Gene Set Enrichment Analysis

Assignment 5

Genomorientierte Bioinfromatik

Malte Weyrich

February 2025

When performing *Enrichment Analysis* in bioinformatics, there are several different tools and methods available to determine whether certain gene sets are statistically enriched. These tools usually approach this problem in different ways, using different statistical tests and assumptions. The two main methods are *Gene Set Over-representation Analysis (ORA)* and *Gene Set Enrichment Analysis (GSEA)*. This report compares a custom Java implementation of both methods for a given list of differentially expressed genes and a standard of truth gene set, with two already existing tools, called *gProfiler (perfoms ORA)* and *fgsea (performs GSEA)*. The Java implementation is also evaluated in terms of runtime and the properties of the *Directed Acyclic Graph (DAG)* used for the analysis.

Contents

1 — Introduction

Due to the vast amount of different genes and gene products, a vocabulary for describing groups of genes and their logical relationships called ***Gene Ontology (GO)***, was invented by the ***Gene Ontology Consortium (GOC)*** (Ashburner et al. 2000). In *Gene Ontology*, genes are sorted into different *gene sets* (referred to as *GO Terms*), with each gene set having its own ID. These gene sets are annotated with a function and a process in which the genes of the set are involved. A gene set can be a component of several other gene sets through an *is_a relationship*, which leads to a hierarchical, tree-like structure. In this case, the structure is a ***Directed Acyclic Graph (DAG)***. The biological functions and processes are sorted into three main categories: *Biological Process (BP)*, *Molecular Function (MF)* and *Cellular Component (CC)*, each forming its own *DAG*. The root of each *DAG* comprises all genes of the *DAG* and is labeled with the one of the *GO Terms*. The amount of genes per gene set decreases the further its location is relative to the root. The same goes for the annotations of gene sets, which get more exact and detailed the further they are away from the root.

In bioinformatics, this vocabulary is crucial for determining the effects of differentially expressed genes in RNA-seq and other experiments. Knowing which genes are significantly up or down-regulated is not sufficient enough for understanding the biological processes that are affected by the changes in gene expression. This is where ***Gene Set Enrichment Analysis (GSEA)*** and ***Gene Over-representation Analysis (ORA)*** come into play. Although these two methods differ in their methodology, the main goal is to determine whether certain gene sets are statistically enriched, providing insights into affected biological pathways.

The analysis conducted in this report tries to compare the results of both methods for a given list of differentially expressed genes and standard of truth gene sets. Additionally, our results are compared to two already existing tools for *ORA*, called ***gProfiler*** (Kolberg et al. 2023) and ***fgsea*** (Sergushichev 2016) for *GSEA*.

2 — Materials & Methods

2.1. Enrichment Methods

2.1.1 Gene Set Over-representation Analysis

In *ORA*, the first step is to define a gene list L of **possible interesting genes** (genes having $padj \leq 0.05$ or $abs(lfc) \geq 1.5$). Then, for each gene set S, an *enrichment p-value* is calculated, representing the probability of observing at least as many genes from L in S, assuming a random selection from the background gene set (Huang, Sherman, and Lempicki 2008). This is commonly simulated using either a *Hypergeometric Test* or the *Fisher's Exact Test*. In our case, we used *Fisher's Exact Test* with the following parameters:

$$P(X \geq k) = 1 - \frac{\binom{K}{k} \cdot \binom{N-K}{n-k}}{\binom{N}{n}}$$

, where

- $N := |G \cap DE|$
- $n := |g \cap DE|$
- $K := |G \cap sDE|$
- $k := |g \cap sDE|$

, with

- $G :=$ All gene symbols of current `DAG`
- $g :=$ Gene symbols of current `GOEntry`
- $DE :=$ All observed differentially expressed genes
- $sDE :=$ Subset of $sDE \subseteq DE$ considered significant

After calculating all *enrichment p-values*, they are corrected for multiple testing using the *Benjamini-Hochberg* method.

One problem in *ORA* is that we have to define the threshold and choose, whether we only want to analyze up-regulated or down-regulated genes, or both, which is not the case in *GSEA*.

2.1.2 Gene Set Enrichment Analysis

GSEA use a different approach to determine the significance of gene sets. It consists of three main steps:

1. Calculate the *Enrichment Score (ES)* of a gene set
2. Evaluate the significance of the calculated Enrichment Score (through *empirical p-value*)
3. Correct the p-values for multiple testing (using *Benjamini-Hochberg*)

In order to calculate the *ES* for a given gene set S, the observed differentially expressed genes of an experiment are ranked based on their *logFoldChange*, resulting in a *ranked list* L. Important here is that all observed genes are used, whereas in 2.1.1 only the significantly differentially expressed genes are used. The *ES* is then calculated by walking down the ranked list L and calculating a *running sum statistic* which increases when a gene is part of the gene set S and decreases when it is not. The maximum value that is encountered during the walk is the *ES* of the gene set S. The *running sum statistic* utilizes a *Kolmogorov-Smirnov* like statistic that weights genes according to their *logFoldChange* magnitude (Subramanian et al. 2005).

The provided *JAR* imports the `KolmogorovSmirnovTest Class` from *org.apache.commons.math3.stat.inference* in order to calculate the *Kolmogorov-Smirnov* statistic and p-value. We do not calculate the *empirical p-value* for the *ES* as it was not required in the assignment, but usually, it is more than required to run a permutation test to determine the true significance of the *ES* in order to avoid too many false positives.

