Generating Splice Graphs based on TranscriptDb Objects

Martin Morgan Marc Carlson Daniel Bindreither

August 23, 2012

1 Introduction

The new `spliceGraph` function included into the *GenomicFeatures* package creates splicing graph [3] structures based on *TranscriptDb* objects. One use case for this function would be the analysis of an RNA-seq experiment. As an use case example we compare serious ovarian cancer (SOC) samples from patients with benign samples of healthy individuals to find evidence for differential expression of splice variants. Currently available R packages like *DEXSeq* offer the capability to identify differentially expressed exons. Since single exons often contain only a few nucleotides, counting the reads associated with this rather short sequences could be problematic, especially if exons overlap each other. This issues also influence steps in the analysis further downstream like the testing for differential expression. This vignette describes how to utilize the splicing graph structures to tackle some of those problems and compares the results of this novel approach to the well established methodology of differential exon expression analysis.

2 Splicing graphs

Alternative splicing is a complex biological process for modifying the primary RNA transcript leading to two or more transcript variants of a certain gene. These variants can often be plentiful, especially for large genes it is usually hard to describe the full complexity of the resulting variants in a formal, logic and short way. To capture the full variety of splice variants in one data structure Heber at al [2] introduced the term splicing graph and provided a formal frame work for representing different choices of the splicing machinery. A splicing graph in general is a directed acyclic graph

(DAG) and consists of two main structure elements designated as vertices and edges. Vertices represent sites on the transcript and edges, which connect the vertices, represent exons or introns of a certain transcript. Wether the edge is representing an exon or an intron is determined by the type of the flanking vertices. Vertices can be either acceptor splice sites, donor splice sites, transcript start sites, transcript ends or artificial sites required for the correctness of the splicing graph frame work. Those artificial sites are called root vertex and leaf vertex. The root vertex is always the first vertex of a splicing graph and therefore the origin. The leaf vertex is always the last vertex and can be seen as the sink of the graph . Root and leaf vertices never represent real sites on the transcripts coming from a certain gene locus.

Figure 1 shows the splice graph representation of the transcript variants of the gene JUNB (entrez gene id 3726). Before the splice graph is constructed the exons of the individual genes respectively transcripts get disjoint to avoid overlapping sequence parts. The green dashed lines in Figure 1 represent genomic locations where exons get disjoint during the splicing graph construction.

Each position where exons are disjoint can be considered as an intron flanked by two vertices, one representing the end of the first part of the disjoint exon and one representing the start of the second exon part. It is important to notice that such a position is always described by two vertices and a zero nucleotide long intron.

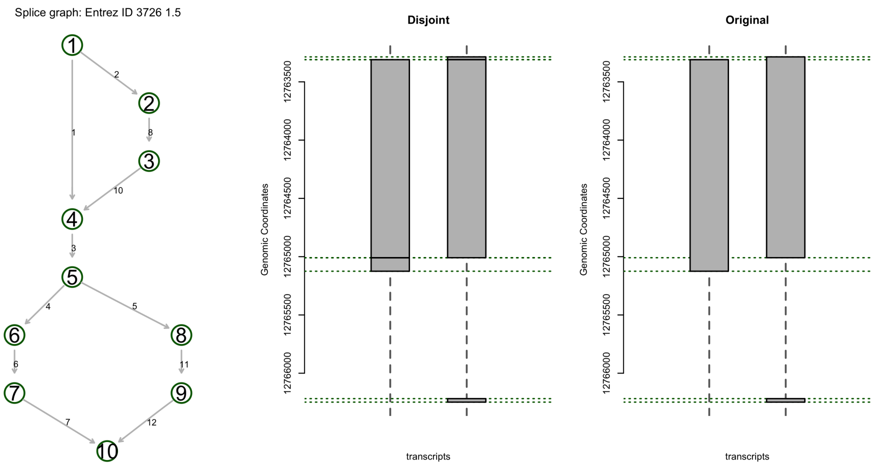

Figure 1: Splice graph representation of the two transcript variants of the gene 3726 (JUNB). Left: splice graph representation. Right: transcript representation

In some case vertices can represent two different types of sites, but the composition of types of this two type vertices is limited. Only splice site acceptors and transcript end sites or splice site donor and transcript start sites can occur as a pair of types within one single vertices.

Such a two type vertex occurs when the start of an exon, which is not the first exon of transcript one, falls together with the transcription start site of transcript two. The same thing happens when the end of an exon, which is not the last exon of transcript two, falls together with the transcript end of transcript one.

As mentioned above the edges can either represent introns or exons depending on the types of the flanking vertices. An edge between a splice site acceptor and a splice site donor vertex for example would represent an intron whereas the edge between an splice cite donor and an splice site acceptor would represent an exon. Exons are also described between transcription start sites and splice site acceptors or splice site donors and transcript ends. The edges between the root vertex and the transcription start sites as well as between the transcript ends and the leaf contain neither an intron nor an exon. Actually such edges contain no sequence information at all.

Since not all splice sites are alternative splice sites the initial splicing graph outlined above can be simplified by removing vertices which have in and out degree equal to one. Those vertices are non informative, because no alternative choices of the splicing machinery are recorded in the annotation. Edges associated with such non informative vertices get sequentially collapsed with the previous ones. The final outcome are edges associated with one or more exons which do not overlap each other. If on edge consists of several exons, those exons are put together and form a larger sequence region which provides advantages in counting reads. The `spliceGraph` function returns the collapsed edges with their associated disjoint exons.

