Gene set enrichment analysis with **topGO**

Adrian Alexa, Jörg Rahnenführer

October 28, 2009
`http://www.mpi-sb.mpg.de/∼alexa`

Contents

1 Introduction

topGO package provides a set of classes and methods which allows the user to run the mentioned algorithms/methods. Moreover the user can use the current framework to develop and test new enrichment algorithms which make use of the GO structure.

This document gives an overview .....

The topGO package is designed to work with different test statistics and with multiple GO graph algorithms, see [Alexa, A., *et al.*, 2006].

The default algorithm used by the topGO package is a mixture between the elim and the weight algorithms and it will be referred as topgo (lower-case letters).

	Fisher	KS	t	globaltest	Category
classic	yes	yes	yes	yes	yes
elim	yes	yes	yes	yes	yes
weight	yes	no	no	no	no
topgo	yes	yes	yes	yes	no
parentChild	yes	no	no	no	no

Table 1: *The algorithms supported by* **topGO**.

In this manuscript we will use the terms gene and probe to refer to the same entity. .....

The next section provides a quick tour into topGO and is thought to be independent of the rest of this manuscript. The following sections will provide details on the functions used in the sample section as well as showing more advance functionality implemented in the topGO package.

2 Sample session

This section describes a simple working-session using topGO. There are only a handful of commands necessary to perform a gene set enrichment analysis and we will be briefly presented them bellow.

A typical session can be divided into three steps:

1. Collection and preprocessing of the data. The list of genes and the correspondent gene' score, the criteria for selecting genes based on their scores and the gene-to-GO annotations are all collected to form a suitable R object.
2. Using the object created in the first step the user can perform enrichment analysis using various statistical tests and methods that deal with the GO topology.
3. Analysis of the results.

But before going through each of these steps we need to decide which biological question we want to answer. This will most likely dictate which test statistic we need to use. The choice of the test is also restricted by the data available at hand. The knowledge of the test will dictate the way the gene expression data needs to be process. Here we will test for enrichment of GO terms with differentially expressed genes using the Kolmogorov-Smirnov test and Fisher's exact test as examples.

2.1 Step 1: Preparing the data

In this step a convenient R object of class topGOdata is created containing all the information required for the remaining two steps. The user needs to provide a list of genes, GO annotations and a criteria for selecting interesting genes.

In most cases we want to test enrichment of GO terms with differentially expressed genes. Thus, the starting point is a list of genes and the respective p-values for differential expression. A toy example of a list of gene identifiers and the respective p-values is provided by the geneList object.

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

The `geneList` data is based on a differential expression analysis of the ALL dataset. It contains just a small number, 323, of genes with the corespondent p-values. For these genes we need GO annotations to be able to form the gene groups. The information on where to find the GO annotations is stored in the ALL object and its easily accessible.

```
> affyLib <- paste(annotation(ALL), "db", sep = ".")
> library(package = affyLib, character.only = TRUE)
```

The chip used in the experiment is the `hgu95av2` from Affymetrix, as we can see from the `affyLib` object. When we loaded the `geneList` object a function which will select the differentially expressed genes from the list was also loaded under the name of `topDiffGenes`. For example using this function we can see that there are 50 genes with a row p-value less than 0.01 out of a total of 323 genes.

```
> sum(topDiffGenes(geneList))
```

```
[1] 50
```

We now have all the data necessary to build on object of type `topGOdata`. This object will contain all the gene identifiers and their score, the GO annotations, the GO hierarchical structure and all the other information needed to perform the enrichment analysis.

```
> sampleGOdata <- new("topGOdata", description = "Simple session", ontology = "BP",
+     allGenes = geneList, geneSel = topDiffGenes, nodeSize = 10, annot = annFUN.db,
+     affyLib = affyLib)
```

A summary of the `sampleGOdata` object can be seen by typing the object name at the R prompt. Having all the data stored into this object facilitates the access to identifiers, annotations and to obtain basic data statics.

```
> sampleGOdata
```

2.2 Step 2: Running the desired tests

Once we have an object of class `topGOdata` we can start with the enrichment analysis. Since for each gene we have a score and there is also a procedure to select interesting genes based on the scores we will use two types of test statistics: Fisher's exact test which is based on gene counts, and a Kolmogorov-Smirnov like test which needs gene scores.

The function `runTest` is used to apply the specified enrichment test and method to the data. It has three basic arguments. The first argument needs to be on object of class `topGOdata`. The second argument is of type character and specifies which method for dealing with the GO graph structure will be used (this argument is optional). And the third argument specifies the test statistic and is of type character.

First we perform a classical enrichment analysis by testing the over-representation of GO terms with differentially expressed genes. In the classical case each GO category is tested independently.

```
> resultFisher <- runTest(sampleGOdata, algorithm = "classic", statistic = "fisher")
```

`runTest` returns an object of class `topGOresult`. A short summary of this object is shown bellow.

```
> resultFisher
```

	GO.ID	Term	Annotated	Significant	Expected	Rank in classicFisher	classicFisher	classicKS	elimKS
1	GO:0007067	mitosis	182	17	28.89	431	0.9999	2.0e-09	2.0e-09
2	GO:0051301	cell division	129	15	20.48	411	0.9711	3.8e-06	3.8e-06
3	GO:0000087	M phase of mitotic cell cycle	187	22	29.68	427	0.9946	5.2e-10	0.0020
4	GO:0000226	microtubule cytoskeleton organization	31	5	4.92	270	0.5675	0.0039	0.0039
5	GO:0022403	cell cycle phase	192	27	30.48	373	0.8949	1.2e-10	0.0044
6	GO:0050793	regulation of developmental process	54	16	8.57	33	0.0036	0.0049	0.0049
7	GO:0030098	lymphocyte differentiation	22	9	3.49	27	0.0031	0.0084	0.0084
8	GO:0008202	steroid metabolic process	11	5	1.75	60	0.0181	0.0091	0.0091
9	GO:0051329	interphase of mitotic cell cycle	19	2	3.02	341	0.8368	0.0092	0.0092
10	GO:0019219	regulation of nucleobase, nucleoside, nu...	52	12	8.25	130	0.0921	0.0097	0.0097

Table 2: *Significance of GO terms according to* classic *and* elim *methods.*

```
Description: Simple session
Ontology: BP
'classic' algorithm with the 'fisher' test
470 GO terms scored: 44 terms with p < 0.01
Annotation data:
    Annotated genes: 315
    Significant genes: 50
    Min. no. of genes annotated to a GO: 10
    Nontrivial nodes: 435
```

Next we will test the enrichment using the Kolmogorov-Smirnov test. We will use the both the classic and the elim methods.

```
> resultKS <- runTest(sampleGOdata, algorithm = "classic", statistic = "ks")
> resultKS.elim <- runTest(sampleGOdata, algorithm = "elim", statistic = "ks")
```

Please note that some statistical tests will not work with any method. The compatibility matrix between the methods and statistical tests is shown in Table 1.

The p-values computed by the `runTest` function are unadjusted for multiple testing. We do note advocate against adjusting the p-values of the tested groups, but in many cases the an adjusted p-value can be misleading.

2.3 Step 3: Analysis of results

After the enrichment tests are performed the researcher needs tools for analysing and interpreting the results. `GenTable` is an easy to use function for analysing the most significant GO terms and the corresponding p-values. For example, we want to see which are the top 10 significant GO terms identified by the elim method. We also want to compare the ranks and the p-values of these GO terms in the case where the classic method was employe used.

```
> allRes <- GenTable(sampleGOdata, classicFisher = resultFisher, classicKS = resultKS,
+     elimKS = resultKS.elim, orderBy = "elimKS", ranksOf = "classicFisher",
+     topNodes = 10)
```

The `GenTable` function returns a data.frame containing the top `topNodes` GO terms identified by the elim algorithm, see `orderBy` argument. The data.frame includes some statistics on the GO terms and the p-values obtained from the methods passes as arguments. Table 2 shows the results.

