Gene set enrichment analysis with **topGO**

Adrian Alexa, Jörg Rahnenführer

November 30, 2006
http://www.mpi-sb.mpg.de/∼alexa

1 Preprocessing

We analyse ALL gene expression data form [Chiaretti, S., *et al.*, 2004]. The dataset consists of 128 microarrays from different patients with ALL. First we load the libraries and the data:

```
> library(topGO)
> library(ALL)
> data(ALL)
```

When the topGO package is loaded three new environments GOBPTerm, GOMFTerm and GOMFTerm are created and binded to the package environment. These environments are build based on GOTERM and they are used for fast recovering of the information specific to each ontology. In order to access all the GO that belong to one of the three ontologies, lets say for Biological Process (BP) ontology, one can do:

```
> BPterms <- ls(GOBPTerm)
> str(BPterms)

 chr [1:11048] "GO:0000001" "GO:0000002" "GO:0000003" "GO:0000004" ...
```

Next we need to load the annotation data. The chip used for the experiment is HGU95aV2 Affymetrix.

```
> affyLib <- "hgu95av2"
> library(package = affyLib, character.only = TRUE)
```

2 Creating a topGOdata object

The first step when using the topGO package is to create a topGOdata object. This object will contain all the information necessary for the *GO analysis*. The gene list, the list of interesting genes, the score of each genes (where there is the case) and the part of the GO ontology (the GO graph) which needs to be used in the analysis.

Thus, we need to define the set of genes that we want to be annotated with GO terms. Usually, one starts with all the genes present on the array, in our case 12625.

```
> geneNames <- featureNames(ALL)
> length(geneNames)
```

In the next step the user needs to define the list of interesting genes or to compute genes score, score which quantifies the significance of each gene. The topGO package deals with these two cases in an unified way. The only difference consists in the way the topGOdata is build.

2.1 Predefined list of interesting genes

If the user has some a priori knowledge about a set of interesting genes, most probably he will like to test the enrichment of GO terms with regard to the list of interesting genes. In this scenario, when only a list of interesting genes is provided, the user is restricted to the use of tests statistics that use only counts of the genes.

To exemplify this lets randomly select lets say 100 genes and consider them as interesting genes.

```
> myInterestedGenes <- sample(geneNames, 100)
> geneList <- factor(as.integer(geneNames %in% myInterestedGenes))
> names(geneList) <- geneNames
> str(geneList)
```

```
 Factor w/ 2 levels "0","1": 1 1 1 1 1 2 1 1 1 1 ...
 - attr(*, "names")= chr [1:12625] "1000_at" "1001_at" "1002_f_at" "1003_s_at" ...
```

The object `geneList` is a named factor which tells which gene is interesting and which not. It should be straightforward to compute such a named vector in a real case in which the user knows which are the genes of interest.

Next we can build the `topGOdata` object. The user needs to specify the ontology of interest (BP, MF or CC) and an annotation function which maps genes/probe IDs to GO terms. The `annFun.hgu` function which comes with the package is such an annotation function. As long as the user is using Affymetrix chips, this function does not need to be modified. For the other cases the user should be able to modify this function without any difficulties.

```
> GOdata <- new("topGOdata", ontology = "MF", allGenes = geneList, annot = annFUN.hgu,
+     affyLib = affyLib)
> GOdata
```

The initialisation of the GOdata object can take around 1 minute depending on the number of annotated genes and on the chosen ontology (in this case we pick the MF as the ontology of interest). By typing `GOdata`, the user can see the values of some slots.

One important point here is that not all the genes that are provided by `geneList` can be annotated to the GO. This can be seen by comparing the number of all available genes (the genes present in `geneList`) the number of feasible genes. Thus, it make sense to use only the feasible genes for the rest of the analysis, since no other information is available for the other genes.

The GO graph shows the number of nodes and edges of the specified GO ontology induced by the `geneList`. This graph contains only GO terms with at least one annotated feasible gene.

2.2 Using the score of the genes

If the set of interesting genes can be computed based on the score assigned to the genes, lets say the p-value returned by a differentially expression study, or the GO analysis needs to use the score of the genes, then the following option of the `topGO` should be used.