The next section describes how such graphs are generated in R and utilized for differential expression analysis of RNA-seq data.

3 Creating the splicing graph

As outlined in the previous section the splicing graph approach extends the differential expression analysis of single exons. The starting point for creating a splice graph is a *TranscriptDb* object containing all required gene and transcript annotations. Here the *TxDb.Hsapiens.UCSC.hg18.knownGene* package is used, but if other annotations are needed such objects can easily be created by using functions of the *GenomicFeatures* package. First we load the selected *TranscriptDb* object and assign it to a shorter name.

```
> library("TxDb.Hsapiens.UCSC.hg18.knownGene")
> txdb <- TxDb.Hsapiens.UCSC.hg18.knownGene
```

Next we have to load the *GenomicFeatures* package implementing the `spliceGraph` function. Creating a splicing graph for the whole *TranscriptDb* object usually takes about 10 minutes on a normal desktop computer. To save some time we utilize only chromosome 19. To perform `spliceGraph` on a subset of the *TranscriptDb* object the *GenomicFeatures* package provides the `isActiveSeq` function to deactivate chromosomes in the *TranscriptDb* object which should not be used. Usually one would deactivate strange chromosome variants which sometimes can be found in the annotation.

```
> activeChr <- ! isActiveSeq(txdb)
> activeChr[names(activeChr) == "chr19"] <- TRUE
> isActiveSeq(txdb) <- activeChr
```

Finally we can execute the `spliceGraph` function onto the *TranscriptDb* object. The function will return a *GrangesList* object containing the collapsed edges of the splicing graph and their associated exons.

```
> library("GenomicFeatures")
> exsByEdges <- spliceGraph(txdb)
```

Now we can have a look onto the resulting object. We will see that the object is of class *GrangesList* and contains the edge IDs as names. The exons within the *Granges* object are not the original exons provided by the *TranscriptDb* object. As mentioned in the previous section during the splicing graph construction overlapping exons within a gene get disjoint. Disjoining produces new exons with new exon ids which differ in size compared to the original exons.

The original exon ids associated with the new exon ids can be found in the metadata column called exon ids. Each element of the column is a character list containing the original exon ids The new disjoint exon ids can be retrieved directly from the metadata column called disJ exon ids. Below the first edge of the object returned by the `spliceGraph` function is shown

```
> exsByEdges[1]

GRangesList of length 1:
$1
GRanges with 1 range and 2 elementMetadata cols:
      seqnames               ranges strand | disJ_exon_id
         <Rle>            <IRanges>  <Rle> |    <integer>
  [1]    chr19 [63556582, 63556677]      - |           10
                      exon_ids
      <CompressedCharacterList>
  [1]                   234547

---
seqlengths:
    chr19
 63811651
```

To get a mapping of the original exons to the new disjoint exons the exons by edge object have to be unlisted first. From the flat *GrangesList* object we can then easily retrieve the original exon ids from the exon ids column. For the mapping we utilize a simple *data.frame* object because the mapping between the new disjoint exon ids and the original exon ids is ambiguous.

```
> exsByEdges.flat <- unlist(exsByEdges, use.names=FALSE)
> orig.Ex <- values(exsByEdges.flat)[["exon_ids"]]
> disJ.Ex <- values(exsByEdges.flat)[["disJ_exon_id"]]
> newExNames <- rep(disJ.Ex, elementLengths(orig.Ex))
> origExToNewEx <- data.frame(orig.Ex=unlist(orig.Ex),
+                             disJ.Ex=newExNames,
+                             row.names=NULL)
> head(origExToNewEx)
```

```
  orig.Ex disJ.Ex
1  234547      10
2  235086    1955
3  235681    1964
4  235092    1963
5  236844    1984
6  236843    1983
```

The gene ids are directly retrievable from the *GrangesList* object returned by the `spliceGraph` function. To get the gene ids the `values` accessor function is utilized. The mapping from edges to genes is used further downstream in the analysis.

```
> gnIDs <- values(exsByEdges)[["gene_id"]]
> edgeIDs <- names(exsByEdges)
> gnToEdge <- data.frame(gnIDs, edgeIDs, row.names=edgeIDs)
> head(gnToEdge)
```

```
       gnIDs edgeIDs
1          1       1
10049 126549   10049
10066 126567   10066
10072   1311   10072
10073   1311   10073
10076   1311   10076
```

4 Counting reads

This section deals with counting the reads associated with the individual edges. To elucidate later in this vignette some advantages of the splice graph approach the reads for the original exon structure are also counted. The reason for obtaining both count tables is that the edges approach will be later compared to the classic exon approach.

Counting can be done in different ways. Here we use the summarizeOverlaps function of the *GenomicRanges* package with default settings.

As a starting point for counting the aligned reads are required. The aligned reads used in this example are stored as BAM files. The BAM files used for counting reads are not provided since they would exceed the usual size of an R package. The code below is therefore only an example code which should not be executed since the BAM files are lacking.

