

# Gene set enrichment analysis with **topGO**

Adrian Alexa, Jörg Rahnenführer

June 4, 2010

<http://www.mpi-sb.mpg.de/~alexa>

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Sample session</b>	<b>2</b>
2.1	Step 1: Preparing the data . . . . .	2
2.2	Step 2: Running the desired tests . . . . .	3
2.3	Step 3: Analysis of results . . . . .	4
<b>3</b>	<b>Setup</b>	<b>7</b>
<b>4</b>	<b>Creating a topGOdata object</b>	<b>7</b>
4.1	Custom annotations . . . . .	8
4.2	Predefined list of interesting genes . . . . .	9
4.3	Using the genes score . . . . .	10
4.4	Filtering and missing GO annotations . . . . .	11
<b>5</b>	<b>Working with the topGOdata object</b>	<b>13</b>
<b>6</b>	<b>Running the group test</b>	<b>14</b>
6.1	Defining and running the test . . . . .	15
6.2	The adjustment of $p$ -values . . . . .	18
6.3	Adding a new test . . . . .	18
6.4	runTest: a high-level interface for testing . . . . .	18
<b>7</b>	<b>Viewing and interpreting the analysis results</b>	<b>19</b>
7.1	The topGOresult object . . . . .	19
7.2	Summarising the results . . . . .	20
7.3	Analysing individual GOs . . . . .	20
7.4	Visualising the GO structure . . . . .	22
<b>8</b>	<b>Session Information</b>	<b>25</b>
	<b>References</b>	<b>25</b>

# 1 Introduction

The **topGO** package is designed to facilitate semi-automated enrichment analysis for Gene Ontology terms. The process consists of input of normalised gene expression measurements, gene-wise correlation or differential expression analysis, enrichment analysis of GO terms, interpretation and visualisation of the results.

One of the main advantages of **topGO**, is the unified gene set testing framework it offers, which besides providing an easy to use set of functions for performing an enrichment analysis, it also enables the user to easily implement new statistical tests or new algorithms that deal with the GO graph structure. Such an unified framework also facilitates the comparison between different GO enrichment methodologies.

There are a number of test statistics and algorithms dealing with the GO graph structured ready to use in **topGO**. Table 1 presents the compatibility table between the test statistics and GO graph methods.

	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.*

The **elim**, **weight** algorithms were introduced in [Alexa 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). The **parentChild** algorithm was introduced by [Grossmann et al., 2007].

The next section provides a quick tour into **topGO** and is thought to be independent of the rest of this manuscript. The remaining 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. *Preparing the data:* The list of genes identifiers, the gene' score, the list of differentially expressed genes or a criteria for selecting genes based on their scores and the gene-to-GO annotations are all collected to form a suitable R object.
2. *Running the desired tests:* Using the object created in the first step the user can perform enrichment analysis using various statistical tests and various methods that deal with the GO topology.
3. *Analysis of the results:* The results obtained in the second step are analysed using summary functions and visualisation tools.

Before going through each of those steps the user needs to decide which biological question he would like to investigate. This most likely, together with the data available at hand, will dictate which test statistic/methods he needs to use.

In this section we will test the enrichment of GO terms with differentially expressed genes using two statistical tests, namely Kolmogorov-Smirnov test and Fisher's exact test.

### 2.1 Step 1: Preparing the data

In the first 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 correspondent  $p$ -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 an 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 statistics.

```
> 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 main arguments. The first argument needs to be an object of class `topGOdata`. The second and the third arguments are of type character and specifies which method for dealing with the GO graph structure will be used and the test statistic, respectively.

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")
```

	GO.ID	Term	Annotated	Significant	Expected	Rank in classicFisher	classicFisher	classicKS	elimKS
1	GO:0007067	mitosis	179	18	28.32	544	0.9996	2.4e-09	2.4e-09
2	GO:0051301	cell division	139	15	21.99	540	0.9909	1.0e-06	1.0e-06
3	GO:0022403	cell cycle phase	192	28	30.38	439	0.8187	6.0e-11	0.00017
4	GO:0033077	T cell differentiation in the thymus	12	8	1.90	1	7e-05	0.00025	0.00025
5	GO:0000278	mitotic cell cycle	194	28	30.70	453	0.8442	1.6e-10	0.00117
6	GO:0007051	spindle organization	24	2	3.80	500	0.9200	0.00279	0.00279
7	GO:0002706	regulation of lymphocyte mediated immuni...	19	7	3.01	102	0.0181	0.00539	0.00539
8	GO:0045449	regulation of transcription	49	12	7.75	163	0.0599	0.00574	0.00574
9	GO:0042089	cytokine biosynthetic process	19	6	3.01	164	0.0609	0.00647	0.00647
10	GO:0030183	B cell differentiation	13	6	2.06	72	0.0083	0.00834	0.00834