A typical example is the study of the ALL dataset where we need to discriminate between ALL cells delivered from either B-cell or T-cell precursors. There are 95 B-cell ALL samples and 33 T-cell ALL samples the dataset.

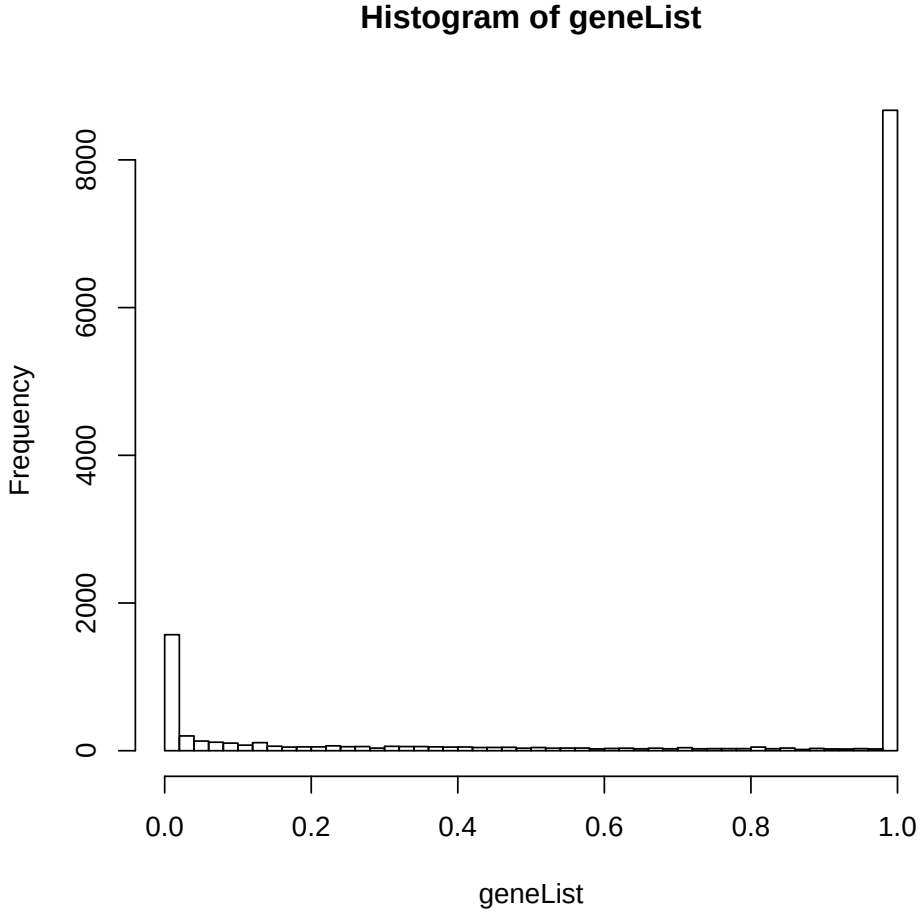

Figure 1: *The distribution of the gene's adjusted p-values.*

```
> discrGenes <- function(eset = ALL) {
+     Tcell.type <- sapply(eset$BT, function(x) return(substr(x, 1, 1) ==
+         "T"))
+     return(as.integer(Tcell.type))
+ }
> y <- discrGenes(ALL)
> str(y)
```

A two-sided t-test can by applied using the function `getPvalues`. By default the function is doing a FDR p-value adjustment to account for multiple testing. A different type of correction can be specified using the `correction` parameter. The distribution of the adjusted p-values is shown in Figure 2.2.

```
> geneList <- getPvalues(exprs(ALL), classlabel = y, alternative = "two.sided")
> hist(geneList, br = 50)
```

Next the user need to define a simple function to select the list of interesting genes. The function needs to do the selection based on the gene scores (in our case the adjusted p-values) and it needs to return a

logical vector which specify which gene is selected and which not. Also the function should have at least one parameter, named `allScore` and should not depend on the names attribute of this parameter. For example, if we consider as interesting genes all genes with an adjusted p-value lower than 0.01 the function will look as following:

```
> topDiffGenes <- function(allScore) {
+     return(allScore < 0.01)
+ }
> x <- topDiffGenes(geneList)
> sum(x)
```