First the location of the BAM files used in here is specified . The BAM files one to three in this directory are the reads of the benign samples and the BAM files 62 to 64 contain the the selected SOC samples. The summarizeOverlaps function requires a *BamFileList* object which is created out of the the file paths to the six selected BAM files.

```
> if(reCnt) {
+   require("Rsamtools")
+   bamPath <- "/shared/labs/EDI/users/mfitzgib/Solexa"
+   fls <- sub(".bai$", "",
+              list.files(bamPath, recursive=TRUE,
+                         pattern="accepted_hits.*bai$",
+                         full=TRUE))
+   fls <- fls[c(1:3, 62:64)]
+   bfs <- BamFileList(fls)
+   names(bfs) <-
+     gsub("/shared/labs/EDI/users/mfitzgib/Solexa/tophat_",
+          "",fls)
+ }
```

4.1 Counting reads per edges

The reads get counted based on the edge model created above. The summarizeOverlaps function reduces first adjacent elements of an edge to one larger range before the counting is done.

Counting is performed for each BAM file separately and can be parallelized to save computation time. After counting the count table contains one row for each edge.

```
> if(reCnt) {
+   library(parallel)
+   resEx.byEdge <-
+     summarizeOverlaps(features = exsByEdges, reads = bfs,
+                         mc.cores = getOption("mc.cores", 3L))
+
+   cD.exByEdge <- assays(resEx.byEdge)$counts
+   colnames(cD.exByEdge) <-
+     sub("/accepted_hits.bam", "", colnames(cD.exByEdge))
+   save(cD.exByEdge, file="cD.exByEdge-SG-Vig.Rda")
+ } else {
+   fn <- system.file("extdata", "cD.exByEdge-SG-Vig.Rda",
+                       package="GenomicFeatures")
+   load(fn)
+ }
```

4.2 Counting reads per exons

Since the properties of the splicing graph structure should be compared to the properties of the exon structure, reads get also counted for the exon model. To ensure fairness all exons used in the edge model should be also present in the exon model. The exon model can be easily retrieved from the *TranscriptDb* object and subset to the original exons used for creating the edge model. Duplicated exons get removed, because they would influence the exon counting process.

```
> exsByGenes <- exonsBy(txdb, "gene")
> gnNames <- rep(names(exsByGenes), elementLengths(exsByGenes))
> exsByGenes.flat <- unlist(exsByGenes, use.names=FALSE)
> # remove duplicates
> notDupl <- !duplicated(values(exsByGenes.flat)[["exon_id"]])
> exsByGenes.flat <- exsByGenes.flat[notDupl]
> names(exsByGenes.flat) <- values(exsByGenes.flat)[["exon_id"]]
> gnNames <- gnNames[notDupl]
> names(gnNames) <- names(exsByGenes.flat)
```

For performing the comparison edge versus exon model a exons to genes map is required. Such a map was already created above called `gnNames`. Now we are ready for counting the reads of the exon model. Counting is performed in the same way as for the edge model. Last we create a exon count table.

```
> if(reCnt) {
+   res.exsByGenes <-
+     summarizeOverlaps(features = exsByGenes.flat, reads = bfs,
+                          mc.cores = getOption("mc.cores", 3L))
+
+   cD.exsByGenes <- assays(res.exsByGenes)$counts
+   colnames(cD.exsByGenes) <-
+     sub("/accepted_hits.bam", "", colnames(cD.exsByGenes))
+   save(cD.exsByGenes, file="cD.exsByGenes-SG-Vig.Rda")
+ } else {
+   fn <- system.file("extdata", "cD.exsByGenes-SG-Vig.Rda",
+                     package="GenomicFeatures")
+   load(fn)
+ }
```

The final result of this section are the two count tables which contain the read counts per exons and the read counts per edges.

5 Testing for differentially expressed edges

This section deals with testing for differential expression of the individual edges and exons. In general genes can consist of one edge or of more than one edge. The same is true for the exons, because there might be also some short genes containing only one single exon. This facts are crucial for performing differential expression analysis in the right way. The most obvious strategy is to use the *DEXSeq* package for genes with multiple exons or edges and the *DESeq* [1] package for genes with only one edge or exon. We would expect much more single edge genes than single exon genes because edges can consist of several exons.

Chromosome 19 has 6885 edges and 14262 exons. Now as an initial part of the comparison of the exons versus the edges approach we want to find out how many single exon genes and single edge genes we actually have.

```
> edIds <- names(exsByEdges)
> gnIds.edges <- gnToEdge[rownames(cD.exByEdge),]$gnIDs
> nrOfEdgesPerGns <-
+   elementLengths(split(edIds, factor(gnIds.edges)))
> sglEdgeGns <- nrOfEdgesPerGns == 1
> exIds <- names(exsByGenes.flat)
> gnIds.exs <- gnNames[names(exsByGenes.flat)]
> nrOfExsPerGns <-
+   elementLengths(split(exIds, factor(gnIds.exs)))
> sglExGns <- nrOfExsPerGns == 1
```

We have 70 single exon genes and 426 single edge genes. Single element genes get tested for differential expression by utilizing the *DESeq* package. For the other genes with multiple edges and multiple exons *DEXSeq* is utilized.

```
> cD.exByEdge <- cD.exByEdge[, colnames(cD.exsByGenes)]
```

