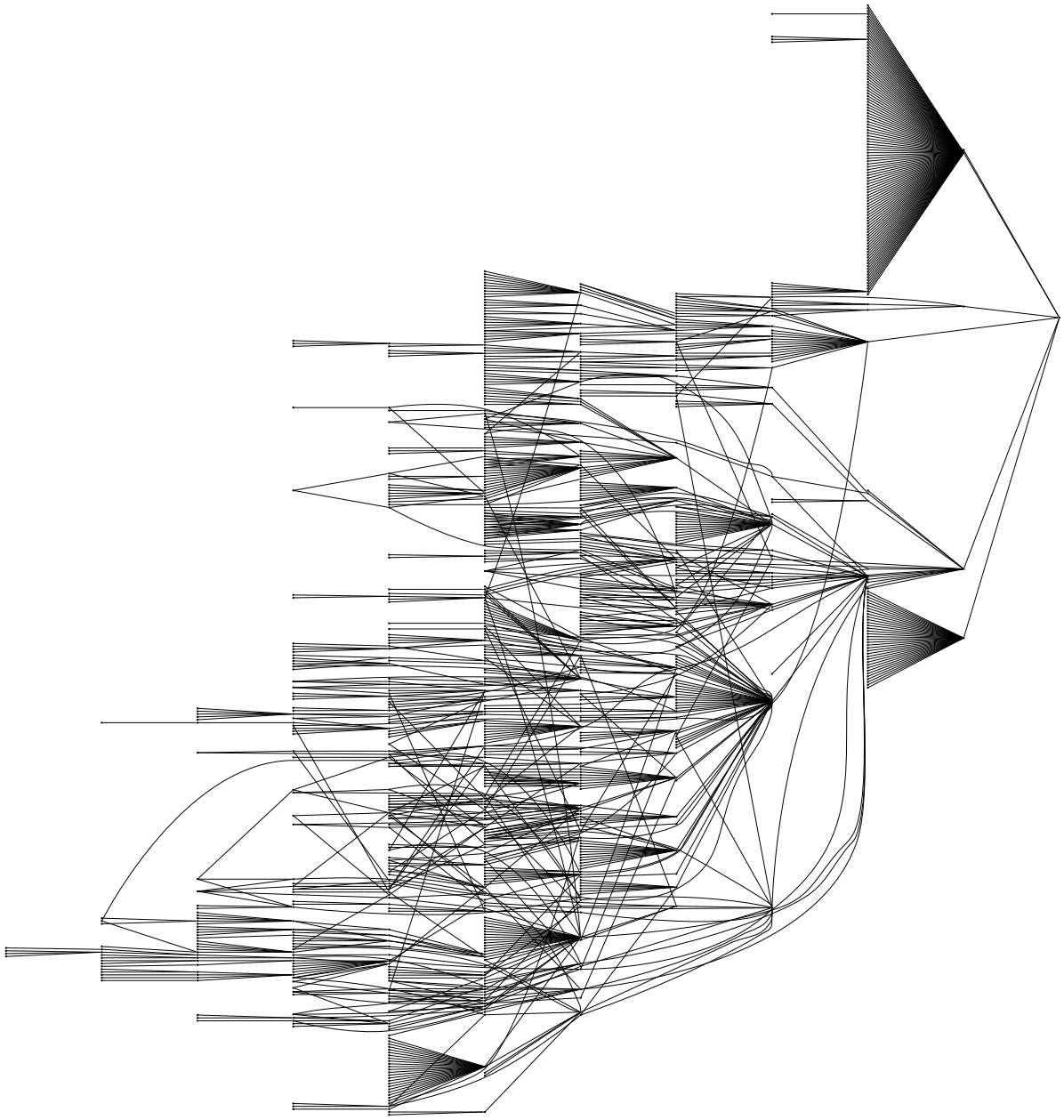

Figure 4: Cellular component, based on 4 Sep GO data package