With all these steps done, the user can now build the `topGOdata` object

```
> GOdata <- new("topGOdata", ontology = "BP", allGenes = geneList, geneSel = topDiffGenes,
+     description = "GO analysis of ALL data based on diff. expression.",
+     annot = annFUN.hgu, affyLib = affyLib)

Building most specific GOs .....        ( 1992 GO terms found. )

Build GO DAG topology ..........        ( 3188 GO terms and 5237 relations. )

Annotating nodes ...............        ( 10141 genes annotated to the GO terms. )

> GOdata

------------------------- topGOdata object -------------------------

 Description:
   -  GO analysis of ALL data based on diff. expression.

 Ontology:
   -  BP

 12625 available genes (all genes from the array):
   - symbol:  1000_at 1001_at 1002_f_at 1003_s_at 1004_at  ...
   - score :  0.012969 1 1 1 1  ...
   - 1402  significant genes.

 10141 feasible genes (genes that can be used in the analysis):
   - symbol:  1000_at 1001_at 1002_f_at 1003_s_at 1004_at  ...
   - score :  0.012969 1 1 1 1  ...
   - 1138  significant genes.

 GO graph:
   - a graph with directed edges
   - number of nodes = 3188
   - number of edges = 5237

------------------------- topGOdata object -------------------------
```

Note that the only difference from the case in which we have a list of interesting genes, is the use of the `geneSel` parameter. The further analysis will use the newly defined `GOdata` object.

3 Working with the topGOdata object

Once the topGOdata object is created the user can use the methods associated to this class to access the information encapsulated in the object.

The description slot contains information about the experiment and can be accessed or replaced using the method with the same name.

```
> description(GOdata)
> description(GOdata) <- paste(description(GOdata), "Object modified on:",
+     format(Sys.time(), "%d %b %Y"), sep = " ")
> description(GOdata)
```

Methods to obtain the list of genes that will be used in the further analysis or methods to obtain the genes score are exemplified bellow.

```
> a <- genes(GOdata)
> str(a)
> numGenes(GOdata)
```

Select some random genes and retrieve the score of these genes. In the case in which the object was build using a list of interesting genes, the factor vector that was provided to the constructor of the object will be returned.

```
> selGenes <- sample(a, 10)
> gs <- geneScore(GOdata, whichGenes = selGenes)
> print(gs)
```

If the user wants an unnamed vector or the score of all genes:

```
> gs <- geneScore(GOdata, whichGenes = selGenes, use.names = FALSE)
> print(gs)
> gs <- geneScore(GOdata, use.names = FALSE)
> str(gs)
```

The list of significant genes can be accessed using the method sigGenes().

```
> sg <- sigGenes(GOdata)
> str(sg)
> numSigGenes(GOdata)
```

Another useful method is updateGenes which allows the user to update/change the list of genes (and their score) from a tooGOdata object. Lets say we want to update the list of genes with only the feasible ones:

```
> .geneList <- geneScore(GOdata, use.names = TRUE)
> GOdata
> GOdata <- updateGenes(GOdata, .geneList, topDiffGenes)
> GOdata
```

There are also methods available to access information related to GO and its structure. First, we want to know which GO terms are available for analysis and get all the genes annotated to a subset of these GO terms.

```
> graph(GOdata)

A graphNEL graph with directed edges
Number of Nodes = 3188
Number of Edges = 5237

> ug <- usedGO(GOdata)
> str(ug)

 chr [1:3188] "GO:0000002" "GO:0000003" "GO:0000004" "GO:0000012" ...
```

Lets select some random GO terms and look at the number of annotated genes and get their annotation.

```
> sel.terms <- sample(usedGO(GOdata), 10)
> num.ann.genes <- countGenesInTerm(GOdata, sel.terms)
> num.ann.genes
> ann.genes <- genesInTerm(GOdata, sel.terms)
> str(ann.genes)
```

When the `sel.terms` parameter is missing then all GO terms will be used. The score for the genes or both the score and the name of the genes can be obtained with the method `scoresInTerm()`.

```
> ann.score <- scoresInTerm(GOdata, sel.terms)
> str(ann.score)
> ann.score <- scoresInTerm(GOdata, sel.terms, use.names = TRUE)
> str(ann.score)
```