5.1 DEXSeq analysis for multi element genes

The initial step for each differential expression analysis is to set up the design of the experiment. Basically we are interested in comparing tumor samples with benign samples. This means we have a three against three design and therefore for each condition three biological replicates.

```
> benign <- grepl("Benign",colnames(cD.exByEdge ) )
> design <- factor(ifelse(benign, "Benign", "SOC"),
+                  levels=c("Benign", "SOC"))
> names(design) <- ifelse(benign, "Benign", "SOC")
> design

Benign Benign Benign    SOC    SOC    SOC
Benign Benign Benign    SOC    SOC    SOC
Levels: Benign SOC
```

Below exon and edge count data sets required for the *DEXSeq* analysis are created. For details about *DEXSeq* see the *DEXSeq* package vignette. Before the element count data sets are finally created we have to prepare annotation information which is in principle not needed for performing the analysis itself, but is essential to create nice plots of top candidate genes afterwards.

The code below extracts the start and end coordinates of each edge and orders the edges according to their genomic starting position.

```
> exsByEdges.flat <- unlist(exsByEdges, use.names=FALSE)
> edgeIDs <- rep(names(exsByEdges), elementLengths(exsByEdges))
> rle <- strand(exsByEdges.flat)
> start <- start(exsByEdges.flat)
> end <- end(exsByEdges.flat)
> temp <- data.frame(start, end, edgeIDs,
+                    strand=rep(rle@values, rle@lengths),
+                    chr=rep(seqnames(exsByEdges.flat)@values,
+                          seqnames(exsByEdges.flat)@lengths))
> temp <- temp[order(temp$edgeIDs, -temp$end), ]
> end <- temp$end[!duplicated(temp$edgeIDs)]
> names(end) <- temp$edgeIDs[!duplicated(temp$edgeIDs)]
> temp <- temp[order(temp$edgeIDs, temp$start), ]
> T <- temp[!duplicated(temp$edgeIDs),]
> rownames(T) <- T$edgeIDs
> T$end <- end[rownames(T)]
> T <- T[order(gnToEdge[T$edgeIDs,]$gnIDs, T$start), ]
```

T contains now all required edge annotation information and we can create a `newExonCountSet`. The ordering of the edges in the annotation information *data.frame* should be the same as in the count table. After creating the *ExonCountSet* we subset to genes containing multiple edges.

```
> library(DEXSeq)
> eData.Edge <-
+   newExonCountSet(countData = cD.exByEdge[T$edgeIDs,] ,
+                   design = design,
+                   geneIDs = gnToEdge[T$edgeIDs,]$gnIDs,
+                   exonIDs = T$edgeIDs,
+                   exonIntervals = T)
> eData.Edge <-
+   eData.Edge[! gnToEdge[T$edgeIDs,]$gnIDs %in%
+              names(sglEdgeGns[sglEdgeGns]), ]
```

The same procedure performed for the edges is also done for the exons. The code below extracts the start and end coordinates of each exon.

```
> T <- data.frame(chr=rep(seqnames(exsByGenes.flat)@values,
+                   seqnames(exsByGenes.flat)@lengths),
+                 start=start(exsByGenes.flat),
+                 end=end(exsByGenes.flat),
+                 strand=rep(strand(exsByGenes.flat)@values,
+                   strand(exsByGenes.flat)@lengths),
+                 row.names=values(exsByGenes.flat)[["exon_id"]])
```

T contains now all required exon annotation information and we can create a `newExonCountSet`. The ordering of the exons in the annotation information *data.frame* should be the same as in the count table. After creating the *ExonCountSet* we subset to genes containing multiple exons.

```
> eData.Ex <-
+   newExonCountSet(countData = cD.exsByGenes,
+                   design = design,
+                   geneIDs = gnNames[rownames(cD.exsByGenes)],
+                   exonIDs = rownames(cD.exsByGenes),
+                   exonIntervals=T[rownames(cD.exsByGenes),])
> eData.Ex <-
+   eData.Ex[! gnNames[rownames(cD.exsByGenes)] %in%
+             names(sglExGns[sglExGns]), ]
```

Next we compute the size factors of the individual libraries, estimate the dispersion and fit the dispersion function for both count data sets. Then we test for differential expression and estimate the fold changes. Last we create a top table for both sets. Keep in mind that *DEXSeq* discards by default row elements in the count tables which have less than 11 counts across all samples. *DEXSeq* also doesn't test genes which have more than 70 exons or edges.

Run the analysis for the exon count data set.

```
> eData.Ex <- estimateSizeFactors(eData.Ex)
> eData.Ex <- estimateDispersions(eData.Ex)
> eData.Ex <- fitDispersionFunction(eData.Ex)
> eData.Ex <- testForDEU(eData.Ex)
> eData.Ex <- estimatelog2FoldChanges(eData.Ex)
> tt.Ex <- DEUresultTable(eData.Ex)
```

Run the analysis for the edge count data set.

```
> eData.Edge <- estimateSizeFactors(eData.Edge)
> eData.Edge <- estimateDispersions(eData.Edge)
> eData.Edge <- fitDispersionFunction(eData.Edge)
> eData.Edge <- testForDEU(eData.Edge)
> eData.Edge <- estimatelog2FoldChanges(eData.Edge)
> tt.Edge <- DEUresultTable(eData.Edge)
```