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

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

```
> resultFisher
```

Description: Simple session

Ontology: BP

'classic' algorithm with the 'fisher' test

581 GO terms scored: 76 terms with  $p < 0.01$

Annotation data:

Annotated genes: 316

Significant genes: 50

Min. no. of genes annotated to a GO: 10

Nontrivial nodes: 549

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 not 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 employed.

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

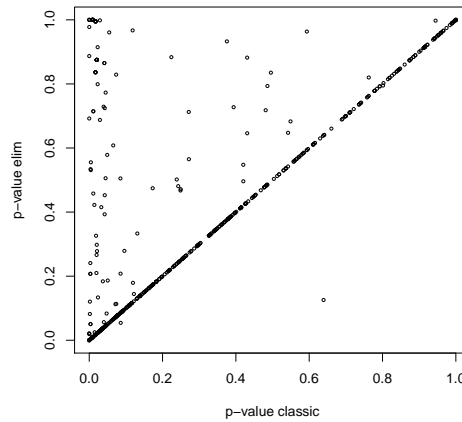
The `GenTable` function returns a data.frame containing the top `topNodes` GO terms identified by the elim algorithm, see `orderBy` argument. The data.frame includes some statistics on the GO terms and the  $p$ -values obtained from the methods passed as arguments. Table 2 shows the results.

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, 47 terms with a  $p$ -value less than 0.01 in the classic method, one would expect that only few GO terms will have different

```

> 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.*

*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 which are found significant by the `classic` method are less significant in the `elim`, as expected.

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")

```



### 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 features.

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 `GOMPTerm` 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 *hgu95av2* array.

### 4 Creating a topGOdata object

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

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 below.

**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 or 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 annotations of 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 a rather small size gene universe which also imply a moderate size GO ontology, especially since we are using the MF ontology.

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 various group testing procedure, which let us test multiple hypothesis or tune 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 be applied using the function `getPvalues` (a wrapping 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 Section 4.2

```
> GOdata <- new("topGOdata", description = "GO analysis of ALL data; B-cell vs T-cell",
+   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.

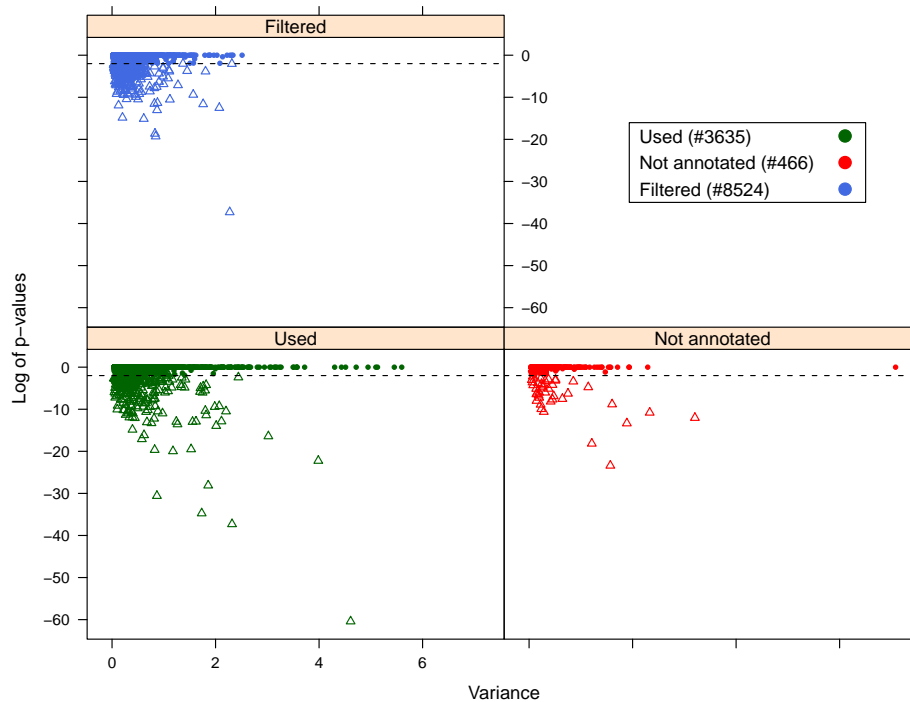
## 4.4 Filtering and missing GO annotations

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.

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
```