Finally, some statistics for a set of GO terms are returned by the method `termStat`. As mentioned previously, if the `sel.terms` parameter is missing then the statistics for all available GO terms are returned.

```
> termStat(GOdata, sel.terms)

           Annotated Significant Expected
GO:0006923         2           0     0.22
GO:0009199        61           7     6.85
GO:0046826         1           0     0.11
GO:0007565        60           5     6.73
GO:0045823         2           0     0.22
GO:0046165        20           4     2.24
GO:0031325       199          23    22.33
GO:0000086        11           2     1.23
GO:0006092       102          11    11.45
GO:0016045         4           2     0.45
```

4 The GO analysis

We are now ready to start the GO analysis. The main function is `getSigGroups()` which takes two parameters. The first parameter is of class `topGOdata` and the second parameter is of class `groupStats`. The `topGO` package was design to work with different test statistics and multiple GO graph algorithms, see [Alexa, A., *et al.*, 2006].

There are three algorithms implemented in the package: classic, elim and weight. Also there are two types of test statistic which can be used: test statistics which are based on gene counts, like Fisher's exact test and test statistics based on the genes scores, like Kolmogorov-Smirnov. To distinguish between all the algorithms and to secure that only the right test statistics are used with the right algorithms two classes are defined for each algorithm.

To understand better how this work lets look at an example. Lets say we decided to do the analysis with the classic algorithm. The two classes defined for this algorithm are named `classicCount` and `classicScore`. If an object of this class is given as parameter to the `getSigGroups()` than the classic algorithm will be used. The `getSigGroups()` function can take a while, depending on the size of the graph (the ontology used), so be patient.

```
> test.stat <- new("classicCount", testStatistic = GOFisherTest, name = "Fisher test")
> resultFis <- getSigGroups(GOdata, test.stat)

                The algorithm is scoring 1521 nontrivial nodes
```

Thus, first one defines a test statistic for the chosen algorithm, in this case classic and the runs the algorithm (second line). The slot `testStatistic` contains the test statistic function. In the above example `GOFisherTest` function which implements Fisher's exact test and is available in the package was used. The user can define his own test statistic function and then apply it with the classic algorithm. For example a function which computes the Z score can be implemented using as an example the `GOFisherTest` function.

For the Kolmogorov-Smirnov (KS) test we have:

```
> test.stat <- new("classicScore", testStatistic = GOKSTest, name = "KS tests")
> resultKS <- getSigGroups(GOdata, test.stat)

                The algorithm is scoring 3188 nontrivial nodes
```

We see that this time we used the `classicScore` class. This is done since the KS test needs score of genes and in this case the *representation* of a group of genes (GO term) is different.

The things presented above for the classic also hold for elim and weight with the only remark that for the weight algorithm there is no test based on genes score implemented. Thus, to run the elim algorithm with Fisher's exact test we need to write:

```
> test.stat <- new("elimCount", testStatistic = GOFisherTest, name = "Fisher test",
+     cutOff = 0.01)
> resultElim <- getSigGroups(GOdata, test.stat)

                The algorithm is scoring 1521 nontrivial nodes

                Parameters:         cutOff = 0.01
```

```
        Level 14:        3 nodes to be scored        (0 eliminated genes)

        Level 13:        14 nodes to be scored        (0 eliminated genes)

        Level 12:        34 nodes to be scored        (0 eliminated genes)

        Level 11:        67 nodes to be scored        (8 eliminated genes)

        Level 10:        133 nodes to be scored        (8 eliminated genes)

        Level 9:        222 nodes to be scored        (31 eliminated genes)

        Level 8:        271 nodes to be scored        (113 eliminated genes)

        Level 7:        263 nodes to be scored        (1408 eliminated genes)

        Level 6:        193 nodes to be scored        (1470 eliminated genes)

        Level 5:        158 nodes to be scored        (1767 eliminated genes)

        Level 4:        106 nodes to be scored        (3122 eliminated genes)

        Level 3:        46 nodes to be scored        (3122 eliminated genes)

        Level 2:        10 nodes to be scored        (3122 eliminated genes)

        Level 1:        1 nodes to be scored        (3122 eliminated genes)
```