For accessing the GO term's p-values from a `topGOresult` object the user should use the `score` functions. As a simple example, we look at the differences between the results of the classic and the elim methods in the case of the Kolmogorov-Smirnov test. Due to the fact that there are few significant GO terms, 22 terms with a p-value less than 0.01 in the classic method, one would expect that only few GO terms will have different p-values between these two methods. This, of course, depends on the value of the cutoff parameter used by the elim method.

We can see in Figure 1 that there are indeed few differences between the two methods. Some GO terms witch are found significant by the classic method are less significant in the elim, as expected.

```
> pValue.classic <- score(resultKS)
> pValue.elim <- score(resultKS.elim)[names(pValue.classic)]
> plot(pValue.classic, pValue.elim, xlab = "p-value classic", ylab = "p-value elim",
+      cex = 0.5)
```

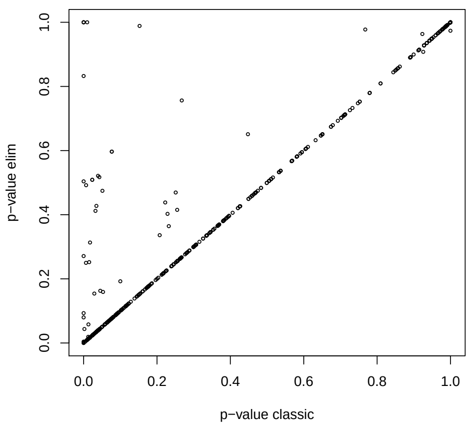


Figure 1: *p-values scatter plot for the* classic *and* elim *methods.*

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 the `elim` algorithm the subgraph induced by the 5 most significant GO terms is plotted. In the plot, the significant nodes are represented as boxes. The plotted graph is the upper induced graph generated by these significant nodes.

```
> showSigOfNodes(sampleGOdata, score(resultKS.elim), firstSigNodes = 5, useInfo = "all")
```

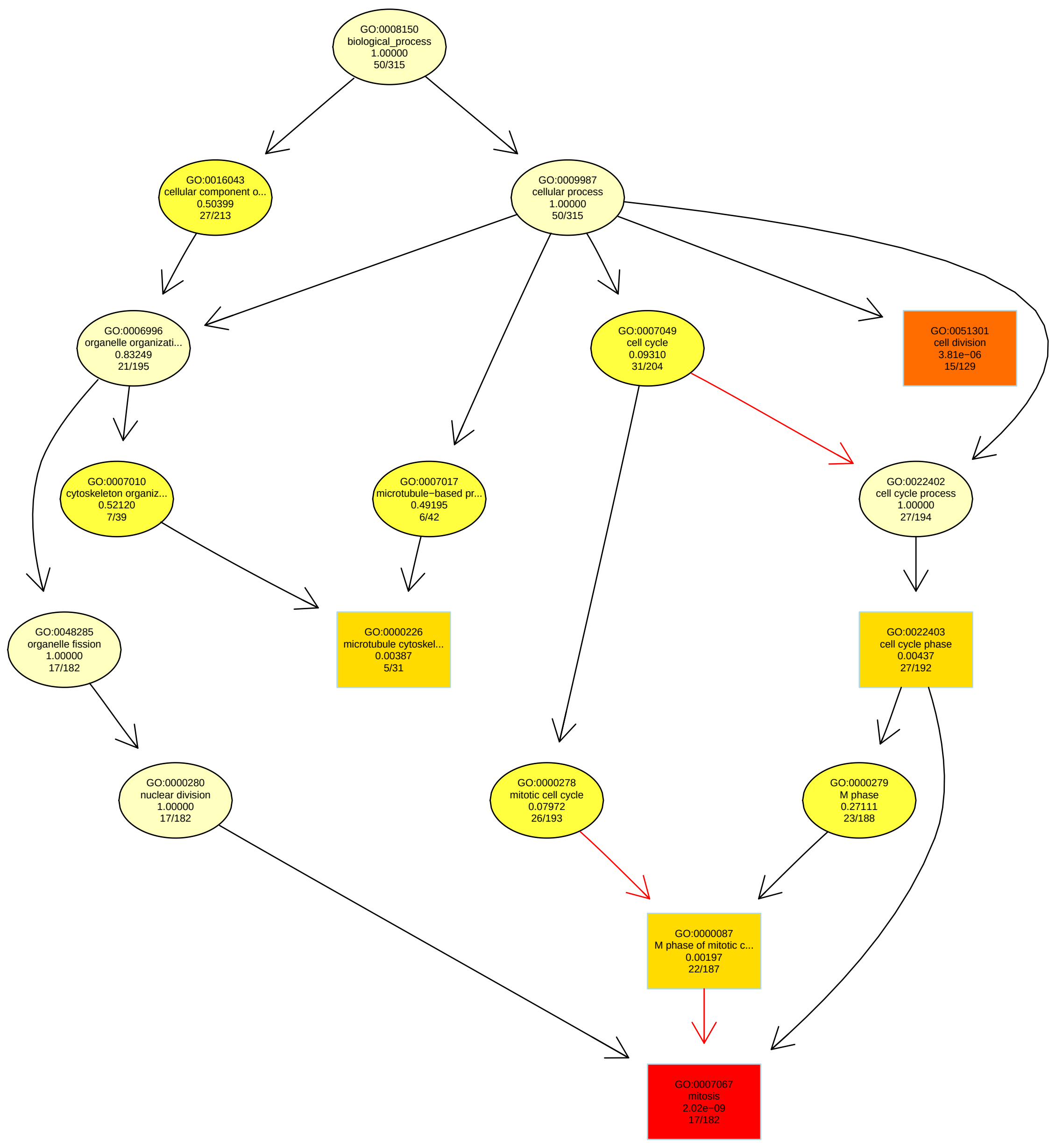


Figure 2: *The subgraph induced by the top* 5 *GO terms identified by the* elim *algorithm for scoring GO terms for enrichment. Rectangles indicate the* 5 *most significant terms. Rectangle color represents the relative significance, ranging from dark red (most significant) to light yellow (least significant). Black arrows indicate* is-a *relationships and red arrows* $part$-of *relationships. For each node, some basic information is displayed. The first two lines show the GO identifier and a trimmed GO name. In the third line the raw p-value is shown. The forth line is showing the number of significant genes and the total number of genes annotated to the respective GO term.*

3 Setup

To demonstrate the package functionality we will use the ALL gene expression data from [Chiaretti, S., *et al.*, 2004]. The dataset consists of 128 microarrays from different patients with ALL. Additionally, custom annotations and artificial datasets will be used to demonstrate specific futures.

We first load the libraries and the data:

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

When the `topGO` package is loaded three environments `GOBPTerm, GOMFTerm` and `GOMFTerm` are created and binded to the package environment. These environments are build based on the `GOTERM` environment from package `GO.db`. They are used for fast recovering of the information specific to each ontology. In order to access all GO groups that belong to a specific ontology, e.g. Biological Process (BP), one can type:

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

```
[1] "GO:0000001" "GO:0000002" "GO:0000003" "GO:0000011" "GO:0000012" "GO:0000017"
```

Usually one needs to remove probes with low expression value and genes which might have very small variability across the samples. Package `genefilter` provides tools for filtering genes. In this analysis we choose to filter as many genes as possible for computational reasons. *Having a smaller gene universe allows as to exemplify here the functionality of the* `topGO` *package in a relatively short time.* The effect of the gene filtering will be discuss in more details in Section 4.4.

```
> library(genefilter)
> selProbes <- genefilter(ALL, filterfun(pOverA(0.2, log2(100)), function(x) (IQR(x) >
+     0.25)))
> eset <- ALL[selProbes, ]
```

The filter selects only 4101 probesets out of 12625 probesets available on the *hgu*95*av*2 array.