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

```
> 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
      3635          466      8524
```

Out of the filtered probes only 89% 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] 3635
```

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 = 2553  
Number of Edges = 5134

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

```
[1] "GO:0000002" "GO:0000003" "GO:0000018" "GO:0000038" "GO:0000041" "GO:0000045"
```

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)
```

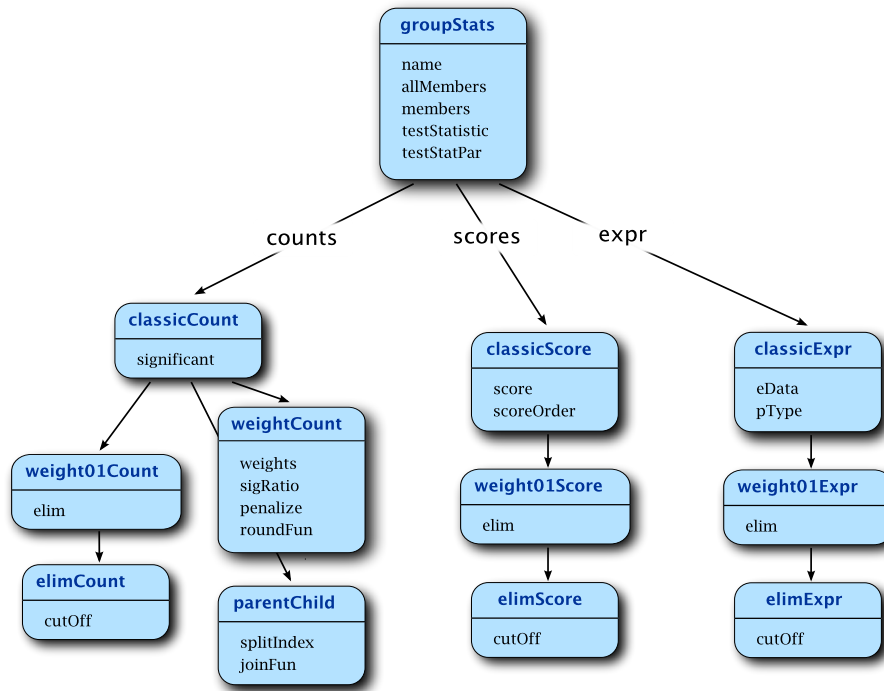
	Annotated	Significant	Expected
GO:0009880	5	0	0.43
GO:0006690	21	4	1.82
GO:0002532	7	0	0.61
GO:0012502	142	18	12.31
GO:0006418	19	1	1.65
GO:0050886	8	0	0.69
GO:0006325	160	14	13.87
GO:0014072	7	0	0.61
GO:0032270	124	5	10.75
GO:0045103	7	1	0.61

## 6 Running the group test

In this section we explain how we can run the desired enrichment method once the `topGOdata` object is available.

`topGO` package was designed to work with various test statistics and various algorithms which take the GO dependencies into account. At the base of this design stands a S4 class mechanism which facilitates defining and executing a (new) group test. Three types of enrichment tests can be distinguish if we look at the data used by the each test.

- Tests based on gene **counts**. This is the most popular family of tests, given that it only requires the presence of a list of interesting genes and nothing more. Tests like Fisher's exact test, Hypergeometric test and binomial test belong to this family. [Draghici et al., 2006]



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

- Tests based on gene **scores** or gene **ranks**. It includes Kolmogorov-Smirnov like tests (also known as GSEA), Gentleman’s Category, *t*-test, etc. []
- Tests based on gene **expression**. Tests like Goeman’s globaltest or GlobalAncova separates from the others since they work directly on the expression matrix. []

There are also a number of strategies/algorithms to account for the GO topology, see Table 1, each of them having specific requirements.

For each test type described above and for each algorithm there is S4 class defined in the package. The main idea is to have a class (container) which can store, for a specified gene set(GO category), all data necessary for computing the desired test statistic, and a method that will iterate over all GO categories. In such a design the user needs to instantiate an object from the class corresponding to the chosen method(test statistic and algorithm) and then run the iterator function on this object.

The defined S4 classes are organised in a hierarchy which is showed in Figure 4.