And for the weight algorithm:

```
> test.stat <- new("weightCount", testStatistic = GOFisherTest, name = "Fisher test",
+     sigRatio = "ratio")
> resultWeight <- getSigGroups(GOdata, test.stat)

                The algorithm is scoring 1521 nontrivial nodes

        Level 14:        3 nodes to be scored.

        Level 13:        14 nodes to be scored.

        Level 12:        34 nodes to be scored.

        Level 11:        67 nodes to be scored.

        Level 10:        133 nodes to be scored.

        Level 9:        222 nodes to be scored.

        Level 8:        271 nodes to be scored.
```

```
        Level 7:        263 nodes to be scored.

        Level 6:        193 nodes to be scored.

        Level 5:        158 nodes to be scored.

        Level 4:        106 nodes to be scored.

        Level 3:        46 nodes to be scored.

        Level 2:        10 nodes to be scored.

        Level 1:        1 nodes to be scored.
```

We next look at the results of the analysis. First we need to put all the resulting p-values into a list. Then we can use the `genTable` function to generate a table with the results.

```
> l <- list(classic = score(resultFis), KS = score(resultKS), elim = score(resultElim),
+     weight = score(resultWeight))
> allRes <- genTable(GOdata, l, orderBy = "weight", ranksOf = "classic",
+     top = 20)
```

`allRes` is a data.frame containing the top 20 GO terms found by the weight algorithm together with statistics and p-values. The parameter `orderBy` let the user decide which p-values should be used for ordering the GO terms. The data.frame is shown in Table 1.

	GO.ID	Term	Annotated	Significant	Expected	classicRank	classic	KS	elim	weight
1	GO:0030333	antigen processing	48.00	28.00	5.39	1.00	3.3e-15	2.1e-10	1.00000	3.3e-15
2	GO:0019882	antigen presentation	52.00	29.00	5.84	2.00	5.5e-15	3.2e-11	0.55761	5.5e-15
3	GO:0009596	detection of pest, pathogen or parasite	12.00	10.00	1.35	8.00	1.6e-08	3.5e-07	1.6e-08	8.9e-07
4	GO:0006636	fatty acid desaturation	11.00	8.00	1.23	10.00	3.0e-06	2.9e-05	3.0e-06	3.0e-06
5	GO:0007044	cell-substrate junction assembly	4.00	4.00	0.45	15.00	0.00016	0.00176	0.00016	0.00016
6	GO:0043297	apical junction assembly	7.00	5.00	0.79	20.00	0.00031	0.00093	0.00031	0.00031
7	GO:0050857	positive regulation of antigen receptor ...	7.00	5.00	0.79	21.00	0.00031	0.00066	0.03478	0.00031
8	GO:0006952	defense response	932.00	173.00	104.59	4.00	2.4e-12	4.3e-10	0.11420	0.00075
9	GO:0007041	lysosomal transport	9.00	5.00	1.01	31.00	0.00151	0.00349	0.00151	0.00151
10	GO:0007516	hemocyte development (sensu Arthropoda)	6.00	4.00	0.67	33.00	0.00196	0.00726	0.00196	0.00196
11	GO:0042116	macrophage activation	6.00	4.00	0.67	34.00	0.00196	0.01117	0.00196	0.00196
12	GO:0007172	signal complex formation	14.00	6.00	1.57	38.00	0.00267	0.01354	0.00267	0.00267
13	GO:0050870	positive regulation of T cell activation	30.00	9.00	3.37	43.00	0.00438	0.00821	0.00438	0.00438
14	GO:0019733	antibacterial humoral response (sensu Ve...	4.00	3.00	0.45	47.00	0.00517	0.02593	0.00517	0.00517
15	GO:0030301	cholesterol transport	4.00	3.00	0.45	48.00	0.00517	0.02349	0.00517	0.00517
16	GO:0006468	protein amino acid phosphorylation	675.00	97.00	75.75	52.00	0.00545	2.9e-18	0.00545	0.00545
17	GO:0006366	transcription from RNA polymerase II pro...	613.00	89.00	68.79	53.00	0.00577	0.05690	0.00577	0.00577
18	GO:0051209	release of sequestered calcium ion into ...	8.00	4.00	0.90	60.00	0.00761	0.02116	0.00761	0.00761
19	GO:0016126	sterol biosynthesis	27.00	8.00	3.03	64.00	0.00772	0.00327	0.00772	0.00772
20	GO:0006801	superoxide metabolism	17.00	6.00	1.91	66.00	0.00818	0.11664	0.00818	0.00818