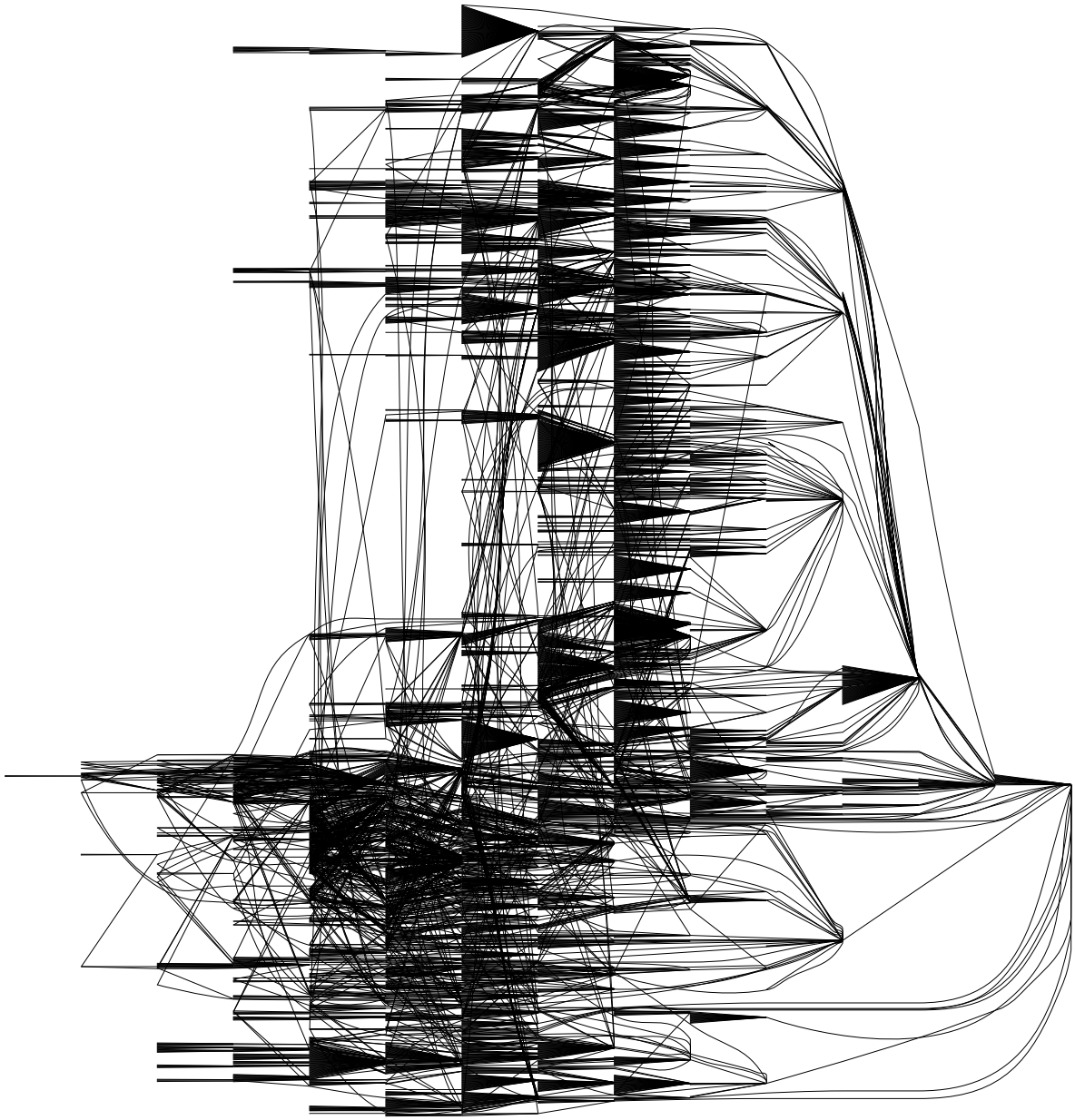

Figure 5: Molecular function, based on 4 Sep GO data package