2.2. Java Programm

The provided *JAR* has the following input specification:

```
java -jar go_enrichment.jar
  -obo                    Path to obo file.
  -root                   One of three Ontology Terms:
                          ["molecular_function",
                          "biological_process",
                          "cellular_component"].
  -mapping                Path to mapping file (SYMBOL -> GO_ID).
  -mappingtype            Format of the mapping-File (go|ensembl).
                          Has to agree with "-mapping" option.
  [-overlapout]           Information about DAG  entries  with shared mapped
                          genes is written into this file.
  -enrich                 Path to enrichment analysis file.
  -o                      Path to output file.
  -minsize                Min amount of genes per GO entry.
  -maxsize                Max amount of genes per GO entry.
```

The specific file formats are described in *Assignment 5*.

2.2.1 Logic

(A) Parsing Input Files:

The provided *JAR* reads in the necessary input files and stores them for later use. Parsing the *obo* file is the first step since it contains the structure of the `DAG`, which itself is an object. The parser only considers entries that map the parameter *"-root"* and ignores entries that are marked as obsolete. It stores entries in a `HashMap<String, GOEntry> nodeMap` to keep track of already-seen gene sets. For each entry, we check if its GO ID is contained in our `nodeMap`. If not, we append a new `GOEntry` object to `nodeMap` and add its parents to the newly created object. Otherwise, we take the already existing `GOEntry` object inside the `nodeMap` and update its parents. Of course, we also have to look up parent GO IDs inside `nodeMap` before creating a new `GOEntry` object. This way we can ensure that the `DAG` is correctly built up. The `root` of the `DAG` is the obo entry which has the same name as the parameter *"-root"*.

Based on the provided *"-mappingtype"*, the *JAR* has two different methods for parsing the file specified in *"-mapping"*. This part populates the created `DAG` object with the gene symbols. The symbols are stored as a `HashSet<String> geneSymbols` in each `GOEntry` object. Lastly, the *JAR* reads in the enrichment analysis file and stores the results in a `HashMap<String, Gene> enrichedGeneMap`. The first lines of our enrichment file also contains "Standard of Truth" (SoT) GO ids, which are gene sets that are known to be enriched for the given list of differentially expressed genes. These SoT GO ids are used to mark the corresponding `GOEntry` objects in the `DAG` and are additionally stored in a `HashSet<GOEntry> trueGoEntries`. We will use this for later comparisons.

(B) Preparing DAG for Enrichment Analysis:

At this point of the program, we have a fully constructed `DAG` object. As described in 1, a parent `GOEntry` object inherits all genes from its children. Calling the method `inheritGeneSymbols()` on the `root` of the `DAG` leads to a recursive update of the gene symbols in each `GOEntry` object from the bottom to the top of the `DAG`. After this step, the `root` contains all gene symbols of the `DAG`. The same logic is used to create a `HashSet<String> reachableGOIDs` in each `GOEntry` object, which will be useful for later calculations. The result file should also contain a field called *"shortest_path_to_a_true"*. This field is supposed to describe the shortest path from a

`GOEntry` A to the closest true `GOEntry` B. These paths can be pre-computed, avoiding expensive breadth-first searches which might compute the same path multiple times. For this purpose, each `GOEntry` object has a field `HashMap<GOEntry, LinkerClass> HighwayMap` which stores the true `GOEntries` **as keys** and a `LinkerClass` object as value. The `LinkerClass` object contains the `GOEntry` object **which has to be traversed** to reach the true `GOEntry` and a value k which describes the remaining distance to the true `GOEntry`.