4 Creating a `topGOdata` object

The central step in using the `topGO` package is to create a `topGOdata` object. *This object will be the input of the testing procedures ...* . This object will contain all information necessary for the GO analysis, namely the list of genes, the list of interesting genes, the gene' score, if available and the part of the GO ontology (the GO graph) which needs to be used in the analysis.

To build such an object the user needs the followings:

- A list of gene identifiers and optionally their score. The gene-wise score can be the t-test statistic (or the p-value) for differential expression, correlation with a phenotype, or any other relevant score.
- A mapping between gene identifiers and GO terms. In most cases this mapping is directly available in Bioconductor as a chip specific annotation package. In this case the user just needs to specify the name of the annotation to be used. For example, the annotation package needed for the ALL dataset is `hgu95av2.db`.

  However it can happen that there is no annotation package available in Bioconductor (custom chips or new mappings between genes and GO terms). Section 4.1 will describe how custom annotations can be used.
- The GO hierarchical structure. The structure is obtained from the `GO.db` package. At the moment user cannot use a different version of the ontology than the one provided by `GO.db`.

The gene universe and the interesting genes
The set of all genes from the array/study will be referred from now on as the gene universe. From the gene universe, the user needs to define the list of interesting genes or to compute gene-wise scores that quantify the significance of each gene. The `topGO` package deals with these two cases in a unified way once the `topGOdata` object is constructed. The only difference stands in the way the object is build.

Usually one starts with all feasible genes present on the array as the gene universe. In the case of the ALL dataset we start with 4101 genes, genes that were not removed by the filtering procedure.

4.1 Custom annotations

This section describes how custom GO annotations can be used in building a `topGOdata` object.

Annotations need to be provided either as *gene-to-GOs* or as *GO-to-genes* mappings. An example of such mapping can be found in the ”topGO/examples” directory. The file contains gene-to-GOs mappings, meaning that for each gene identifier the GO terms to which this genes is specifically annotated are listed. To read this file we use the `readMappings` function.

```
> ensembl2GO <- readMappings(file = system.file("examples/ensembl2go.map",
+     package = "topGO"))
> str(head(ensembl2GO))
```

```
List of 6
 $ 068724: chr [1:5] "GO:0005488" "GO:0003774" "GO:0001539" "GO:0006935" ...
 $ 119608: chr [1:6] "GO:0005634" "GO:0030528" "GO:0006355" "GO:0045449" ...
 $ 049239: chr [1:13] "GO:0016787" "GO:0017057" "GO:0005975" "GO:0005783" ...
 $ 067829: chr [1:16] "GO:0045926" "GO:0016616" "GO:0000287" "GO:0030145" ...
 $ 106331: chr [1:10] "GO:0043565" "GO:0000122" "GO:0003700" "GO:0005634" ...
 $ 214717: chr [1:7] "GO:0004803" "GO:0005634" "GO:0008270" "GO:0003677" ...
```

The object returned by the `readMappings` function is a named list of character vectors. The list names are genes identifiers and for each gene identifier the character vector contains the GO identifiers it maps to. It is sufficient for the mapping to contain only the most specific GO annotations. However, there is no problem if all or some ancestors of the most specific GO terms are included. This redundancy is not making for a faster running time and if possible it should be avoided.

The user can read the annotations from text files or they can build an object such as `ensembl2GO` directly into R. The text file format required by the `readMappings` function is very simple. It consists of one line for each gene with the following line structure:

```
gene_ID<TAB>GO_ID1, GO_ID2, GO_ID3, ....
```

Reading GO-to-genes mappings from a file is done in the same way using the `readMappings` function. However, it is the user responsibility to know the direction of the mappings. The user can easily transform a mapping from gene-to-GOs to GO-to-genes (or vice-versa) using the `inverseList` function:

```
> GO2ensembl <- inverseList(ensembl2GO)
> str(head(GO2ensembl))
```

```
List of 6
 $ GO:0000122: chr "106331"
 $ GO:0000139: chr [1:6] "133103" "111846" "109956" "161395" ...
 $ GO:0000166: chr [1:10] "067829" "157764" "100302" "074582" ...
 $ GO:0000186: chr "181104"
 $ GO:0000209: chr "159461"
 $ GO:0000228: chr "214717"
```

4.2 Predefined list of interesting genes

If the user has some a priori knowledge about a set of interesting genes, he can test the enrichment of GO terms with regard to this list of interesting genes. In this scenario, when only a list of interesting genes is provided, the user can use only tests statistics that are based on gene counts, like Fisher's exact test, Z score and alike.

To demonstrate how custom annotation can be used this section is based on the toy dataset, the `ensembl2GO` data, from Section 4.1. The gene universe in this case is given by the list names:

```
> geneNames <- names(ensembl2GO)
> head(geneNames)
```

Since for the available genes we do not have any measurement and thus no criteria to select interesting genes, we randomly select 10% genes from the gene universe and consider them as interesting genes.

```
> myInterestedGenes <- sample(geneNames, length(geneNames)/10)
> geneList <- factor(as.integer(geneNames %in% myInterestedGenes))
> names(geneList) <- geneNames
> str(geneList)
```

```
 Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 1 ...
 - attr(*, "names")= chr [1:100] "068724" "119608" "049239" "067829" ...
```

The `geneList` object is a named factor that indicates which genes are interesting and which not. It should be straightforward to compute such a named vector in a real case situation, where the user has his own predefined list of interesting genes.

We now have all the elements to construct a `topGOdata` object. The arguments of the `initialize` function used to construct an instance of this object are described bellow.

`ontology:` character string specifying the ontology of interest (BP, MF or CC)

`description:` character string containing a short description of the study [optional].

`allGenes:` named object of type numeric of factor. The names attribute contains the genes identifiers. The genes listed in this object define the gene universe.

`geneSelectionFun:` function to specify which genes are interesting based on the gene scores. It should be present iff the `allGenes` object is of type numeric.

`nodeSize:` an integer larger or equal to 1. This parameter is used to prune the GO hierarchy from the terms which have less than `nodeSize` annotated genes (after the true path rule is applied).

`annotationFun:` function which maps genes identifiers to GO terms. There are a couple of annotation function included in the package trying to address the user's needs. The annotation functions take three arguments. One of those arguments is specifying where the mappings can be found, and needs to be provided by the user. Here we give a short description of each:

- `annFUN.db` this function is intended to be used as long as the chip used by the user has an annotation package available in Bioconductor.
- `annFUN.org` this function is using the mappings from the "org.XX.XX" annotation packages. Currently, the function supports the following gene identifiers: Entrez, GenBank, Alias, Ensembl, Gene Symbol, GeneName and UniGene.
- `annFUN.gene2GO` this function is used when the annotations are provided as a gene-to-GOs mapping.
- `annFUN.GO2gene` this function is used when the annotations are provided as a GO-to-genes mapping.
- `annFUN.file` this function will read the annotationsof the type gene2GO or GO2genes from a text file.

`...:` list of arguments to be passed to the `annotationFun`

To build the `topGOdata` object, we will use the MF ontology. The mapping is given by the `ensembl2GO` list which will be used with the `annFUN.gene2GO` function.

```
> GOdata <- new("topGOdata", ontology = "MF", allGenes = geneList, annot = annFUN.gene2GO,
+     gene2GO = ensembl2GO)
```

The building of the `GOdata` *object can take some time, depending on the number of annotated genes and on the chosen ontology.*

In our example the running time is quite fast given that we have few genes and the size of the ontology is relatively small (compared to the BP ontology)

xxxxxxxxxxxxxxxxxx

The advantage of having all the necessary data stored into one object is ......

The advantage of having (information on) the gene scores (or better genes measurements) as well as a way to define which are the interesting genes, in the `topGOdata` *object is that one can apply different group test. ....*

We are able to test multiple hypothesis or play with different parameters .....

By typing `GOdata` at the R prompt, the user can see a summary of the data.

```
> GOdata
```

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

 Description:
   -  

 Ontology:
   -  MF 

 100 available genes (all genes from the array):
   - symbol:  068724 119608 049239 067829 106331  ...
   - 10  significant genes. 