First we want to compare the estimated dispersion for both cases. Figure 2 shows the resulting plot. The dispersion estimates look quite similar.

```
> plotDisp <- function(eData, case="") {
+   meanvalues <- rowMeans(counts(eData))
+   plot(meanvalues, fData(eData)$dispBeforeSharing,
+         main = paste(case, " mean vs CR dispersion", sep=":"),
+         frame.plot=FALSE, pch=16, cex=0.8, log = "xy",
+         ylab="Dispersion", xlab="Mean counts")
+   x <- 0.01:max(meanvalues)
+   y <- eData@dispFitCoefs[1] + eData@dispFitCoefs[2]/x
+   lines(x, y, col = "purple", lwd=2)
+ }
> fn <- "Disp-plots.png"
> png(fn,  width=11, height=5.5, res=300, units="in")
> par(mfrow=c(1,2))
> plotDisp(eData.Ex, "Exons")
> plotDisp(eData.Edge, "Edges")
> dev.off()
```

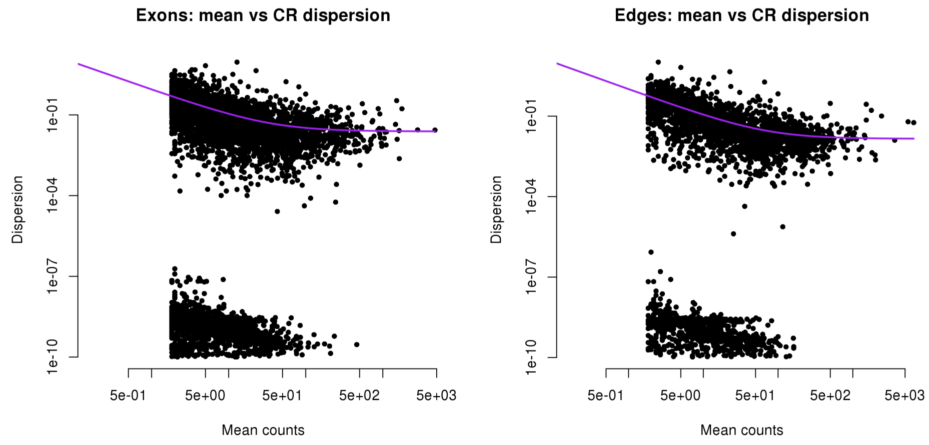

Figure 2: Dispersion estimates. Left: classic exon model. Right: splicing graph edge model

5.2 DESeq analysis for single element genes

In this section the *DESeq* analysis of the single edge and single exon genes is performed. First we create count data sets and set up the testing conditions for the differential expression analysis.

```
> library(DESeq)
> cds.Ex <- newCountDataSet(countData=cD.exsByGenes,
+                           conditions=design)
> cds.Ex <- cds.Ex[sglExGns,]
> cds.Edge <- newCountDataSet(countData=cD.exByEdge[sglEdgeGns,],
+                             conditions=design)
> cds.Edge <- cds.Edge[sglExGns,]
```

Next we compute the size factor, estimate the dispersion and test for differentially expression for both sets of single elements

Test for differential expression of single exon genes.

```
> cds.Ex <- estimateSizeFactors( cds.Ex )
> cds.Ex <- estimateDispersions( cds.Ex )
> ttSingle.Ex <- nbinomTest( cds.Ex,"SOC", "Benign" )
```

Test for differential expression of single edge genes.

```
> cds.Edge <- estimateSizeFactors( cds.Edge )
> cds.Edge <- estimateDispersions( cds.Edge )
> ttSingle.Edge <- nbinomTest( cds.Edge,"SOC", "Benign" )
```

Now the differential expression analysis for all elements is completed. As a last step the results of the *DEXSeq* and the *DESeq* analysis are combined into one common top table containing the fold changes and their associated p-values.

```
> int.DEXSeq <- c("exonID", "pvalue", "padjust",
+                 "meanBase", "log2fold(SOC/Benign)")
> int.DESeq <- c("id", "pval", "padj", "baseMean",
+                "log2FoldChange")
> temp <- tt.Edge[, int.DEXSeq]
> colnames(temp) <- int.DESeq
> finTT.Edge <- rbind(ttSingle.Edge[, int.DESeq], temp)
> finTT.Edge$gnID <- gnToEdge[finTT.Edge$id,]$gnIDs
> temp <- tt.Ex[, int.DEXSeq]
> colnames(temp) <- int.DESeq
> finTT.Ex <- rbind(ttSingle.Ex[, int.DESeq], temp)
> finTT.Ex$gnID <- gnNames[finTT.Ex$id]
```

Now we have a look onto the individual top tables to see if there are some genes in common between the exon model results and the edge model results.