There are two possibilities(or interfaces) for applying a test statistic to an object of class `topGOdata`. The basic interface, which provides the core of the testing procedure in `topGO`, offers more flexibility to the experienced R user allowing him to implement new test statistics or new algorithms. The second interface is more user friendly but at the same time 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 for running the GO enrichment is `getSigGroups()` and it takes two arguments. The first argument is an instance of class `topGOdata` and the second argument is an instance of class `groupStats` or any of its descendents.

*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.*

## The groupStats classes

Next we show how an instance of the `groupStats` class can represent a gene set and how the test statistic is performed.

We compute the enrichment of *cellular lipid metabolic process* (GO:0044255) term using Fisher's exact test. In order to do this we need to define the gene universe, to obtain the genes annotated to GO:0044255, and to 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  $2 \times 2$  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	20	125
notAnno	295	3195

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

```
[1] 0.02341345
```

The slot `testStatistic` contains the function (or to be more precise, the method) which computes the test statistic. We used the `GOFisherTest` function which is available in the `topGO` package and as the name states it implements Fisher's exact test. The user can define his own test statistic function and then apply it using the preferred algorithm. The function, however, should use the methods defined for the `groupStats` class to access the data encapsulated in such an object. (For example a function which computes the  $Z$  score can be easily implemented using as an example the `GOFisherTest` method.)

The `runTest` method is defined for the `groupStats` class and its used to run/compute the test statistic, by calling the `testStatistic` function. The value returned by the `runTest` method in this case is the value returned by `GOFisherTest` method, which is the Fisher's exact test  $p$ -value. The `contTable` method, showed in the example above, is only defined for the classes based on gene counts and its used to compile the two-dimensional contingency table based on the object.

To show how the same interface is used for the classes based on gene counts we next build an instance for the `elimCount` class. We randomly select 25% of the annotated genes as genes that should be removed.

```
> set.seed(123)
> elim.genes <- sample(go.genes, length(go.genes)/4)
> elim.group <- new("elimCount", testStatistic = GOFisherTest, name = "fisher",
+   allMembers = gene.universe, groupMembers = go.genes, sigMembers = sig.genes,
+   elim = elim.genes)
> contTable(elim.group)
```

	sig	notSig
anno	13	96
notAnno	295	3195

```
> runTest(elim.group)
```



```
[1] 0.1362211
```

We see that the interface accounts for the genes that need to be eliminated, once the object is instantiated. The same mechanism applies for the other hierarchy of classes (the score based and expression based classes), except that each hierarchy has its own specialised methods for computing statistics from the data.

Please note that the `groupStats` class or any descendent class does not depend on GO, and an object of such a class can be instantiated using any gene set.

## Performing the test

According to the mechanism described above, one first defines a test statistic for the chosen algorithm, meaning that an instance of object specific for the algorithm is constructed in which only the test statistic must be specified, and then calls a *generic function* (interface) to run the algorithm.

*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.)*

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

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

```
> resultFisher
```

```
Description: GO analysis of ALL data; B-cell vs T-cell Object modified on: 04 Jun 2010
Ontology: BP
'classic' algorithm with the 'Fisher test' test
2553 GO terms scored: 103 terms with p < 0.01
Annotation data:
  Annotated genes: 3635
  Significant genes: 315
  Min. no. of genes annotated to a GO: 5
  Nontrivial nodes: 1780
```

To use the Kolmogorov-Smirnov (KS) test one needs to provide the gene-wise scores and thus we need to instantiate an object of a class which is able to deal of the scores. Such a class is the `classicScore` class, see Figure 4 which will let us run the classic algorithm.

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

The mechanism presented above for classic also hold for `elim` and `weight`. The user should pay attention to the compatibility between the chosen class and the function for computing the test statistic, since no incompatibility test are made when the object is instantiated. For example the `weight` algorithm will not work with classes based on gene-wise scores. To run the `elim` algorithm with KS test one needs to type:

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

Similarly, for the `weight` algorithm with Fisher's exact test 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, which 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.

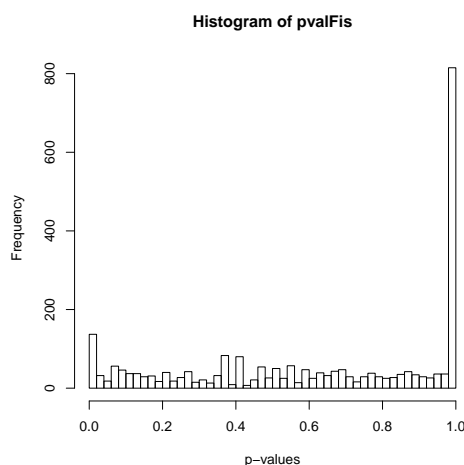
*Since the test statistic can return either a  $p$ -value or a statistic of the data, we will refer them as scores!*

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.