 89 feasible genes (genes that can be used in the analysis):
   - symbol:  068724 119608 049239 067829 106331  ...
   - 9  significant genes. 

 GO graph (nodes with at least  1  genes):
   - a graph with directed edges
   - number of nodes = 226 
   - number of edges = 269 

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

One important point to notice is that not all the genes that are provided by `geneList`, the initial gene universe, can be annotated to the GO. This can be seen by comparing the number of all available genes, the genes present in `geneList`, with the number of feasible genes. We are therefore forced at this point to restrict the gene universe to the set of feasible genes for the rest of the analysis.

The summary on the GO graph shows the number of GO terms and the relations between them of the specified GO ontology. This graph contains only GO terms which have at least one gene annotated to them.

4.3 Using the genes score

In many cases the set of interesting genes can be computed based on a score assigned to all genes, for example based on the p-value returned by a study of differential expression. In this case, the `topGOdata` object can store the genes score and a rule specifying the list of interesting genes. The advantage of having both the scores and the procedure to select interesting genes encapsulated in the `topGOdata` object is that the user can choose different types of tests statistics for the GO analysis without modifying the input data.

A typical example for the ALL dataset is the study where we need to discriminate between ALL cells delivered from either B-cell or T-cell precursors.

```
> y <- as.integer(sapply(eset$BT, function(x) return(substr(x, 1, 1) == "T")))
> table(y)
```

There are 95 B-cell ALL samples and 95 T-cell ALL samples in the dataset. A two-sided *t*-test can by applied using the function `getPvalues` (a wraping function for the `mt.teststat` from the `multtest` package). By default the function computes FDR (false discovery rate) adjusted *p*-value in order to account for multiple testing. A different type of correction can be specified using the `correction` argument.

```
> geneList <- getPvalues(exprs(eset), classlabel = y, alternative = "greater")
```

`geneList` is a named numeric vector. The gene identifiers are stored in the names attribute of the vector. This set of genes defines the gene universe.

Next, a function for specifying the list of interesting genes must be defined. This function needs to select genes based on their scores (in our case the adjusted *p*-values) and must return a logical vector specifying which gene is selected and which not. The function must have one argument, named `allScore` and must not depend on any attributes of this object. In this example we will consider as interesting genes all genes with an adjusted *p*-value lower than 0.01. This criteria is implemented in the following function:

```
> 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. For a short description of the arguments used by the `initialize` function see Section4.2

```
> GOdata <- new("topGOdata", description = "GO analysis of ALL data based on diff. expression.",
+     ontology = "BP", allGenes = geneList, geneSel = topDiffGenes, annot = annFUN.db,
+     nodeSize = 5, affyLib = affyLib)
```

It is often the case that many GO terms which have few annotated genes are detected to be significantly enriched due to artifacts in the statistical test. These small sized GO terms are of less importance for the analysis and in many cases they can be omitted. By using the `nodeSize` argument the user can control the size of the GO terms used in the analysis. Once the genes are annotated to the each GO term and the true path rule is applied the nodes with less than `nodeSize` annotated genes are removed from the GO hierarchy. We found that values between 5 and 10 for the `nodeSize` parameter yield more stable results. The default value for the `nodeSize` parameter is 1, meaning that no pruning is performed.

Note that the only difference in the initialisation of an object of class `topGOdata` to the case in which we start with a predefined list of interesting genes is the use of the `geneSel` argument. All further analysis depends only on the `GOdata` object.

4.4 Filtering and missing GO annotations

Before going further with the enrichment analysis we analyse which of the probes available on the array can be used in the analysis.

We want to see if the filtering performed in Section 3 removes important probes. There are a total of 12625 probes on the *hgu95av2* chip. One assumes that only the noisy probes, probes with low expression values or small variance across samples are filtered out from the analysis.

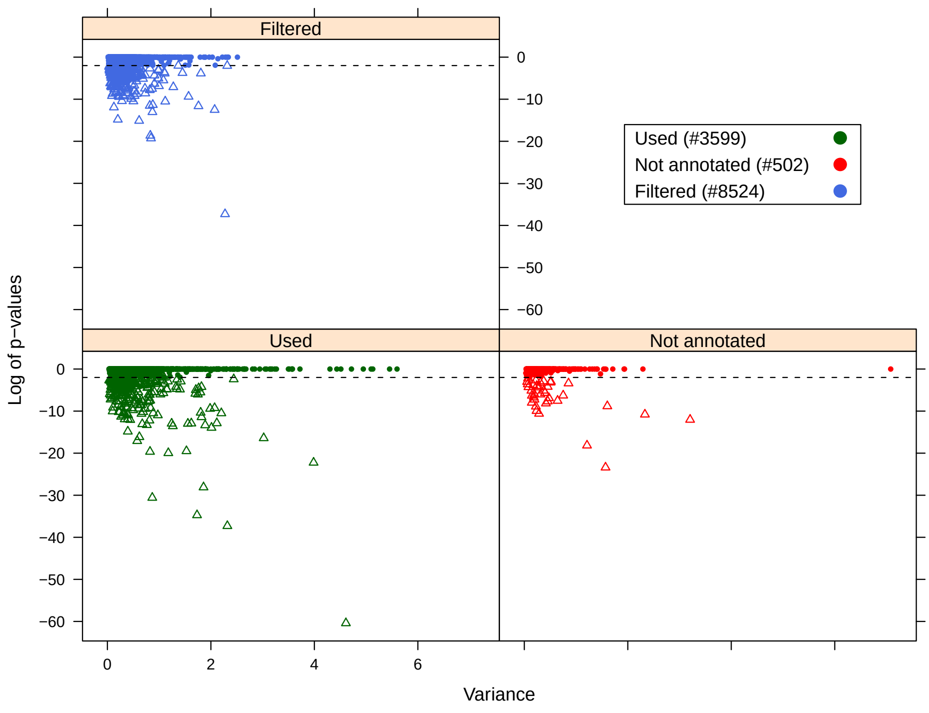


Figure 3: *Scatter plot of FDR adjusted p-values against variance of probes. Points below the horizontal line are significant probes.*

The number of probes have a direct effect on the multiple testing adjustment of p-values. Too many probes will result in too conservative adjusted p-values which can bias the result of tests like Fisher's exact test. Thus it is important to carefully analyse this step.

```
> allProb <- featureNames(ALL)
> groupProb <- integer(length(allProb)) + 1
> groupProb[allProb %in% genes(GOdata)] <- 0
> groupProb[!selProbes] <- 2
> groupProb <- factor(groupProb, labels = c("Used", "Not annotated", "Filtered"))
> tt <- table(groupProb)
> tt
```

```
groupProb
         Used Not annotated      Filtered
         3599           502          8524
```

Out of the filtered probes only 88% have annotation to GO terms. The filtering procedure removes 8524 probes which is a very large percentage of probes (more than 50%), but we did this intentionally to reduce the expression set for computational purposes.

We perform a differential expression analysis on all available probes and we check if differentially expressed genes are leaved out from the enrichment analysis.

```
> pValue <- getPvalues(exprs(ALL), classlabel = y, alternative = "greater")
> geneVar <- apply(exprs(ALL), 1, var)
> dd <- data.frame(x = geneVar[allProb], y = log10(pValue[allProb]), groups = groupProb)
> xyplot(y ~ x | groups, data = dd, groups = groups)
```

Figure 3 shows for the three groups of probes the adjusted p-values and the gene-wise variance. Probes with large changes between conditions have large variance and low p-value. In an ideal case, one would expect to have a large density of probes in the lower right corner of **Used** panel and few probes in this region in the other two panels. We can see that the filtering process throws out some significant probes and in a real analysis a more conservative filtering needs to be applied. However, there are also many differentially expressed probes without GO annotation which cannot be used in the analysis.

5 Working with the `topGOdata` object

Once the `topGOdata` object is created the user can use various methods defined for this class to access the information encapsulated in the object.

The `description` slot contains information about the experiment. This information 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 for obtaining all gene scores are exemplified below.