6 Comparing the edge and the exon models

This section deals with comparing the results of the exon model approach with the results of the edge model approach. The aim of utilizing the splicing graph methodology was to get more robust results and to increase the statistical power of the tests for differential expression. Remember that before constructing the splicing graph some of the exons got disjoint in smaller non overlapping parts, when overlapping exons of the same gene where present. By just using the original exon structure, still containing overlapping parts of exons, the counting process would discard reads which map to those parts because of their ambiguousness. The first indication for which method could be more appropriate is to retrieve the total number of raw counts of the exons count table and the edges count table.

```
> sum(cD.exByEdge)

[1] 1653813

> sum(cD.exsByGenes)

[1] 1142598

> sum(cD.exsByGenes*100)/sum(cD.exByEdge)

[1] 69.08871
```

The comparison of the raw counts reveals that the exon count table contains about 30% fewer counts than the edge count table. Regarding the total number of counts it seems that using the splicing graph approach leads to some improvement. But what does this mean for the differential expression analysis. Does more counts provide more robust results? What about the number of differentially expressed elements?

To gain more insight into this issue mean expression versus fold change plots are created to visualize the differences in expression of the individual edges and exons between SOC samples and benign samples. See Figure 3 for the resulting plot. By considering the fold change region about -4 in both plots it is noticeable that the proportion of significant edges to non significant edges is higher than the proportion of significant exons to non significant exons. This in turn could be an indication for more robust results.

```
> fn <- "MA-plots.png"
> png(fn,  width=11, height=5.5, res=300, units="in")
> par(mfrow=c(1,2))
> plot(finTT.Ex$baseMean, finTT.Ex$log2FoldChange, log = "x",
+      col = ifelse(finTT.Ex$padj < 0.1, "purple", "black"),
+      ylim = c(-5, 5), main = "Exons MvsA", pch=16, cex=0.7,
+      frame.plot=FALSE)
> plot(finTT.Edge$baseMean, finTT.Edge$log2FoldChange,
+      col = ifelse(finTT.Edge$padj < 0.1, "purple", "black"),
+      ylim = c(-5, 5), main = "Edges MvsA", pch=16, cex=0.7,
+      log = "x", frame.plot=FALSE)
> dev.off()
```

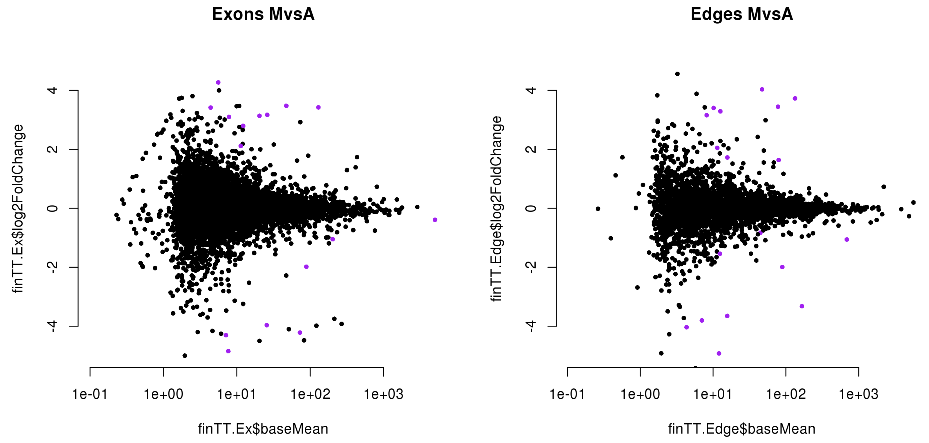

Figure 3: Mean expression vs log2 fold change plot, significant ($p < 0.1$) hits are colored in purple. Left: classic exon model. Right: splicing graph edge model

Next we want to see which percentage of exons and edges is significantly differentially expressed between SOC samples and benign samples.

```
> ex.test <- nrow(finTT.Ex)
> edge.test <- nrow(finTT.Edge)
> sig.diff.ex <- sum(finTT.Ex$padj < 0.05, na.rm=TRUE)
> sig.diff.edge <- sum(finTT.Edge$padj < 0.05, na.rm=TRUE)
> sig.diff.ex*100/ex.test
```

```
[1] 0.1005699
```

```
> sig.diff.edge*100/edge.test

[1] 0.2431611
```

To visualize the final results of the edges versus exons comparison we create a plot where different p-value cut offs are plotted against the associated number of significant differentially expressed exons and edges.

```
> p.cuts <- 10^seq( 0, -6, length.out=100 )
> ps.Ex <- sapply(p.cuts, function(p.cut) {
+   sum(finTT.Ex$pval < p.cut, na.rm=TRUE)*100/
+     length(finTT.Ex$pval)
+ })
> ps.Edge <- sapply(p.cuts, function(p.cut) {
+   sum(finTT.Edge$pval < p.cut, na.rm=TRUE)*100/
+     length(finTT.Edge$pval)
+ })
> fn <- "Comparison-edges-exons.png"
> png(fn,  width=7, height=7, res=300, units="in")
> plot(ps.Edge, p.cuts, type="l", col="purple",
+      xlab="% of differentially expressed elements",
+      ylab="P-value cut off", lwd=2, frame.plot=FALSE,
+      log="xy", main="Exons versus edges")
> lines(ps.Ex, p.cuts, lty=2, lwd=2)
> grid(lwd=2)
> legend("bottomright", legend=c("Edges", "Exons"),
+        col=c("purple", "black"), lty=c(1,2), lwd=2,
+        border=NA, box.col=NA, bg=NA)
> dev.off()
```

Figure 4 shows the comparison of both approaches regarding the percentage of significantly differentially expressed elements. By looking onto the characteristics of the curves it seems that the splice graph edges approach provides more statistical power than the classic *DEXSeq* approach.