Table 1: *Significance of the algorithms.*

We can take a quick look at the p-values computed by each algorithm, Figure 4:

```
> par(mfrow = c(2, 2))
> for (nn in names(l)) {
+     p.val <- l[[nn]]
+     hist(p.val[p.val < 1], br = 50, xlab = "p values", main = paste("Histogram for method:",
+         nn, sep = " "))
+ }
```

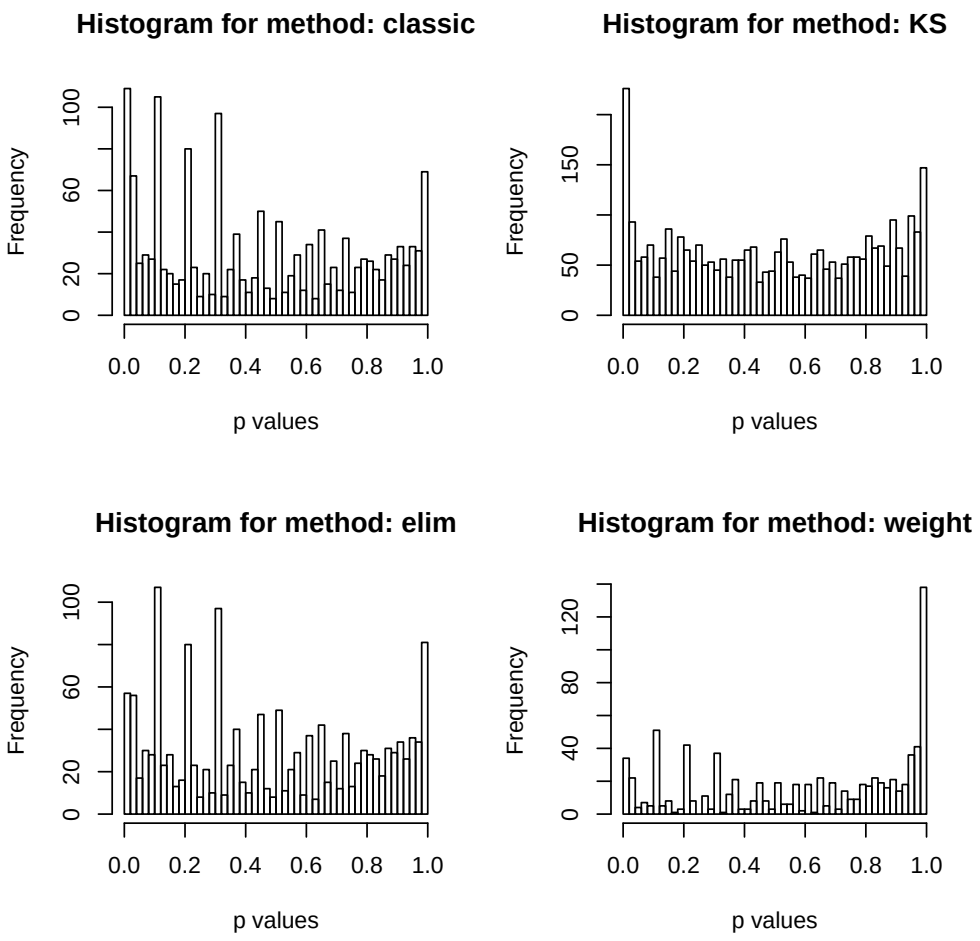

Figure 2: *The distribution of the p-values returned by each method.*

Another insightful way of looking at the results of the analysis is to investigate how the significant GO terms are distributed over the GO graph. For each algorithm the subgraph induced by the most significant GO terms is plotted. In the plots, the *significant nodes* are represented as boxes. The plotted graph is the upper induced graph generated by these *significant nodes*.

```
> showSigOfNodes(GOdata, score(resultFis), firstTerms = 5, useInfo = "all")
> showSigOfNodes(GOdata, score(resultWeight), firstTerms = 5, useInfo = "def")
```