```
> a <- genes(GOdata)
> head(a)
```

```
[1] "1000_at"   "1005_at"   "1007_s_at" "1008_f_at" "1009_at"   "100_g_at"
```

```
> numGenes(GOdata)
```

```
[1] 3599
```

Next we describe how to retrieve the score of a specified set of genes, e.g. a set of randomly selected genes. If the object was constructed using a list of interesting genes, then the factor vector that was provided at the building 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 scores) from a `topGOdata` object. If one wants to update the list of genes by including only the feasible ones, one can type:

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

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

```
> graph(GOdata)
```

```
A graphNEL graph with directed edges
Number of Nodes = 2191
Number of Edges = 4355
```

```
> ug <- usedGO(GOdata)
> head(ug)
```

```
[1] "GO:0000003" "GO:0000018" "GO:0000041" "GO:0000059" "GO:0000060" "GO:0000070"
```

Next, we select some random GO terms, count the number of annotated genes and obtain their annotation.

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

When the `sel.terms` argument is missing all GO terms are used. The scores for all genes, possibly annotated with names of the genes, can be obtained using the method `scoresInTerm()`.

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

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

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

```
           Annotated Significant Expected
GO:0060191        19           3     1.68
GO:0048741        17           2     1.50
GO:0019059        14           1     1.24
GO:0043087        31           5     2.74
GO:0070227        10           1     0.88
GO:0055085        73           8     6.45
GO:0046888         6           0     0.53
GO:0045913         8           0     0.71
GO:0009261         5           0     0.44
GO:0048641         7           2     0.62
```

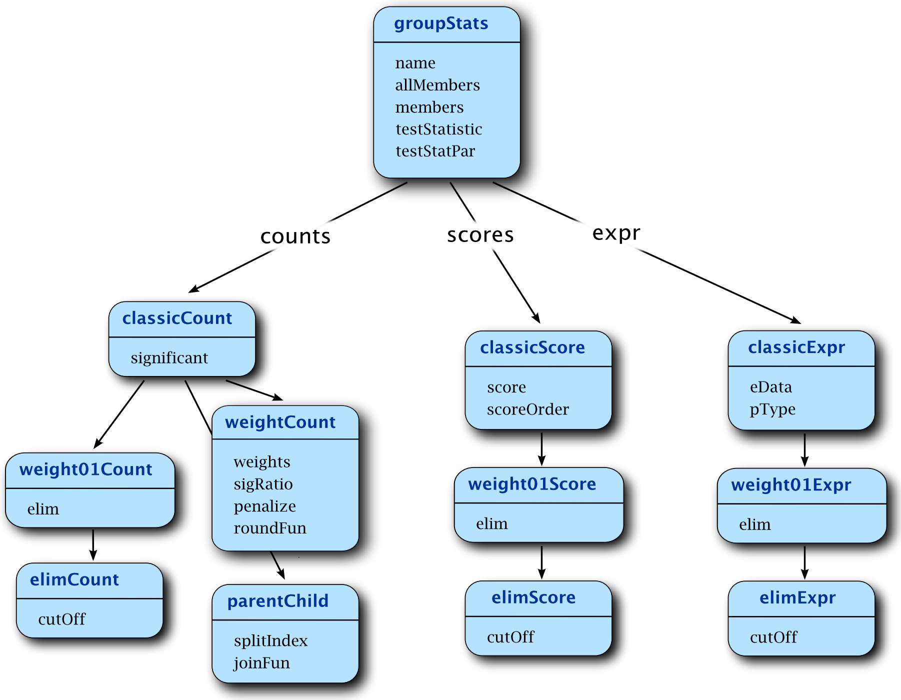


Figure 4: *The class structure....*

6 Running the group test

We are now ready to start the GO analysis.

There are three algorithms implemented in the package: classic, elim and weight. Also there are two types of test statistics which can be used, test statistics based on gene counts (like Fisher's exact test) and test statistics based on the genes scores (like Kolmogorov-Smirnov test).

There are two possibilities(interfaces) for applying a test statistic to an object of class `topGOdata`. The base interface offers a lot of flexibility to the experienced R user allowing him to implement new test statistics or new algorithms. This interface provides the hart of the testing procedure in `topGO`. The second interface is more user friendly but also more restrictive in the choice of the tests and algorithms used. We will further explain how these two interfaces work.

6.1 Defining and running the test

The main function is `getSigGroups()` which takes two arguments. The first argument is an instance of class `topGOdata` and the second argument is an instance of class `groupStats` or ....

To distinguish between all the algorithms and to secure that all test statistics are only used with the appropriate algorithms, two classes are defined for each algorithm.

To better understand this principle consider the following example. Assume we decided to apply the classic *algorithm. The two classes defined for this algorithm are* `classicCount` *and* `classicScore`*. If an object of this class is given as a argument to* `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.*

According to this mechanism, one first defines a test statistic for the chosen algorithm, in this case classic and then runs the algorithm (see the 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 `topGO` package was used. A user can define his own test statistic function and then apply it using the classic algorithm. (For example a function which computes the Z score can be implemented using as an example the `GOFisherTest` method.)

Next we show how an instance of the `groupStats` class can be used to perform the test statistic for a gene set. We will to compute the enrichment of *cellular lipid metabolic process*(GO:0044255) term using Fisher's exact test.

First we need to define which is the gene universe, to obtain the genes annotated to GO:0044255 and define the set of significant genes.

```
> goID <- "GO:0044255"
> gene.universe <- genes(GOdata)
> go.genes <- genesInTerm(GOdata, goID)[[1]]
> sig.genes <- sigGenes(GOdata)
```

Now we can instantiate an object of class `classicCount`. Once the object is constructed we can get the 2×2 contingency table or apply the test statistic.

```
> my.group <- new("classicCount", testStatistic = GOFisherTest, name = "fisher",
+     allMembers = gene.universe, groupMembers = go.genes, sigMembers = sig.genes)
> contTable(my.group)
```

```
        sig notSig
anno     26    159
notAnno 292   3122
```

```
> runTest(my.group)
```

```
[1] 0.01053972
```

We can see that ....

Please note that the `groupStats` class or any ... does not depend on GO, and an object of this class can be constructed using any gene universe/gene set.

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

A short summary on the used test and the results is printed at the R console.

```
> resultFisher
```

```
Description: Simple session
Ontology: BP
'classic' algorithm with the 'fisher' test
470 GO terms scored: 44 terms with p < 0.01
Annotation data:
    Annotated genes: 315
    Significant genes: 50
    Min. no. of genes annotated to a GO: 10
    Nontrivial nodes: 435
```

To use the Kolmogorov-Smirnov (KS) test we need to define a test based on a class which contains the gene score information

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

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

The mechanism presented above for classic also hold for elim and weight with the only remark that for the weight algorithm no test based on gene scores is implemented. To run the elim algorithm with Fisher's exact test one needs to write:

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

Similarly, for the weight algorithm one types:

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

6.2 The adjustment of p-values

The p-values return by the `getSigGroups` function are row p-values. There is no multiple testing correction applied to them, unless the test statistic directly incorporate such a correction. Of course, the researcher can perform an adjustment of the p-values if he considers it is important for the analysis. The reason for not automatically correcting for multiple testing are:

- In many cases the row p-values return by an enrichment analysis are not that extreme. A FDR/FWER adjustment procedure can in this case produce very conservative p-values and declare no, or very few, terms as significant. This is not necessary a bad thing, but it can happen that there are interesting GO terms which didn't make it over the cutoff but they are omitted and thus valuable information lost. In this case the researcher might be interested in the ranking of the GO terms even though no top term is significant at a specify FDR level.
- One should keep in mind that an enrichment analysis consist of many steps and there are many assumptions done before applying, for example, Fisher's exact test on a set of GO terms. Performing a multiple testing procedure accounting only on the number of GO terms is far from being enough to control the error rate.
- For the methods that account for the GO topology like elim and weight, the problem of multiple testing is even more complicated. Here one computes the p-value of a GO term conditioned on the neighbouring terms. The tests are therefore not independent and the multiple testing theory does not directly apply. We like to interpret the p-values returned by these methods as corrected or not affected by multiple testing.

6.3 Adding a new test

Example for the Category test ....

6.4 `runTest`: a high-level interface for testing

Over the basic interface we implemented an abstract layer to provide the users with a higher level interface for running the enrichment tests. The interface is composed by a function, namely the `runTest` function and it can be used only with a predefined set of test statistics and algorithms. In fact `runTest` is a warping function for the set of commands used for defining and running a test presented in Section 6.1.

There are three main arguments that this function takes. The first argument is an object of class `topGOdata`. The second argument, named `algorithm`, is of type character and specifies which method for dealing with the GO graph structure will be used. The third argument, named `statistic`, specifies which group test statistic will be used.

To perform a classical enrichment analysis by using the classic method and Fisher's exact test, the user needs to type:

```
> resultFis <- runTest(GOdata, algorithm = "classic", statistic = "fisher")
```

Various algorithms can be easily combine with various test statistics. However not all the combinations will work, as seen in Table 1. In the case of a mismatch the function will throw an error. The `algorithm` argument is optional and if not specified the weight01 method will be used. Bellow we can see more examples using the `runTest` function.

```
> weight01.fisher <- runTest(GOdata, statistic = "fisher")
> weight01.t <- runTest(GOdata, algorithm = "weight01", statistic = "t")
> elim.ks <- runTest(GOdata, algorithm = "elim", statistic = "ks")
> weight.ks <- runTest(GOdata, algorithm = "weight", statistic = "ks") #will not work!!!
```

The last line will return an error because we cannot use the weight method with the Kolmogorov-Smirnov test. The methods and the statistical tests which are accessible via the `runTest` function are available via the following two functions:

```
> whichTests()
```

```
[1] "fisher"     "ks"         "t"          "globaltest"
```

```
> whichAlgorithms()
```

```
[1] "classic"     "elim"        "weight"      "weight01"    "parentchild"
```

There is no advantage of using the runTest() over getSigGroups() except that it is more user friendly and it gives cleaner code. However, if the user wants to define his own test statistic or implement a new algorithm based on the available `groupStats` classes, then it would be not possible to use the `runTest` function.

Finally, the function can pass extra arguments to the initialisation method for an `groupStats` object. Thus, one can specify different cutoffs for the elim method, or arguments for the weight method.

7 Viewing and interpreting the analysis results

This section present the available tools for analysing and interpreting the results of the performed tests. Both `getSigGroups` and `runTest` functions return an object of type `topGOresult`, and most of the following functions work with this object.

7.1 The `topGOresult` object

The structure of the `topGOresult` object is quite simple. It contains the p-values or the statistics returned by the test and basic informations on the used test statistic/algorithm. The information stored in the `topGOdata` object is not carried over this object, and both of these objects will be needed by the diagnostic tools.

Since the test statistic can return either a p-value or a statistic of the data, we will refer both as scores!

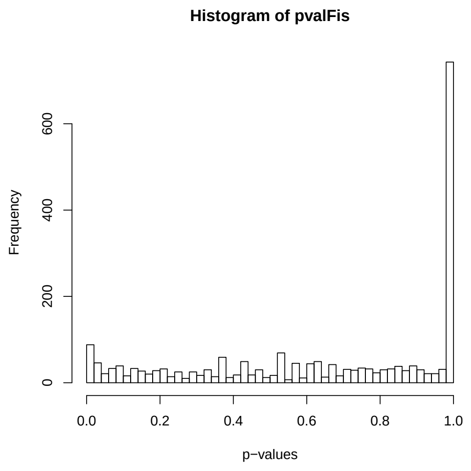
To access the stored p-values, the user should use the function `score`. It returns a named numeric vector, were the names are GO identifiers. For example, we can look at the histogram of the results of the Fisher's exact test and the classic algorithm.

By default, the `score` function does not warranty the order in which the p-values are returned, as we can see if we compare the `resultFis` object with the `resultWeight` object:

```
> head(score(resultWeight))
> pvalFis <- score(resultFis)
> head(pvalFis)

GO:0000003 GO:0000018 GO:0000041 GO:0000060 GO:0000075 GO:0000077
 0.4564572  0.6039805  0.7002201  0.8116163  0.5683847  0.4812522

> hist(pvalFis, 50, xlab = "p-values")
```



```
GO:0006605 GO:0043627 GO:0006606 GO:0030029 GO:0009190 GO:0051493
 0.9967073  0.6021982  1.0000000  0.9802640  1.0000000  1.0000000
```

However, the `score` method has a parameter, `whichGO`, which takes a list of GO identifiers and returns the scores for these terms in the specified order. Only the scores for the terms found in the intersection between the specified GOs and the GOs stored in the `topGOresult` object are returned. To see how this work lets compute the correlation between the p-values of the classic and weight methods:

```
> pvalWeight <- score(resultWeight, whichGO = names(pvalFis))
> head(pvalWeight)

GO:0000003 GO:0000018 GO:0000041 GO:0000060 GO:0000075 GO:0000077
 0.9213617  0.6039805  1.0000000  1.0000000  0.8109163  0.4812522

> cor(pvalFis, pvalWeight)

[1] 0.5053703
```

Basic information on input data can be accessed using the `geneData` function. The number of annotated genes, the number of significant genes (if it is the case), the minimal size of a GO category as well as the number of GO categories which have at least one significant gene annotated are listed:

```
> geneData(resultWeight)

  Annotated Significant    NodeSize    SigTerms 
       3599         318           5        1499 
```

7.2 Summarising the results

We can use the `GenTable` function to generate a summary table with the results from one or more tests applied to the same `topGOdata` object. The function can take a variable number of `topGOresult` objects and it returns a `data.frame` containing the top `topNodes` GO terms identified by the method specified with the `orderBy` argument. This argument allows the user decide which p-values should be used for ordering the GO terms.

	GO.ID	Term	Annotated	Significant	Expected	Rank in classic	classic	KS	elim	weight
1	GO:0050857	positive regulation of antigen receptor-...	6	5	0.53	10	2.9e-05	5e-05	2.9e-05	2.9e-05
2	GO:0045582	positive regulation of T cell differenti...	16	7	1.41	19	0.00022	0.00481	0.0499	0.00022
3	GO:0033077	T cell differentiation in the thymus	21	8	1.86	21	0.00025	0.00056	0.2471	0.00025
4	GO:0016125	sterol metabolic process	26	8	2.30	30	0.00129	0.00383	1.0000	0.00129
5	GO:0042110	T cell activation	80	23	7.07	1	1.7e-07	0.00058	0.1683	0.00225
6	GO:0050871	positive regulation of B cell activation	17	6	1.50	34	0.00242	0.01286	0.0024	0.00242
7	GO:0050853	B cell receptor signaling pathway	9	4	0.80	44	0.00527	0.04900	0.0053	0.00527
8	GO:0051209	release of sequestered calcium ion into ...	9	4	0.80	45	0.00527	0.06359	0.0053	0.00527
9	GO:0050852	T cell receptor signaling pathway	14	5	1.24	53	0.00535	0.03092	0.1832	0.00535
10	GO:0006098	pentose-phosphate shunt	5	3	0.44	56	0.00597	0.02605	0.0060	0.00597
11	GO:0009113	purine base biosynthetic process	5	3	0.44	57	0.00597	0.00666	0.0060	0.00597
12	GO:0030101	natural killer cell activation	5	3	0.44	58	0.00597	0.06312	0.0060	0.00597
13	GO:0030851	granulocyte differentiation	5	3	0.44	59	0.00597	0.04250	0.0060	0.00597
14	GO:0009156	ribonucleoside monophosphate biosyntheti...	10	4	0.88	65	0.00817	0.00571	0.0082	0.00817
15	GO:0032313	regulation of Rab GTPase activity	10	4	0.88	66	0.00817	0.13963	0.0082	0.00817
16	GO:0006631	fatty acid metabolic process	60	12	5.30	52	0.00528	0.01925	0.0053	0.01254
17	GO:0050671	positive regulation of lymphocyte prolif...	23	6	2.03	76	0.01257	0.01504	0.0126	0.01257
18	GO:0070668	positive regulation of mast cell prolife...	23	6	2.03	77	0.01257	0.01504	0.0126	0.01257
19	GO:0051668	localization within membrane	7	3	0.62	84	0.01827	0.06047	0.0183	0.01827
20	GO:0008585	female gonad development	13	4	1.15	90	0.02250	0.00318	0.0225	0.02250

Table 3: *Significance of GO terms according to different tests.*