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

```
GO:0000003 GO:0000018 GO:0000038 GO:0000041 GO:0000060 GO:0000075
0.6886856 0.1347948 0.0627876 0.6928494 0.8376279 0.4010151
```

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



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))
```

```
GO:0042278 GO:0006919 GO:0051963 GO:0033089 GO:0030029 GO:0045047
1.000000000 0.672622613 0.005648273 1.000000000 0.905098428 0.419727761
```

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:0000038 GO:0000041 GO:0000060 GO:0000075
0.9931733 1.0000000 1.0000000 0.6928494 1.0000000 1.0000000
```

```
> cor(pvalFis, pvalWeight)
```

```
[1] 0.4848243
```

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
3635	315	5	1780

## 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.

```
> allRes <- GenTable(GOdata, classic = resultFis, KS = resultKS, 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

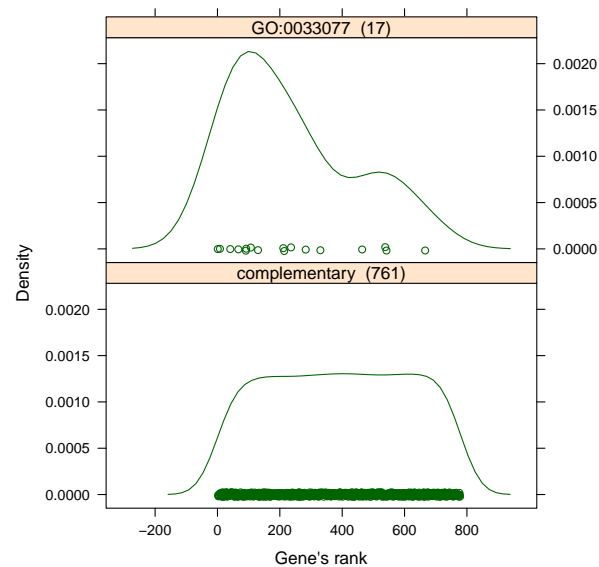
Next we want to analyse the distribution of the genes annotated to a GO term of interest. In an enrichment analysis one expects that the genes annotated to a significantly enriched GO term have higher scores than the average gene' score of the gene universe.

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.

	GO.ID	Term	Annotated	Significant	Expected	Rank in classic	classic	KS	weight
1	GO:0033077	T cell differentiation in the thymus	31	12	2.69	4	4.5e-06	1.9e-05	4.5e-06
2	GO:0050857	positive regulation of antigen receptor...	6	5	0.52	10	2.6e-05	4.8e-05	2.6e-05
3	GO:0042110	T cell activation	107	26	9.27	1	7.0e-07	0.00062	0.00014
4	GO:0050852	T cell receptor signaling pathway	21	8	1.82	24	0.00022	0.00807	0.00022
5	GO:0050671	positive regulation of lymphocyte prolif...	29	9	2.51	29	0.00052	0.01067	0.00052
6	GO:0001912	positive regulation of leukocyte mediate...	6	4	0.52	34	0.00072	0.00554	0.00072
7	GO:0045086	positive regulation of interleukin-2 bio...	6	4	0.52	35	0.00072	0.01288	0.00072
8	GO:0045588	positive regulation of gamma-delta T cel...	6	4	0.52	36	0.00072	0.01288	0.00072
9	GO:0050871	positive regulation of B cell activation	21	7	1.82	50	0.00137	0.01015	0.00137
10	GO:0045577	regulation of B cell differentiation	13	5	1.13	66	0.00340	0.00319	0.00340
11	GO:0016125	sterol metabolic process	31	8	2.69	67	0.00389	0.01029	0.00389
12	GO:0006631	fatty acid metabolic process	64	13	5.55	61	0.00274	0.04269	0.00410
13	GO:0050850	positive regulation of calcium-mediated ...	9	4	0.78	71	0.00491	0.01622	0.00491
14	GO:0006098	pentose-phosphate shunt	5	3	0.43	78	0.00565	0.02578	0.00565
15	GO:0030854	positive regulation of granulocyte diffe...	5	3	0.43	79	0.00565	0.05140	0.00565
16	GO:0031952	regulation of protein amino acid autopho...	5	3	0.43	80	0.00565	0.05328	0.00565
17	GO:0051963	regulation of synaptogenesis	5	3	0.43	81	0.00565	0.04709	0.00565
18	GO:0009168	purine ribonucleoside monophosphate bios...	10	4	0.87	92	0.00763	0.03479	0.00763
19	GO:0032313	regulation of Rab GTPase activity	10	4	0.87	93	0.00763	0.13671	0.00763
20	GO:0006107	oxaloacetate metabolic process	6	3	0.52	104	0.01058	0.10759	0.01058

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

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



**Figure 5:** Distribution of the gene' rank from GO:0033077, compared with the null distribution.

For example, let's 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:0033077 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)
```