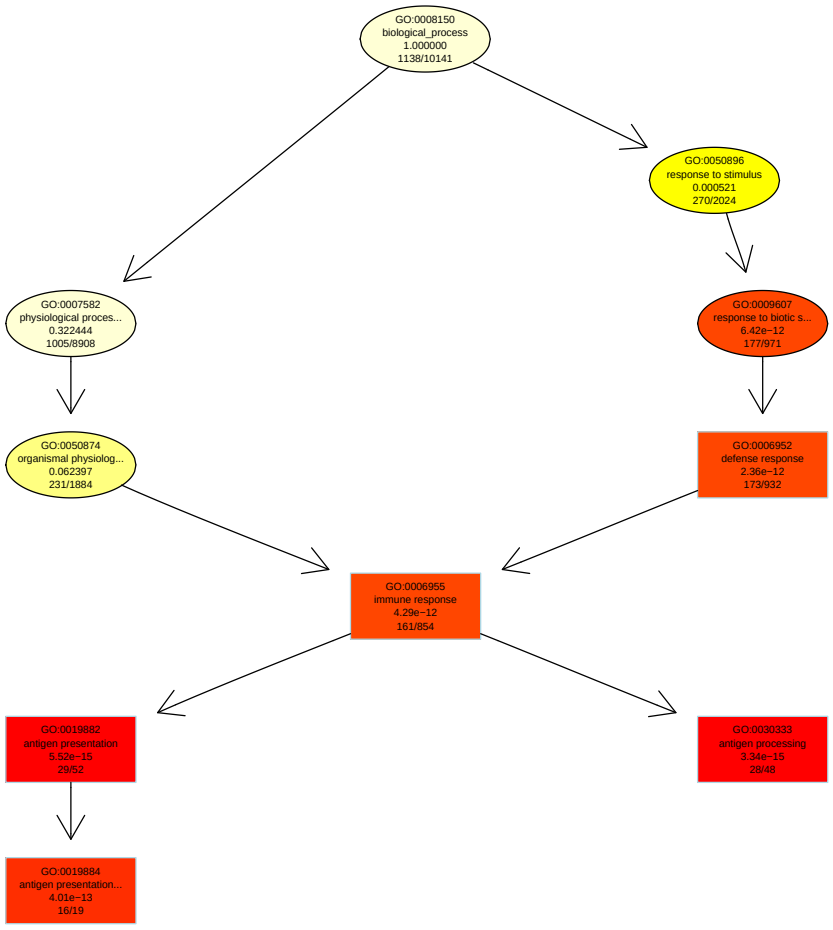

Figure 3: *The subgraph induced by the top 5 GO terms identified by the* classic *algorithm for scoring GO terms for enrichment. Boxes indicate the 10 most significant terms. Box color represents the relative significance, ranging from dark red (most significant) to light yellow (least significant). Black arrows indicate* $is\text{-}a$ *relationships and red arrows* $part\text{-}of$ *relationships.*

If we want to print the graphs to .pdf or .ps file, then we can use the following command:

```
> printGraph(GOdata, resultWeight, firstSigNodes = 5, fn.prefix = "ALL_BT",
+     pdfSW = TRUE)
```

```
ALL_BT_weightCount_5_def  --- no of nodes:  31
```

Or to emphasise the difference between two methods:

```
> printGraph(GOdata, resultWeight, firstSigNodes = 10, resultFis, fn.prefix = "ALL_BT",
+     useInfo = "def")
```

```
ALL_BT_weightCount_classicCount_10_def  --- no of nodes:  69
```