```
> allRes <- GenTable(GOdata, classic = resultFis, KS = resultKS, elim = resultElim,
+     weight = resultWeight, orderBy = "weight", ranksOf = "classic", topNodes = 20)
```

Please note that we need to type the full names (the exact name) of the function arguments: `topNodes`, `rankOf`, etc. This is the price paid for flexibility of specifying different number of `topGOresults` objects. The table includes statistics on the GO terms plus the p-values returned by the other algorithms/test statistics. Table 3 shows the informations included in the `data.frame`.

7.3 Analysing individual GOs

Next we want to analyse the distribution of the genes annotated to a GO term of interest. In an enrichment analysis one expects that the genes annotated to a significantly enriched GO term have higher scores than the average gene' score of the gene universe.

One way to check this hypothesis is the compare the distribution of the gene scores annotated to the specified GO term with the distribution of the scores of the complementary gene set (all the genes in the gene universe which are not annotated to the GO term). This can be easily achieved using the `showGroupDensity` function. For example, lets look at the distribution of the genes annotated to the most significant GO term w.r.t. the weight algorithm.

We can see in Figure 5 that the genes annotated to GO:0050857 have low ranks (genes with low p-value of the t-test). The distribution of the ranks is skewed on the left side compared with the reference distribution given by the complementary gene set. This is a nice example in which there is a significant difference in the distribution of scores between the gene set and the complementary set, and we see from Table 3 that this GO is found as significantly enriched by all methods used.

In the above example, the genes with a p-value equal to 1 were omitted. They can be included using the value `FALSE` for the `rm.one` argument in the `showGroupDensity` function.

Another useful function for analysing terms of interest is `printGenes`. The function will generate a table with all the genes/probes annotated to the specified GO term. Various type of identifiers, the gene name and the p-values/statistics are provided in the table.

```
> goID <- allRes[10, "GO.ID"]
> gt <- printGenes(GOdata, whichTerms = goID, chip = affyLib, numChar = 40)
```

The `data.frame` containing the genes annotated to GO:0006098 is shown in Table 4. One or more GO identifiers can be given to the function using the `whichTerms` argument. When more than one GO is specified, the function returns a list of data.frames, otherwise only one `data.frame` is returned. *The function has a argument* file *which, when specified, will save the results into a file using the CSV format.*

For the moment the function will work only when the chip used has an annotation package available in Bioconductor. It will not work with other type of custom annotations.

```
> goID <- allRes[1, "GO.ID"]
> print(showGroupDensity(GOdata, goID, ranks = TRUE))
```

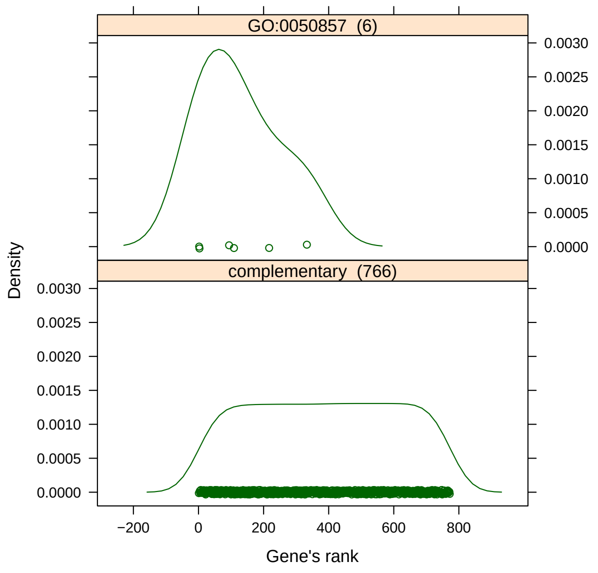


Figure 5: *Distribution of the gene' rank from GO:0050857, compared with the null distribution.*

	Chip ID	LL.id	Symbol.id	Gene name	raw p-value
36963_at	36963_at	5226	PGD	phosphogluconate dehydrogenase	6.18e-06
34003_at	34003_at	7167	TPI1	triosephosphate isomerase 1	0.00184
41221_at	41221_at	5223	PGAM1	phosphoglycerate mutase 1 (brain)	0.00222
37311_at	37311_at	6888	TALDO1	transaldolase 1	0.70146
34066_at	34066_at	9563	H6PD	hexose-6-phosphate dehydrogenase (glucos...	1.00000

Table 4: *Genes annotated to GO:0006098.*

7.4 Visualising the GO structure

An insightful way of looking at the results of the analysis is to investigate how the significant GO terms are distributed over the GO graph. We plot the subgraphs induced by the most significant GO terms reported by classic and weight methods. There are two functions available. The `showSigOfNodes` will plot the induced subgraph to the current graphic device. The `printGraph` is a warping function of `showSigOfNodes` and will save the resulting graph into a PDF or PS file.

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