	Chip ID	LL.id	Symbol.id	Gene name	raw p-value
40518_at	40518_at	5788	PTPRC	protein tyrosine phosphatase, receptor t...	4.33e-07
33869_at	33869_at	6777	STAT5B	signal transducer and activator of trans...	1.48e-05
40520_g_at	40520_g_at	5788	PTPRC	protein tyrosine phosphatase, receptor t...	0.000758
506_s_at	506_s_at	6776	STAT5A	signal transducer and activator of trans...	0.001317
1506_at	1506_at	3561	IL2RG	interleukin 2 receptor, gamma (severe co...	0.003984
1065_at	1065_at	2322	FLT3	fms-related tyrosine kinase 3	1.000000
1630_s_at	1630_s_at	6850	SYK	spleen tyrosine kinase	1.000000
172_at	172_at	3635	INPP5D	inositol polyphosphate-5-phosphatase, 14...	1.000000
255_s_at	255_s_at	3623	INHA	inhibin, alpha	1.000000
266_s_at	266_s_at	100133941	CD24	CD24 molecule	1.000000
34583_at	34583_at	2322	FLT3	fms-related tyrosine kinase 3	1.000000
36885_at	36885_at	6850	SYK	spleen tyrosine kinase	1.000000
38578_at	38578_at	939	CD27	CD27 molecule	1.000000

**Table 4:** Genes annotated to GO:0045577.

The `data.frame` containing the genes annotated to GO:0045577 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.