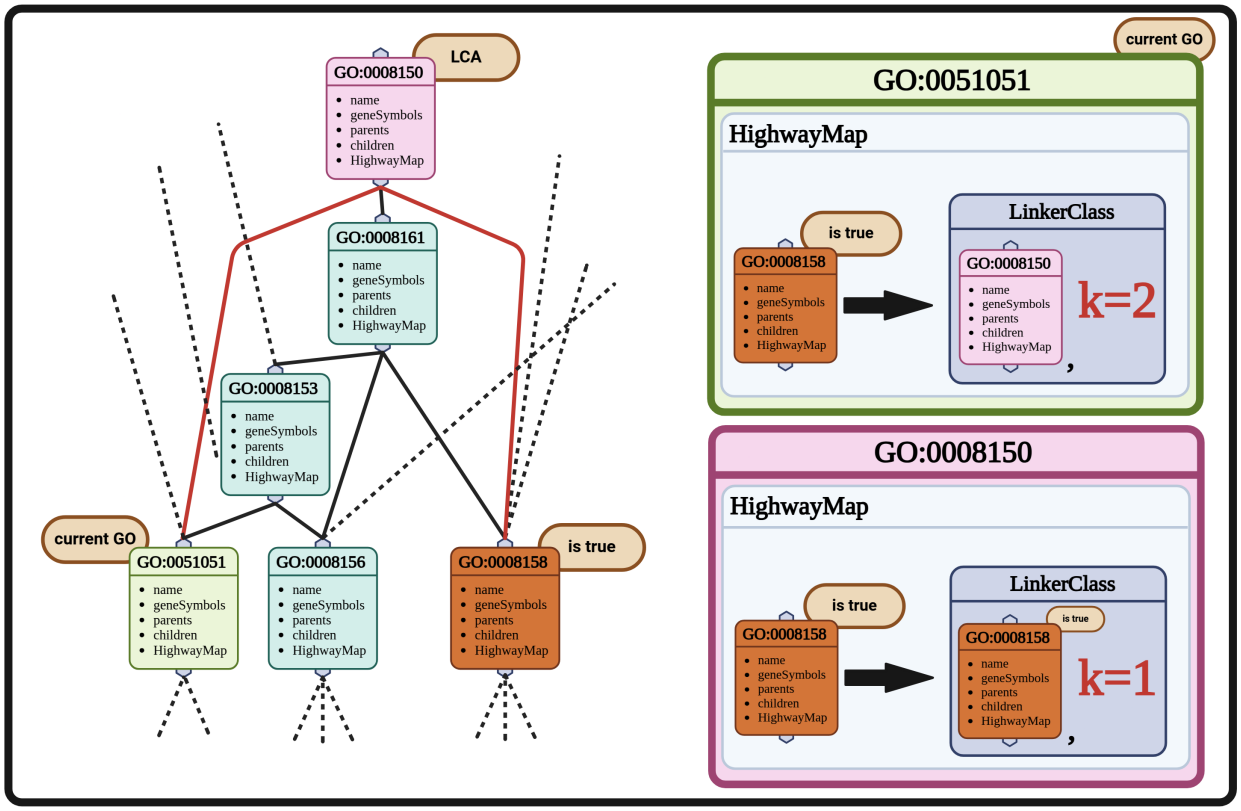

Figure 1 — A simple example of how the `HighwayMap` uses the true `GOEntries` as key and stores a `LinkerClass` object as value, serving as a highway to the true `GOEntry`. In this case there's only one `GOEntry` marked as *true*, that's why the `HighwayMap` only contains one element per `GOEntry`. The figure was created using BioRender 2024.

We start populating the `HighwayMap` with the true `GOEntries` by fist calling the method `signalShortestPathUp(0, GOEntry trueGO)` (Algorithm 1) on each true `GOEntry`. This method is called recursively on each parent of the true `GOEntry` and propagates the shortest path to the true `GOEntry` upwards in the `DAG`. This alone is insufficient since we also have to propagate the shortest path downwards in the `DAG`. For this purpose, we call the method `propagateShortestPaths(trueGo)` (Algorithm 2) on the **`root`** of the `DAG`. While the method

`propagateShortestPaths()` ensures that the shortest paths propagate down from the `root`, it does not explicitly finalize the correct shortest paths for every node. This is done by calling the method `signalShortestPathDown(0, trueGo)` (Algorithm 3) on all true `GOEntries`. It makes sure that once the shortest paths are propagated down from the `root`, every node (especially those closer to `trueGo`) has the correct, finalized shortest path to the trueGo.

Algorithm 1: signalShortestPathUp(k, trueGo)

Input: k, trueGo
Output: Updated shortest path values for all parents

```
1 foreach parent in parents do
2     if parent.highwayMap does not contain trueGo then
3         link = new LinkerClass(this, k + 1) ;
4         parent.highwayMap.put(trueGo, link) ;
5     else if parent.highwayMap.get(trueGo).getK() > k + 1 then
6         parent.highwayMap.get(trueGo).setGo(this) ;
7         parent.highwayMap.get(trueGo).setK(k + 1) ;
8     parent.signalShortestPathUp(k + 1, trueGo) ;
```

Algorithm 2: propagateShortestPaths(trueGo)

```
Input: trueGo
Output: Updated shortest path values for all descendants
currentPath ← highwayMap.get(trueGo) ;
if currentPath is not null then
    nextNode ← currentPath.getGo() ;
    if nextNode.highwayMap contains trueGo then
        pathThroughLinker ← nextNode.highwayMap.get(trueGo).getK() + 1 ;
        if pathThroughLinker < currentPath.getK() then
            currentPath.setK(pathThroughLinker) ;
currentK ← currentPath is not null ? currentPath.getK() : Integer.MAX_VALUE ;
foreach child in children do
    childPath ← child.highwayMap.get(trueGo) ;
    childK ← childPath is not null ? childPath.getK() : Integer.MAX_VALUE ;
    if currentK is not Integer.MAX_VALUE and currentK + 1 < childK then
        if childPath is null then
            childPath ← new LinkerClass(this, currentK + 1) ;
            child.highwayMap.put(trueGo, childPath) ;
        else
            childPath.setGo(this) ;
            childPath.setK(currentK + 1) ;
    child.propagateShortestPaths(trueGo) ;
```

Algorithm 3: signalShortestPathDown(k, trueGo)

```
Input: k, trueGo
Output: Updated shortest path values for all descendants
k ← k + 1 ;
foreach child in children do
    if child.highwayMap does not contain trueGo then
        link ← new LinkerClass(this, k) ;
        child.highwayMap.put(trueGo, link) ;
    else if child.highwayMap.get(trueGo).getK() > k then
        child.highwayMap.get(trueGo).setGo(this) ;
        child.highwayMap.get(trueGo).setK(k) ;
    child.signalShortestPathDown(k, trueGo) ;
```

(C) Enrichment Analysis:

The actual enrichment analysis is conducted in the method `analyzeParallel()` of the class `EnrichmentAnalysis`. This method iterates over all `GOEntry` objects and checks whether their amount of genes is within the specified range of *"-minsize"* and *"-maxsize"*. An object called `AnalysisEntry` is created for each `GOEntry` object which fulfills the requirements. It then calculates the overlap of the gene symbols of the current `GOEntry` with the gene symbols of the differentially expressed genes and stores the result in the *"size"* field of the `AnalysisEntry` object ($\equiv n$). The attribute *"noverlap"* is the amount of significantly differentially expressed genes which also occur in the current `GOEntry` ($\equiv k$). The universe size is the intersection of all gene symbols of the differentially expressed genes and all gene symbols of the `DAG` ($\equiv N$). Lastly, K is the amount of significantly differentially expressed genes intersected with all gene symbols of the `DAG`. For the Kolmogorow-Smirnow-Test we define the following values:

I. *In-Set Distribution*: Genes present in the current *GOEntry* and differentially expressed genes.

II. *Background Distribution*: Genes not in the *In-Set* but part of the background gene pool.

For both of these distributions, we store the *logFoldChange* values of the differentially expressed genes into two arrays. With these values, the statistical tests described in 2.1 are conducted to determine the significance of the enrichment of the current `GOEntry`. As a result, we obtain the following values and store them in the `AnalysisEntry` object:

- *hgPval*: Hypergeometric p-value.
- *fejPval*: Jackknife Fisher's Exact Test p-value.
- *ksPval*: Kolmogorov-Smirnov p-value.
- *ksStat*: Kolmogorov-Smirnov statistic ($\equiv$ *Enrichment Score*).