```
> printGraph(GOdata, resultFis, firstSigNodes = 5, fn.prefix = "tGO", useInfo = "all",
+     pdfSW = TRUE)
> printGraph(GOdata, resultWeight, firstSigNodes = 5, fn.prefix = "tGO", useInfo = "def",
+     pdfSW = TRUE)
```

In the plots, the *significant nodes* are represented as rectangles. The plotted graph is the upper induced graph generated by these *significant nodes*. These graph plots are used to see how the significant GO terms are distributed in the hierarchy. It is a very useful tool to realize behaviour of various enrichment methods and to better understand which of the significant GO terms are really of interest.

We can emphasise differences between two methods using the `printGraph` function:

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

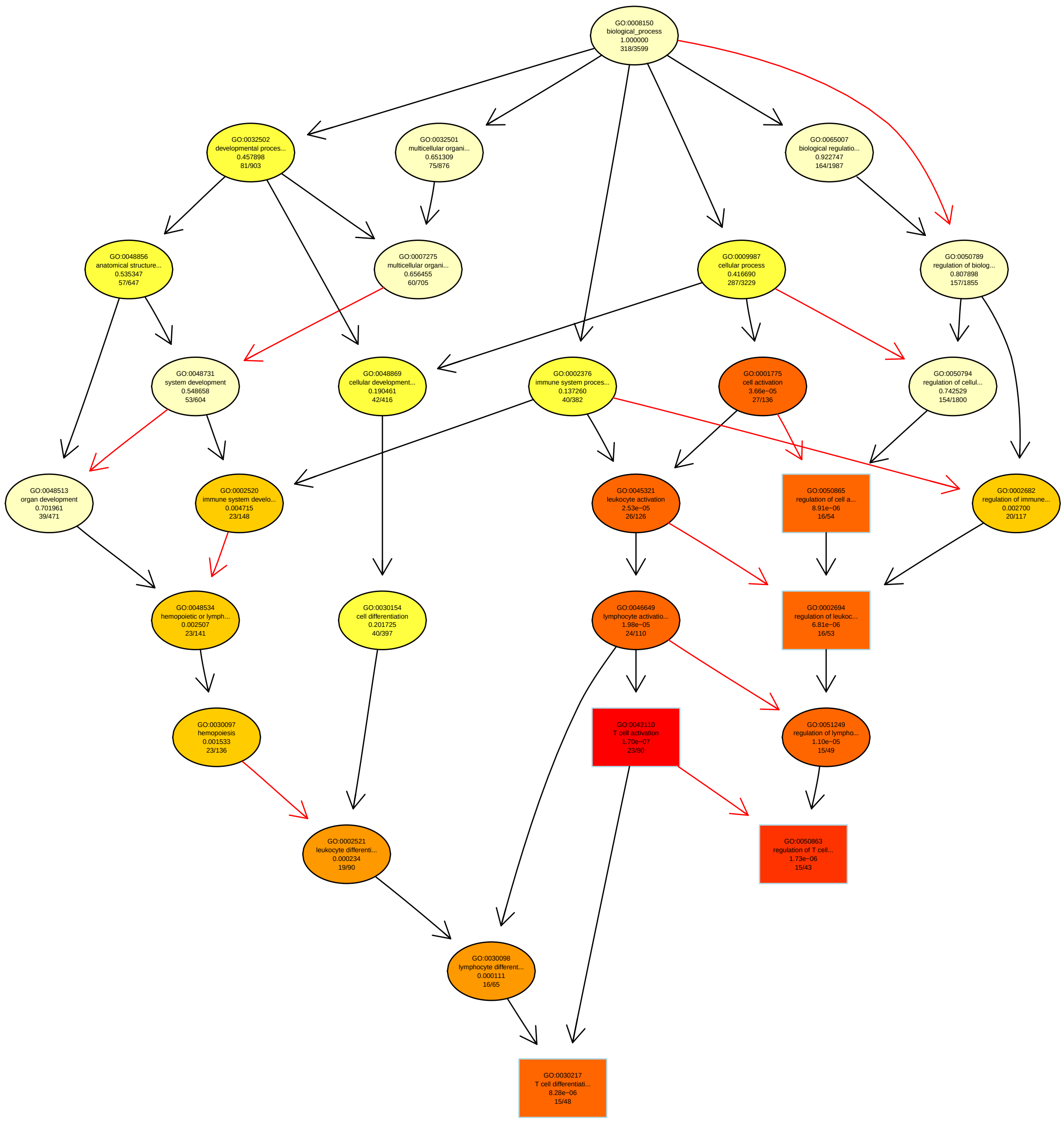


Figure 6: *The subgraph induced by the top 5 GO terms identified by the* classic *algorithm for scoring GO terms for enrichment. Boxes indicate the 5 most significant terms. Box color represents the relative significance, ranging from dark red (most significant) to light yellow (least significant). Black arrows indicate* $is\text{-}a$ *relationships and red arrows* $part\text{-}of$ *relationships.*

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

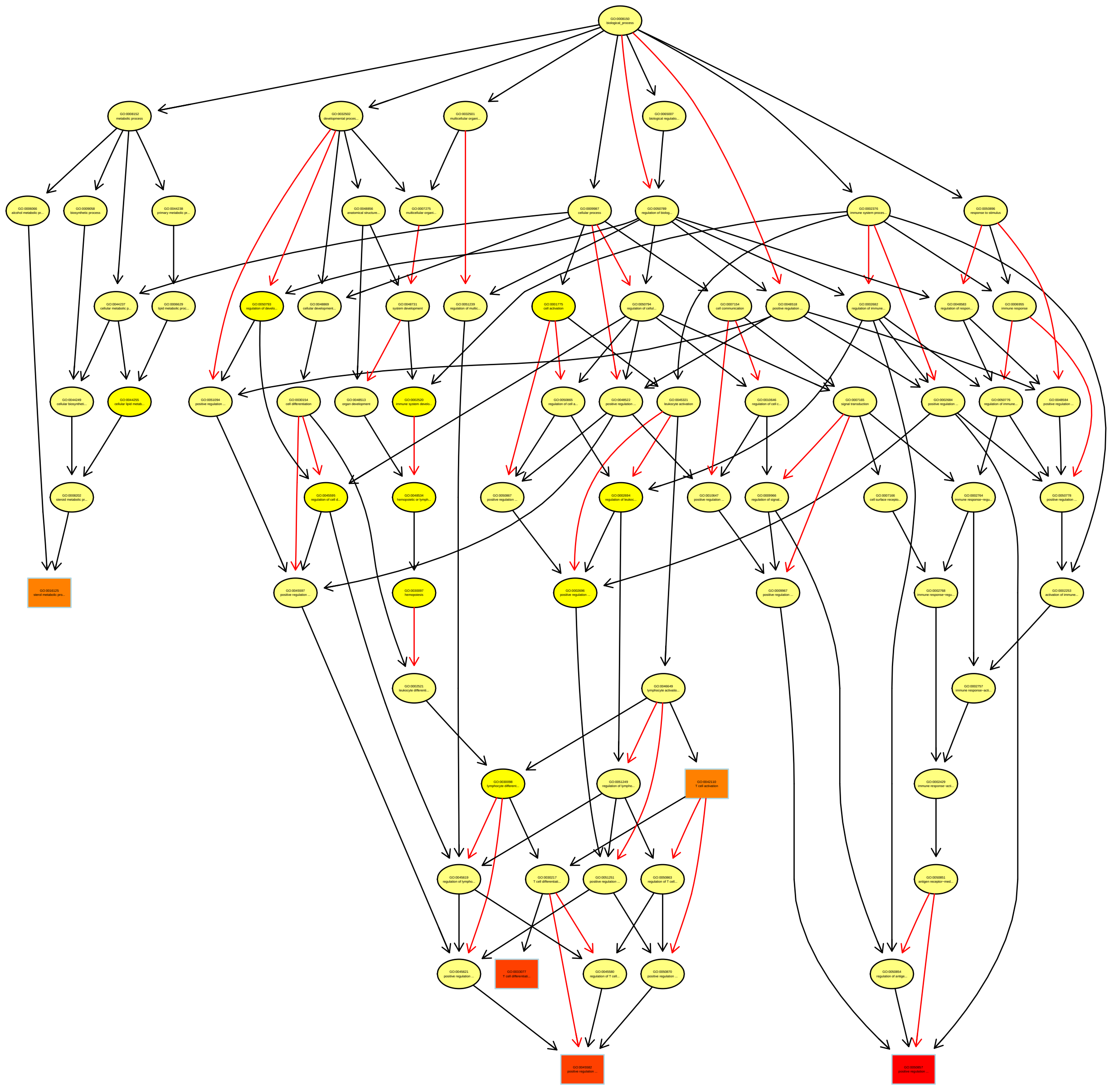


Figure 7: *The subgraph induced by the top 5 GO terms identified by the* weight *algorithm for scoring GO terms for enrichment. Boxes indicate the 5 most significant terms. Box color represents the relative significance, ranging from dark red (most significant) to light yellow (least significant). Black arrows indicate* $is\text{-}a$ *relationships and red arrows* $part\text{-}of$ *relationships.*

8 Session Information

The version number of R and packages loaded for generating the vignette were:

- R version 2.10.0 (2009-10-26), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=en_US.UTF-8`, `LC_MONETARY=C`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=en_US.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, graphics, grDevices, grid, methods, stats, tools, utils
- Other packages: ALL 1.4.7, AnnotationDbi 1.8.0, Biobase 2.6.0, DBI 0.2-4, genefilter 1.28.0, GO.db 2.3.5, graph 1.24.0, hgu95av2.db 2.3.5, lattice 0.17-26, multtest 2.2.0, org.Hs.eg.db 2.3.6, Rgraphviz 1.24.0, RSQLite 0.7-3, SparseM 0.80, topGO 1.14.0, xtable 1.5-5
- Loaded via a namespace (and not attached): annotate 1.24.0, MASS 7.3-3, splines 2.10.0, survival 2.35-7

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.