## 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.

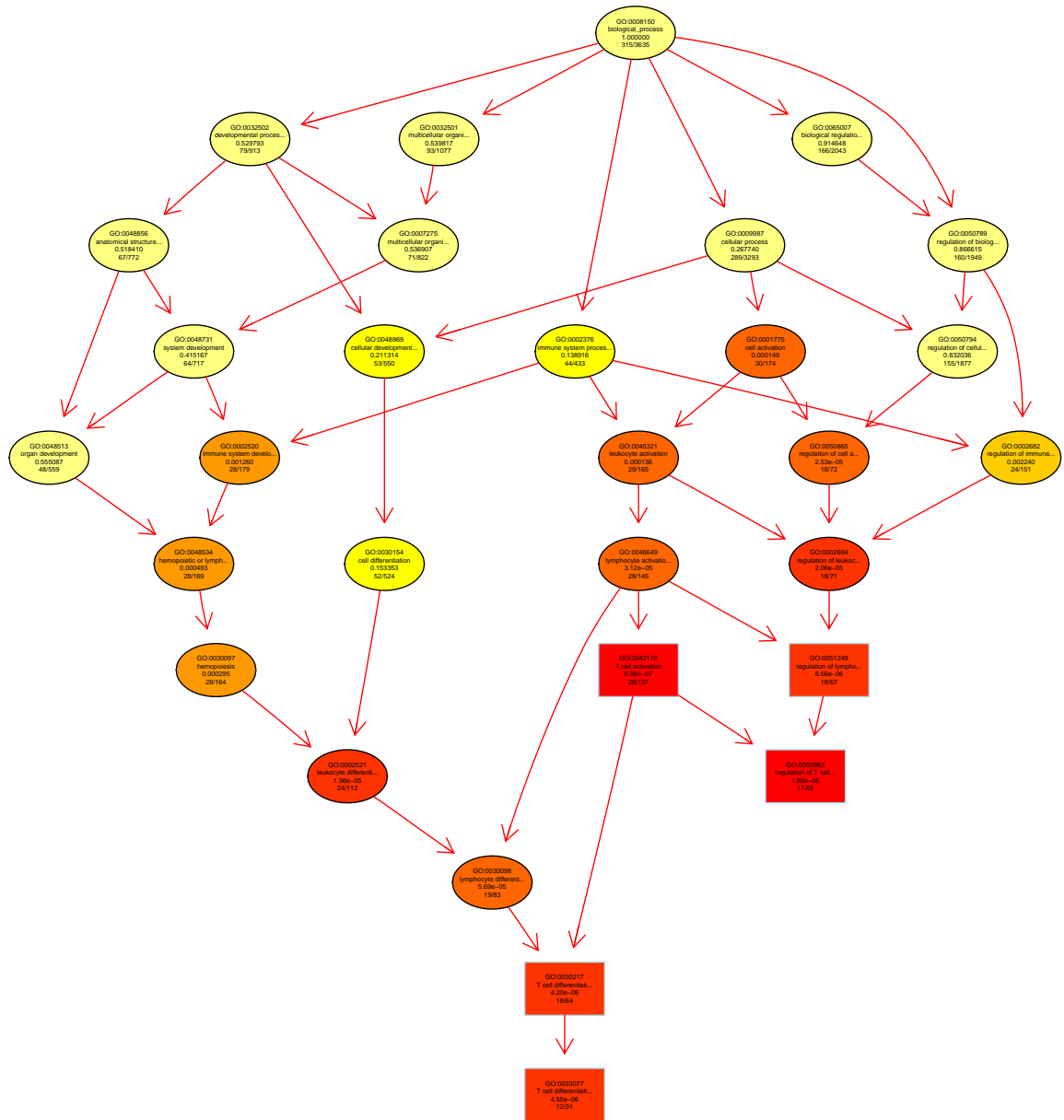
```
> showSigOfNodes(GOdata, score(resultFis), firstSigNodes = 5, useInfo = "all")
> showSigOfNodes(GOdata, score(resultWeight), firstSigNodes = 5, useInfo = "def")

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

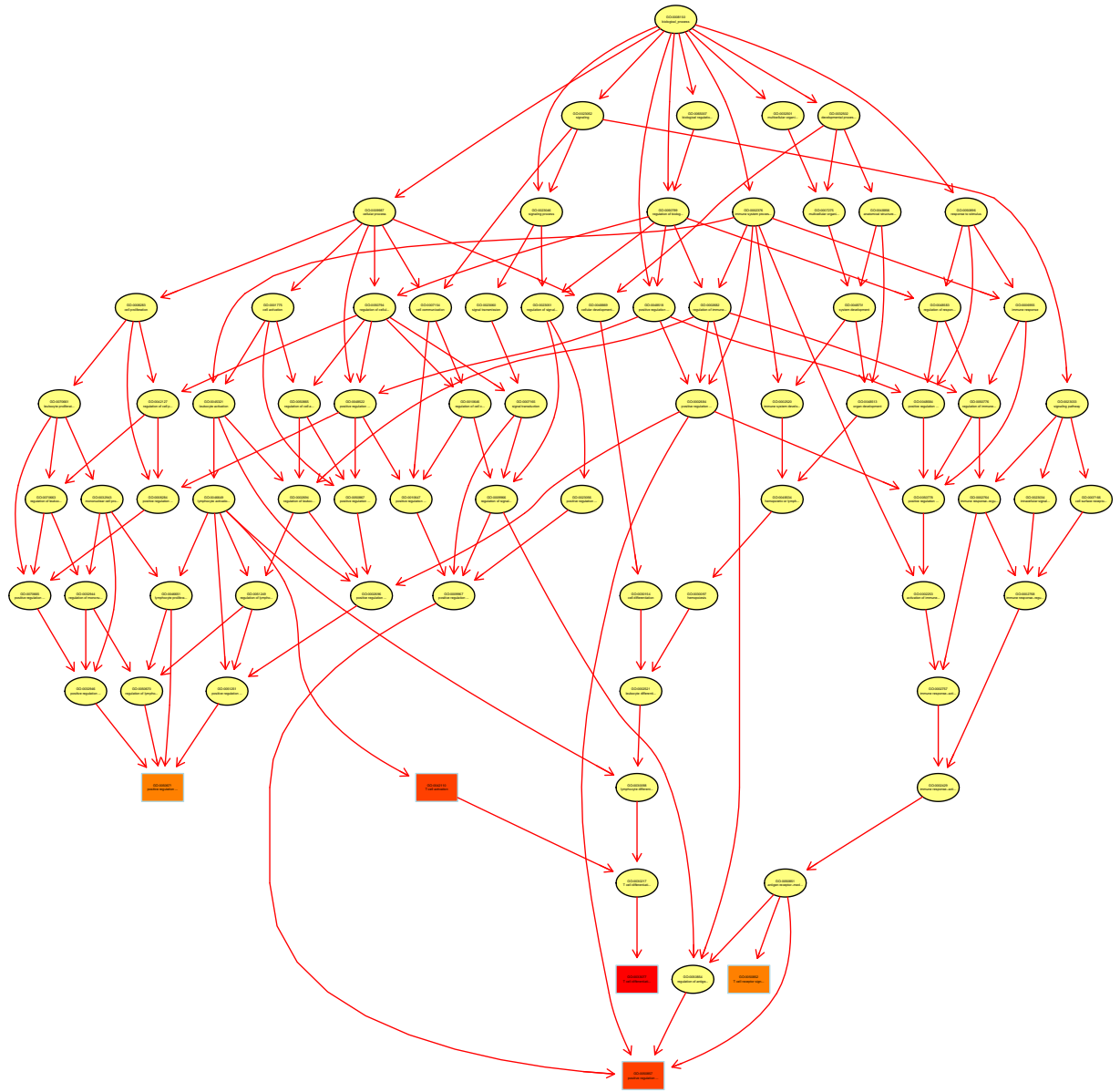
In the plots, the *significant nodes* are represented as rectangles. The plotted graph is the upper induced graph generated by these *significant nodes*. These graph plots are used to see how the significant GO terms are distributed in the hierarchy. It is a very useful tool to realize behaviour of various 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")
> printGraph(GOdata, resultElim, firstSigNodes = 15, resultFis, fn.prefix = "tGO",
+   useInfo = "all")
```