Now we calculate the shortest path to the closest true `GOEntry` for each `AnalysisEntry` by utilizing the pre-computed `HighwayMap` of the current `GOEntry`. We simply choose the `GOEntry` object which points to the `LinkerClass` with the smallest k value in the `HighwayMap` as our `target GOEntry`. For each visited `GOEntry` object on the path, we check if the `HighwayMap` still contains the `target GOEntry` as key. If so, we follow its `LinkerClass` object until we reach the `target`. We know when we hit the *least common ancestor (LCA)* of the current `GOEntry` and the

closest true `GOEntry`, by checking if the `reachableGOIDs` object contains our `target GOEntry` for each visited `GOEntry` object on the current path.

After all `AnalysisEntry` objects have been processed, all p-values are corrected for multiple testing using the *Benjamini-Hochberg* method. The corrected p-values are stored in the `AnalysisEntry` objects , and each object is written to the output file. The above procedure is conducted in parallel in order to speed up the process.

(D) GO Features:

The optional parameter *"-overlapout"* specifies an additional output file that stores further features of the underlying `DAG`. For this, each unique pair of `GOEntries` (A, B) fulfilling the *"-minsize"* and *"-maxsize"* parameter are intersected by their *gene symbols*. If there is an overlap, we calculate the shortest path between A and B. In this case, we use a *Breadth-First Search* algorithm to explore all ancestors of A and B and return the shortest path. Additionally, we write a `boolean` value into the output file, indicating whether A can be directly reached from B or vice versa. Lastly, we store the max percentage of shared genes between A and B:

$$\textit{max percentage} = \frac{|A \cap B|}{\textit{max}\{|A|, |B|\}} \times 100.$$

3 — Results

3.1. Runtime

The runtime of the *JAR* was timed using a `Logger Class`, which timed certain analysis steps. The two main analysis methods are implemented as single-threaded and multi-threaded and will be compared in this section. We used the provided *obo* file, *mapping* file and *enrichment* file for the analysis and ran the *JAR* with *"-minsize" 50* and *"-maxsize" 500*.

Figure 2 shows the total runtime of the *JAR* and also splits it into several sub-tasks like parsing the input files, preparing the `DAG` for the enrichment analysis itself. Running the *JAR* with *"-mappingtype ensembl"* seems to be faster compared to the *"go"* mapping. This holds for almost every category, except for "optimizing DAG", which takes 0.01s longer, but this is most likely a hardware inconsistency. The *"ensembl"* mapping file has fewer genes mapped to `GOEntries` than the *"go"* mapping file, and because the runtime of every step is highly dependent on the number

of genes inside a *gene set*, having fewer genes in total also leads to a faster runtime.

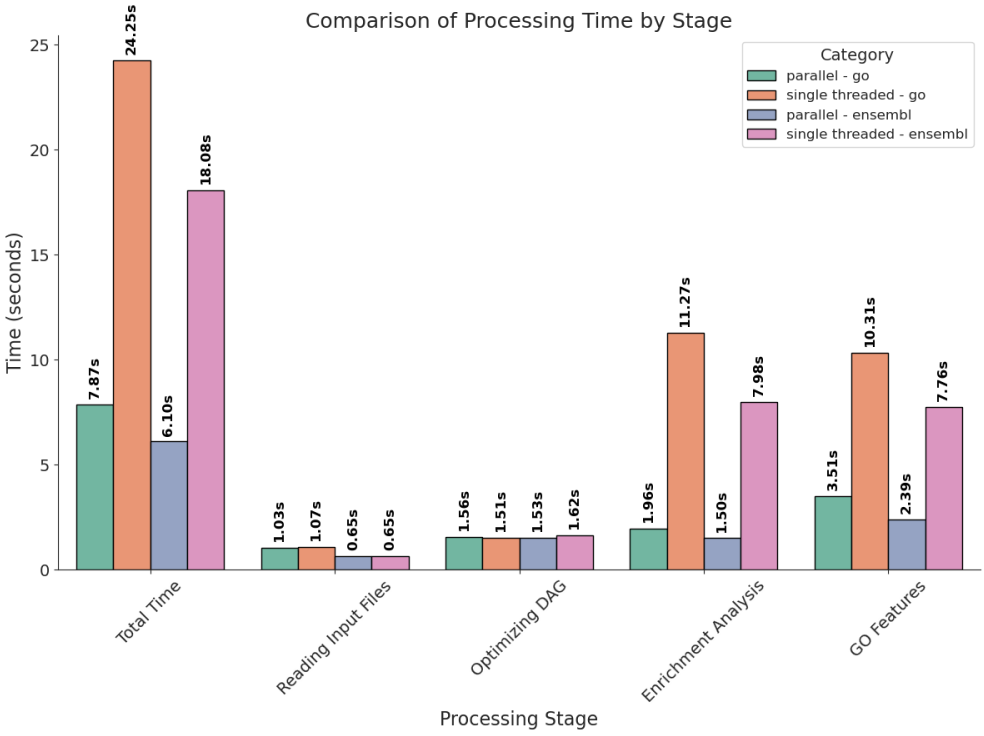

Figure 2 — Comparison the time duration of key setps of the program based on *"-mappingtype"* and implementation variant. Note that only the last two steps, "Enrichment Analysis" and "GO Features", were parallellized.