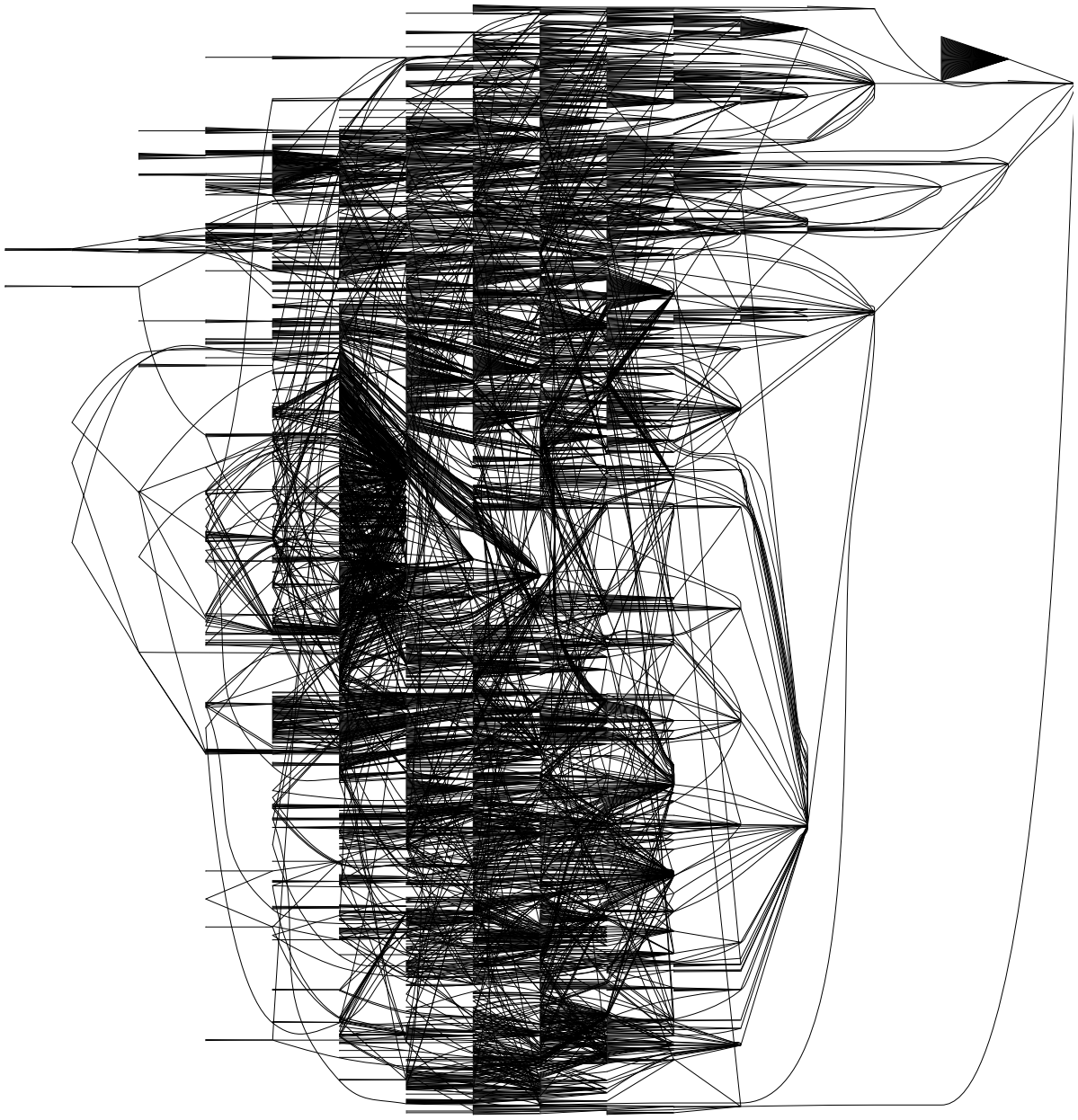

Figure 6: Biological process based on 4 Sep GO data package