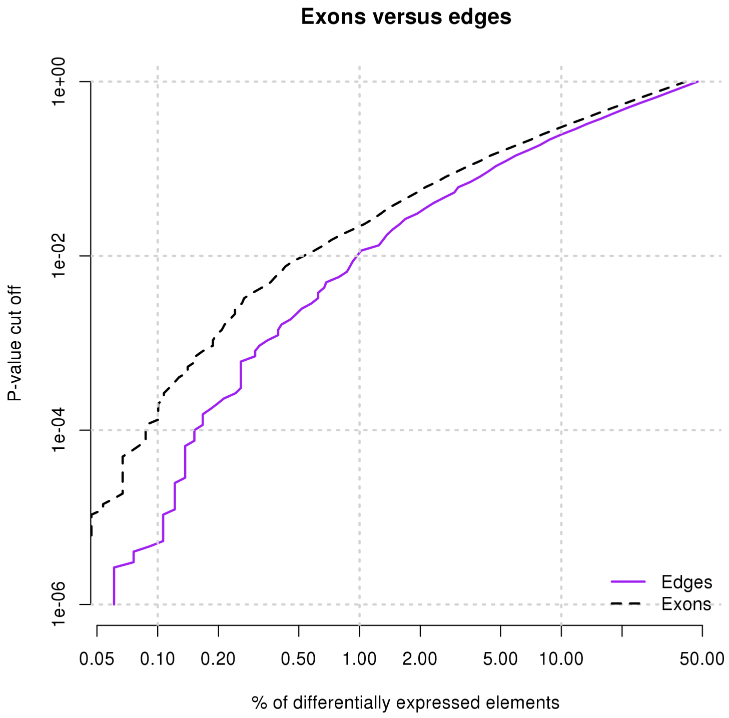

Figure 4: Comparison of differential expression results of edges versus exons.

Table 1 and Table 2 show the 10 most significant differentially expressed edges and exons. The most promising hit common in both tables is entrez gene id 90522 (YIF1B, Yip1 interacting factor homolog B). This protein is a multi-pass membrane protein, but there is not much known about its function. However it just serves as an example in here.

id	pval	padj	baseMean	log2FoldChange	gnID
243810	0.00	0.00	202.48	-1.05	259307
239871	0.00	0.00	129.44	3.42	90522
244364	0.00	0.00	12.21	2.80	89790
244365	0.00	0.00	7.11	-4.30	89790
238289	0.00	0.00	5020.36	-0.39	388524
235053	0.00	0.00	88.72	-1.98	1785
245411	0.00	0.00	11.72	-Inf	7138
243917	0.00	0.00	14.39	-5.54	79784
243927	0.00	0.00	72.43	-4.21	79784
238288	0.00	0.01	5.59	4.27	388524

Table 1: Top table of the exon model

id	pval	padj	baseMean	log2FoldChange	gnID
71269	0.00	0.00	133.59	3.73	90522
70565	0.00	0.00	12.59	3.29	89790
70559	0.00	0.00	7.02	-3.80	89790
6272	0.00	0.00	44.65	-0.83	112703
24343	0.00	0.00	47.09	4.03	2788
38026	0.00	0.00	79.61	1.64	51599
14781	0.00	0.00	89.08	-1.99	1785
38040	0.00	0.00	166.21	-3.32	51599
68408	0.00	0.00	4.32	-4.04	84941
10076	0.00	0.01	123.75	-8.62	1311

Table 2: Top table of the edge model

Last we want to have a look on the individual counts of the elements of entrez gene 90522 (YIF1B). For visualizing the counts of this gene we use the plot routines of the *DEXSeq* package.

```
> plotExp <- function(fn, eDat) {
+   png(fn,  width=11, height=5.5, res=300, units="in")
+   par(mfrow=c(1,2))
+   COL <- c("#3399FF", "#FF3333")
+   plotDEXSeq(eDat, "90522", cex.axis = 1.2, cex = 1.3,
+              lwd = 2, legend = TRUE, color=COL,
+               color.samples=COL[design], splicing=TRUE)
+   plotDEXSeq(eDat, "90522", expression = FALSE,
+               norCounts = TRUE, cex.axis = 1.2, cex = 1.3,
+               lwd = 2, legend = TRUE,
+               color=COL,
+               color.samples=COL[design], splicing=TRUE)
+   dev.off()
+ }
> fn <- "Edge-plot.png"
> plotExp(fn, eData.Edge)
> fn <- "Ex-plot.png"
> plotExp(fn, eData.Ex)
```

Figure 5 and 6 show the normalized count data for the splice graph edges model and for the classic exons model. Also the corresponding edges and exons model is shown. In the case of gene 90522 (YIF1B) both approaches come to the similar results, but it is noticeable that in the exon model case some exons have zero counts, which are indicated by white filled rectangles, in one condition, whereas in the edges model case we still have counts. This is another sign for increased robustness in the splice graph edges approach compared to the classic exon model approach.