The two analysis steps "Enrichment Analysis" (**step (C)**) and "GO Features" (**step (D)**) seem to scale ruffly proportional between *"-mappingtype"* and implementation type. An interesting aspect is that parallelizing step (D) seems to save less time compared to the parallelization of step (C). This is because of the large amount of set operations required during step (C). For the parameters used in the evaluation, a total of $\frac{n(n-1)}{2}$ many unique pairs have to be compared by intersecting their gene symbols. In this case n is the number of `GOEntries` of the `DAG` with a gene set size $|g|$ of $50 \leq |g| \leq 500$. This results in 3 378 224 pairs with the *"go mappingtype"* and 2 483 776 pairs with the *"ensembl mappingtype"*, whereas in step (C), only 1838 ("go") or 1576 ("ensembl") many `GOEntries` are analyzed.

Table 1 — DAG Properties summary for "-minsize 50" and "-maxsize 500". The table shows the number of genes, gene sets, leaves, the length of the shortest S and longest L path to the `root` in the DAG and the amount of analyzed `GOEntries` which agree with the "-minsize" and "-maxsize" parameters.

Mapping Type	#Genes	`#GOEntries`	#GOs analyzed	#Leafs	S	L
GO	17026	29385	1838	14991	GO:0031629 $\rightarrow$ 2	GO:1905741 $\rightarrow$ 17
ENSEMBL	15496	29385	1576	14991	GO:0031629 $\rightarrow$ 2	GO:1905741 $\rightarrow$ 17

Table 1 shows the properties of the `DAG` for both mapping types. The `DAG` itself contains 29385 gene sets, with 14991 leaves. The shortest path to the `root` is 2 edges long, while the longest path is 17 edges long. These properties are the same for both mapping types since the `DAG` structure remains the same for both "go" and "ensembl". The only difference is the gene symbols which are stored in the `GOEntries` (15496 vs. 17026) and the amount of `GOEntries` analyzed. Figure 3 additionally shows the distribution of gene set sizes in the `DAG` for both mapping types, as well as the difference in gene set sizes between parent and child nodes. It seems that the difference in gene set size per traversed edge is slightly more pronounced for the `GO` mapping type compared to the `"ensembl"` mapping type, while the overall distribution of gene set sizes is similar for both mapping types.

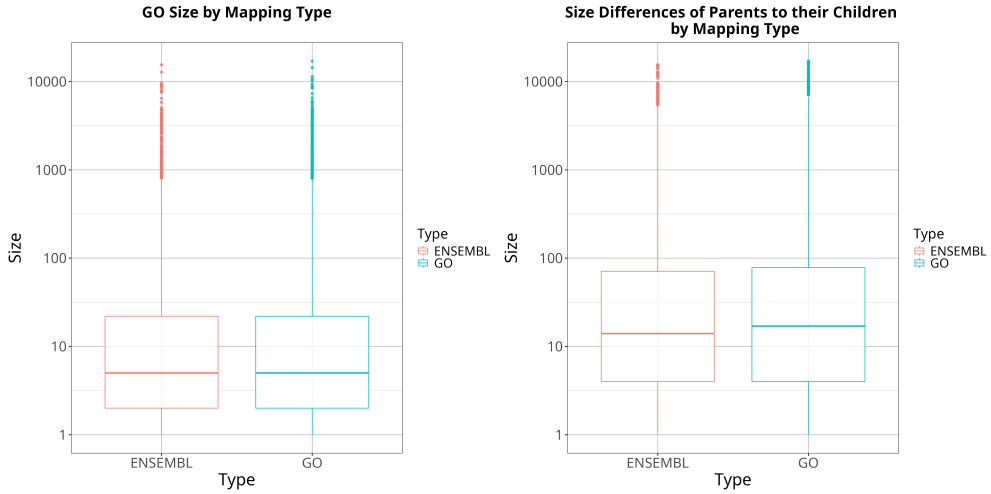

Figure 3 — Comparison of gene set sizes in the DAG as well as the difference in gene set sizes between parent and child nodes, per mapping type.

3.3. *Enrichment Results*

Before we analyzed the results of the enrichment analysis, we filtered the 19 *SoT* gene sets by their size, leaving us with 13 *SoT's* which could be discovered by our *JAR*.

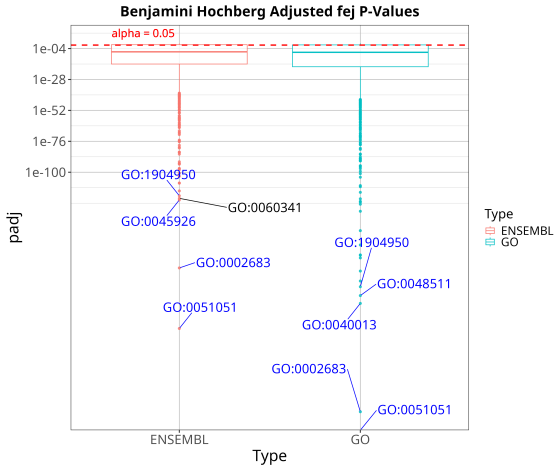

Figure 4 — Distribution of BH adjusted (Jackknife) Fisher Exact Test p-values.