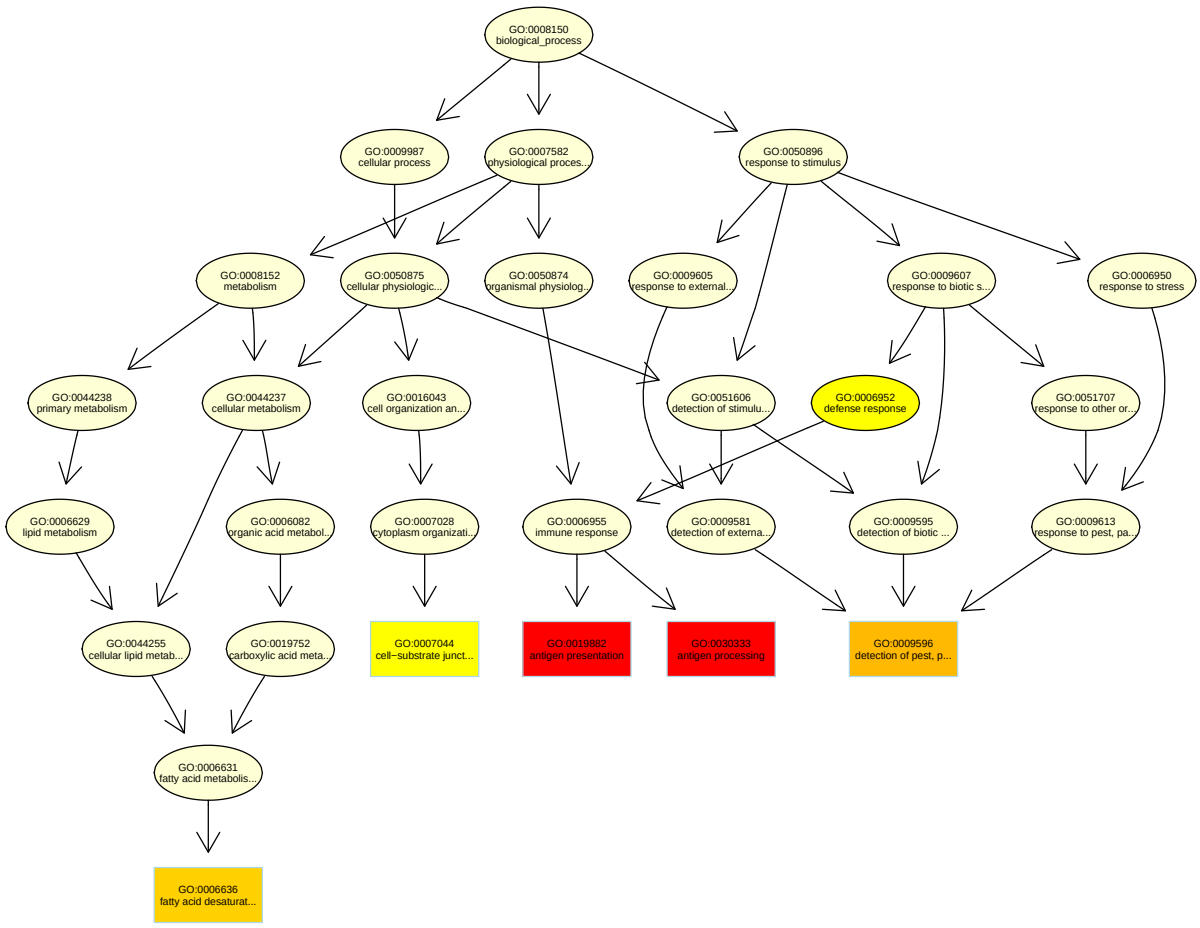

Figure 4: *The subgraph induced by the top 5 GO terms identified by the* weight *algorithm for scoring GO terms for enrichment. Boxes indicate the 10 most significant terms. Box color represents the relative significance, ranging from dark red (most significant) to light yellow (least significant). Black arrows indicate* is-a *relationships and red arrows* $part$-of *relationships.*

```
> printGraph(GOdata, resultElim, firstSigNodes = 15, resultFis, fn.prefix = "ALL_BT",
+     useInfo = "all")

ALL_BT_elimCount_classicCount_15_all  --- no of nodes:  68
```

References

[Alexa, A., *et al.*, 2006] Alexa, A., *et al.* (2006). Improvined scoring of functional groups from gene expression data be decorrelating go graph structure. *Bioinformatics*, 22(13):1600–1607.

[Chiaretti, S., *et al.*, 2004] Chiaretti, S., *et al.* (2004). Gene expression profile of adult T-cell acute lymphocytic leukemia identifies distinct subsets of patients with different response to therapy and survival. *Blood*, 103(7):2771–2778.