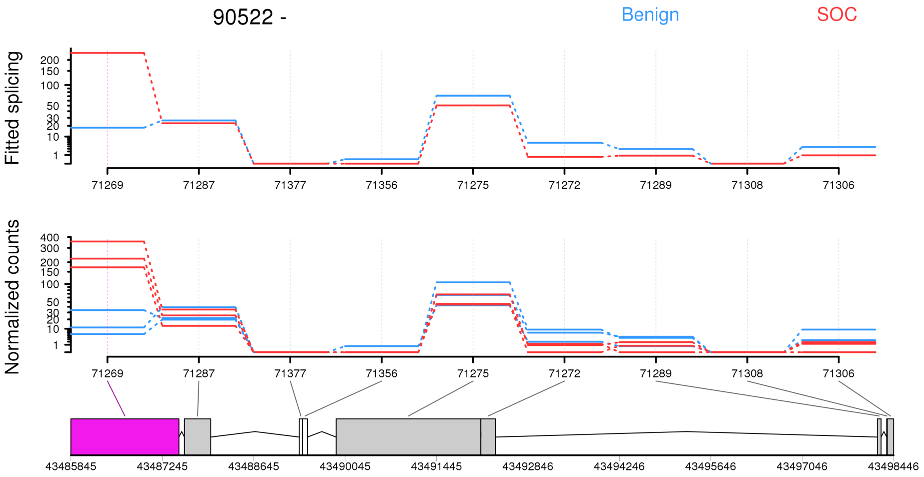

Figure 5: Expression levels of the edges of gene 90522 (YIF1B). The purple rectangle in the edges model indicate significant (FDR<0.1) differentially expressed edges. Grey rectangles indicate edges with at least on read count and white rectangles reflect edges with zero counts.

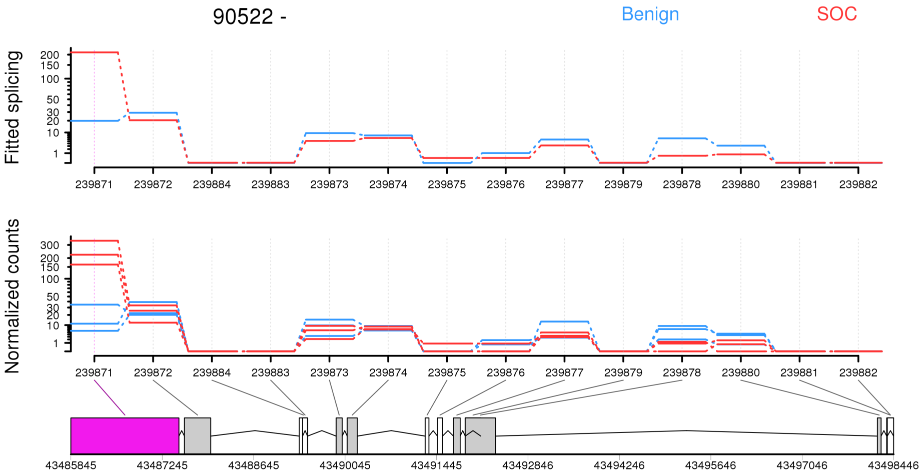

Figure 6: Expression levels of the exons of gene 90522 (YIF1B). The purple rectangle in the exons model indicate significant (FDR<0.1) differentially expressed exons. Grey rectangles indicate exons with at least on read count and white rectangles reflect exons with zero counts.

7 Summary

The introduced splice graph approach to identify differential expression of annotated splice variants based on RNA-seq data seems to be promising. Especially a higher number of total read counts, increased robustness as well as increased statistical power for the testing seem to be the most remarkable advantages of this new methodology.

References

[1] Simon Anders and Wolfgang Huber. Differential expression analysis for sequence count data. *Genome Biology*, 11:R106, 2010.

[2] Steffen Heber, Max Alekseyev, Sing-Hoi Sze, Haixu Tang, and Pavel A. Pevzner. Splicing graphs and est assembly problem. 18(suppl 1):S181–S188, 2002.

[3] Michael Sammeth. Complete alternative splicing events are bubbles in splicing graphs. 16:1117–1140, 2010.

8 Session Information

```
R version 2.15.1 (2012-06-22)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=C                 LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 [1] xtable_1.7-0
 [2] DESeq_1.8.3
 [3] locfit_1.5-8
 [4] DEXSeq_1.2.1
 [5] TxDb.Hsapiens.UCSC.hg18.knownGene_2.7.1
 [6] GenomicFeatures_1.8.3
 [7] AnnotationDbi_1.18.1
 [8] Biobase_2.16.0
 [9] GenomicRanges_1.8.12
[10] IRanges_1.14.4
[11] BiocGenerics_0.2.0
```

```
loaded via a namespace (and not attached):
 [1] BSgenome_1.24.0    Biostrings_2.24.1  DBI_0.2-5          RColorBrewer_1.0-5
 [5] RCurl_1.91-1       RSQLite_0.11.1     Rsamtools_1.8.6    XML_3.9-4
 [9] annotate_1.34.1    biomaRt_2.12.0     bitops_1.0-4.1     genefilter_1.38.0
[13] geneplotter_1.34.0 grid_2.15.1        hwriter_1.3        lattice_0.20-10
[17] plyr_1.7.1         rtracklayer_1.16.3 splines_2.15.1     statmod_1.4.15
[21] stats4_2.15.1      stringr_0.6.1      survival_2.36-14   tools_2.15.1
[25] zlibbioc_1.2.0
```