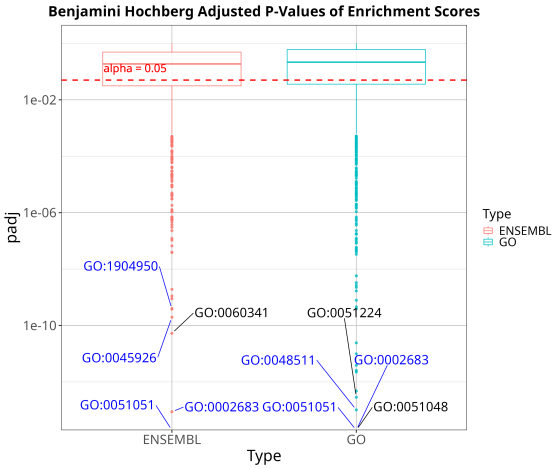

Figure 5 — Distribution of BH adjusted Kolmogorov-Smirnov p-values.

Figure 4 and 5 show the distribution of the adjusted p-values, where the top 5 most significant `GOEntries` are labeled for each "-mappingtype". SoT gene sets are colored in blue. Right away, a major discrepancy is that there are way more significant results for the *ORA* (Figure 4) approach compared to the *GSEA* approach in Figure 5. The α threshold of 0.05 is leveling with the $Q3$ of the boxplot, meaning that the majority of the significant results are below the α threshold, even after multiple testing correction.

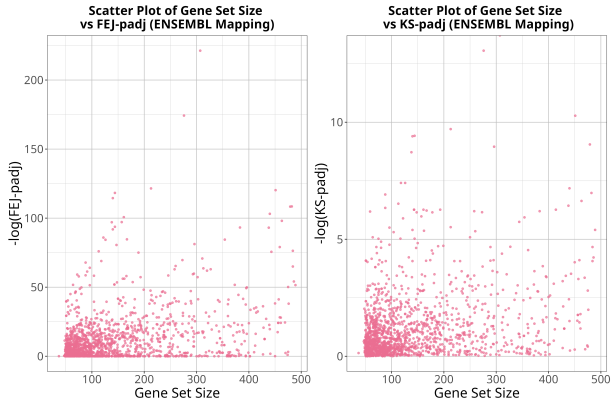

Figure 6 — Ensembl Mapping Scatter Plot: significance vs gene set size.

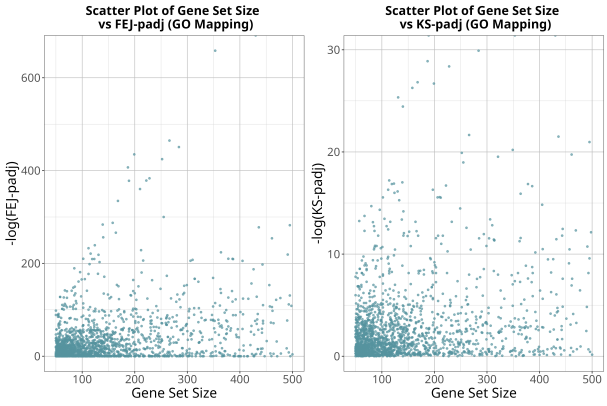

Figure 7 — GO Mapping Scatter Plot: significance vs gene set size.

We can also see how smaller gene sets are more often considered significant for both *"-mappingtype" options*, as shown in Figure 6 and 7. The *"go"* mapping type has a higher amount of significant gene sets compared to the *"ensembl"* mapping type because the overall amount of genes in the `DAG` is higher (see Table 1).

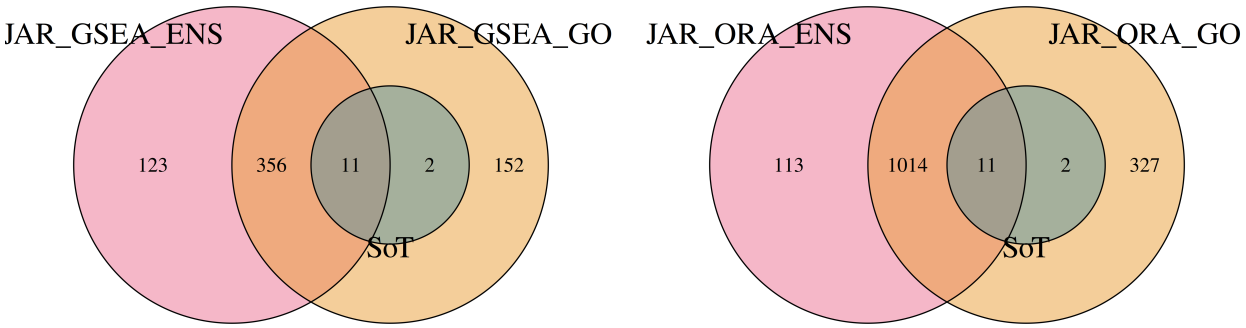

Figure 8 — Comparison of the results of the *GSEA* and *ORA* approach based on the *"-mappingtype"*. Only `GOEntries` with a padj below 0.05 are considered. In this case, ORA uses the adjusted Jackknife Fisher Exact Test (fej-pval), and GSEA uses the adjusted Kolmogorov-Smirnov p-value.