**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-a* relationships and red arrows *part-of* relationships.



**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-a* relationships and red arrows *part-of* relationships.



## 8 Session Information

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

- R version 2.11.0 (2010-04-22), x86\_64-unknown-linux-gnu
- Locale: LC\_CTYPE=en\_US, LC\_NUMERIC=C, LC\_TIME=en\_US, LC\_COLLATE=C, LC\_MONETARY=C, LC\_MESSAGES=en\_US, LC\_PAPER=en\_US, LC\_NAME=C, LC\_ADDRESS=C, LC\_TELEPHONE=C, LC\_MEASUREMENT=en\_US, LC\_IDENTIFICATION=C
- Base packages: base, datasets, grDevices, graphics, grid, methods, stats, tools, utils
- Other packages: ALL 1.4.7, AnnotationDbi 1.10.0, Biobase 2.8.0, DBI 0.2-5, GO.db 2.4.1, RSQLite 0.8-4, Rgraphviz 1.26.0, SparseM 0.85, genefilter 1.30.0, graph 1.26.0, hgu95av2.db 2.4.1, lattice 0.18-5, multtest 2.4.0, org.Hs.eg.db 2.4.1, topGO 1.16.2, xtable 1.5-6
- Loaded via a namespace (and not attached): MASS 7.3-5, annotate 1.26.0, splines 2.11.0, survival 2.35-8

## References

- [Alexa et al., 2006] Alexa, A., Rahnenführer, J., and Lengauer, T. (2006). Improved scoring of functional groups from gene expression data by decorrelating go graph structure. *Bioinformatics (Oxford, England)*, 22:1600–1607. 10.1093/bioinformatics/btl140.
- [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.
- [Draghici et al., 2006] Draghici, S., Sellamuthu, S., and Khatri, P. (2006). Babel’s tower revisited: a universal resource for cross-referencing across annotation databases. *Bioinformatics (Oxford, England)*, 22:bt1372v1–2939. 10.1093/bioinformatics/btl372.
- [Goeman and Bühlmann, 2007] Goeman, J. J. and Bühlmann, P. (2007). Analyzing gene expression data in terms of gene sets: methodological issues. *Bioinformatics (Oxford, England)*, 23:980–987. 10.1093/bioinformatics/btm051.
- [Grossmann et al., 2007] Grossmann, S., Bauer, S., Robinson, P. N., and Vingron, M. (2007). Improved detection of overrepresentation of gene-ontology annotations with parent child analysis. *Bioinformatics*, 23:3024. 10.1093/bioinformatics/btm440.
- [Yon Rhee et al., 2008] Yon Rhee, S., Wood, V., Dolinski, K., and Draghici, S. (2008). Use and misuse of the gene ontology annotations. *Nature reviews. Genetics*, advanced online publication:509–515. 10.1038/nrg2363.