Again, we can see how the *GSEA* approach seems to be more conservative compared to the *ORA* approach, since there are less significant results for the *GSEA* approach. Both approaches identify almost all SoT gene sets, which is a good sign. For *ORA* and *GSEA*, two of the SoT gene sets are not found, which are *"GO:0098754 "* and *"GO:2000242"*. These gene sets are non-existent in the *"ensembl mapppingtype"* and are therefore not discovered.

```
→ grep -e "GO:2000242|GO:0098754" goa_human_ensembl.tsv | wc -l
0
```

Figure 9 shows how, in both "-mappingtype" cases, almost all significant `GOEntries` discovered by *GSEA*, are also discovered by *ORA*. For the *"ensembl"* mapping, *GSEA* finds 6 entries not covered by *ORA*. For the "go" mapping, this number is reduced to 3. This means that in our case, *GSEA*

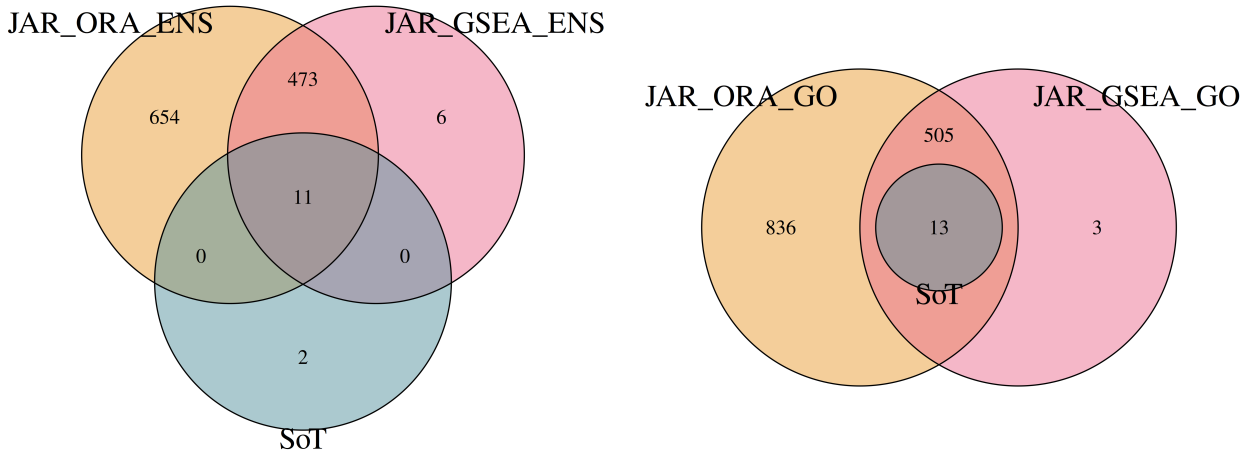

Figure 9 — Comparison of overlap of significant `GOEntries` of same *"-mappingtype"* between ORA and GSEA.

and *ORA* are mostly aligned with each other, with the exception of *GSEA* having less *FPs* than *ORA*. The actual amount of significant results of *GSEA* is likely even less than what is shown in Figure 9, since, as mentioned in Section 2.1.2, the *enrichment p-vals* are not properly evaluated through *phenotype permutation* (Subramanian et al. 2005).

3.4. Comparison with Online Tool gProfiler (ORA)

For the comparison with *gProfiler*, we uploaded the list of all *significant* genes of the *enrichment file* to the web application. *gProfiler* performs *ORA*, using known gene sets and pathways from several different databases. We filtered for gene sets complying with out *"-minsize 50"* and *"-maxsize 500"* parameters (see Figure 10) and only looked at the *BP Ontology*.

Figure 10 — Filtering *gProfiler* results to match *"-minsize"* and *"-maxsize"* parameters.

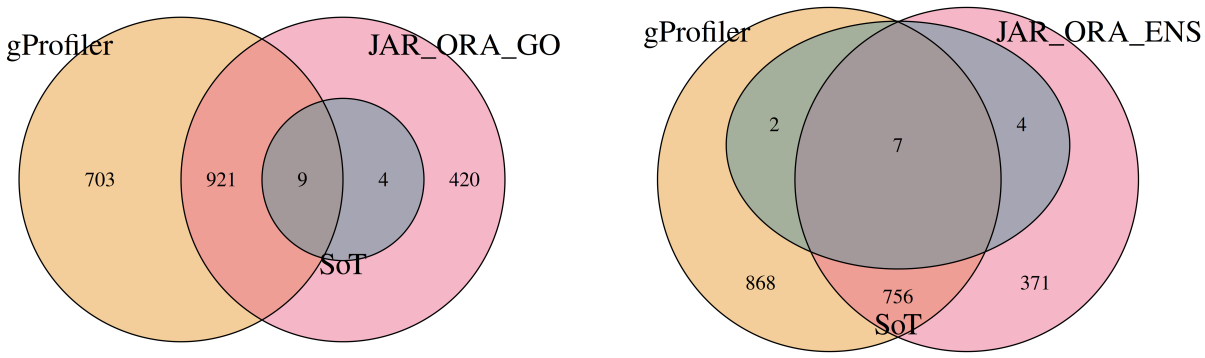

Figure 11 — Comparing significantly enriched gene sets discovered by gProfiler and our JAR, based on "-mappingtype".

We can see how in Figure 11, *gProfiler* identifies even more significant gene sets than our *JAR*. At the same time, the external tool seems to struggle with 4 `GOEntries` in both cases which could be due to a similar issue where GO IDs of the *SoT* list are not present in the `GOEntries` *gProfiler* uses for its analysis.

3.5. Comparison with R library fgsea (GSEA)

We ran the library *fgsea* on our data in order to compare the *GSEA* analysis of our *JAR* with a state-of-the-art implementation. The results are shown in Figure 12 and show that, as expected, *fgsea* finds less significant gene sets compared to our *JAR*. This is due to the fact that *fgsea* computes the empirical p-value for the *ES* (Sergushichev 2016) and therefore avoids too many FPs. As gene sets we used the library *msigdb* (Bhuva, Smyth, and Garnham 2024) with the following specifications:

```
# read provided enrichment file
genes <- fread("genes.tsv")
#      id      fc signif
#           <char>   <num> <lgcl>
# 1: DNAJC25-GNG10 -1.3420  FALSE
# 2:      IGKV2-28 -2.3961  FALSE
# 3:      ...      ...      ...

# order genes by‌fc
genes <- genes[order(genes$fc, decreasing = TRUE),]
geneList <- genes$fc
names(geneList) <- genes$id

#‌oad gene sets from database
gene_sets <- msigdbr(species = "Homo sapiens", category = "C5")
gene_sets  <- as.data.table(gene_sets)
pathways <- split(gene_sets$gene_symbol, gene_sets$gs_name)
# run fgsea with min, max size == 50, 500
fgsea_results <- fgsea(pathways = pathways,
                       stats = geneList,
                       minSize = 50, maxSize = 500)
```

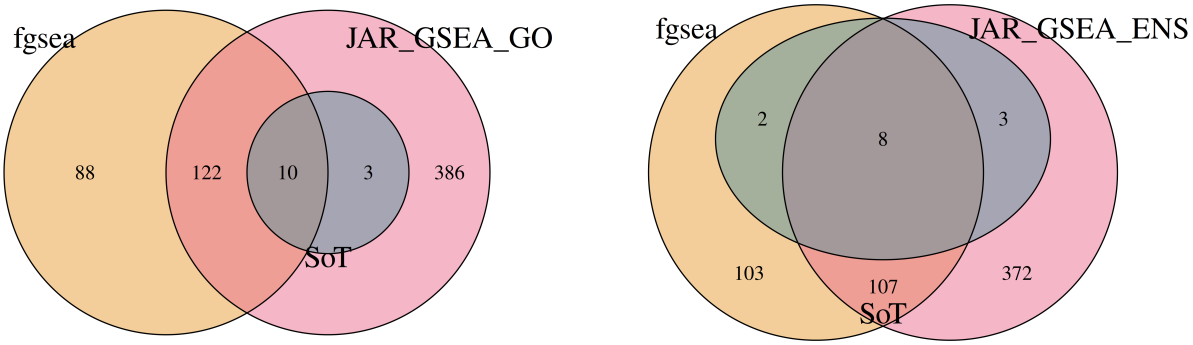

Figure 12 — Comparing significantly enriched gene sets discovered by fgsea and our JAR, based on "-mappingtype".

In Figure 12, it also seems as if the three `GOEntries` not discovered by *fgsea* have likely been replaced or are simply non-existent in the *msigdb C5* gene set database. For both *"-mappingtypes"*, our *JAR* is able to at least identify more than half of the gene sets discovered by *fgsea*.

References

Ashburner, Michael, Catherine A. Ball, Judith A. Blake, David Botstein, Heather Butler, J. Michael Cherry, Allan P. Davis, et al. 2000. "Gene Ontology: tool for the unification of biology." *Nature Genetics* 25, no. 1 (May): 25–29. ISSN: 1546-1718. https://doi.org/10.1038/75556. https://doi.org/10.1038/75556. (Cited on page 3).

Bhuva, Devika, Gordon Smyth, and Alice Garnham. 2024. *msigdb: An ExperimentHub Package for the Molecular Signatures Database (MSigDB).* R package version 1.14.0. https://davislaboratory.github.io/msigdb. (Cited on page 19).

BioRender. 2024. *BioRender - Biological Figure Creation Tool.* https://BioRender.com. Accessed: 2024-12-09. (Cited on page 8).

Huang, Da Wei, Brad T. Sherman, and Richard A. Lempicki. 2008. "Bioinformatics enrichment tools: paths toward the comprehensive functional analysis of large gene lists." *Nucleic Acids Research* 37, no. 1 (November): 1–13. ISSN: 0305-1048. https://doi.org/10.1093/nar/gkn923. eprint: https://academic.oup.com/nar/article-pdf/37/1/1/17059338/gkn923.pdf. https://doi.org/10.1093/nar/gkn923. (Cited on page 4).

Kolberg, Liis, Uku Raudvere, Ivan Kuzmin, Priit Adler, Jaak Vilo, and Hedi Peterson. 2023. "g:Profiler—interoperable web service for functional enrichment analysis and gene identifier mapping (2023 update)." *Nucleic Acids Research* 51, no. W1 (May): W207–W212. ISSN: 0305-1048. https://doi.org/10.1093/nar/gkad347. eprint: https://academic.oup.com/nar/article-pdf/51/W1/W207/50736882/gkad347.pdf. https://doi.org/10.1093/nar/gkad347. (Cited on page 3).

Sergushichev, Alexey A. 2016. "An algorithm for fast preranked gene set enrichment analysis using cumulative statistic calculation." *bioRxiv*, https://doi.org/10.1101/060012. eprint: https://www.biorxiv.org/content/early/2016/06/20/060012.full.pdf. https://www.biorxiv.org/content/early/2016/06/20/060012. (Cited on pages 3, 19).

Subramanian, Aravind, Pablo Tamayo, Vamsi K. Mootha, Sayan Mukherjee, Benjamin L. Ebert, Michael A. Gillette, Amanda Paulovich, et al. 2005. "Gene set enrichment analysis: A knowledge-based approach for interpreting genome-wide expression profiles." *Proceedings of the National Academy of Sciences* 102 (43): 15545–15550. https://doi.org/10.1073/pnas.0506580102. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.0506580102. https://www.pnas.org/doi/abs/10.1073/pnas.0506580102. (Cited on pages 5, 17).