Splicing graphs and RNA-seq data

Hervé Pagès Daniel Bindreither Marc Carlson Martin Morgan

Last modified: January 2019; Compiled: October 29, 2024

Contents

1 Introduction

The *SplicingGraphs* package allows the user to create and manipulate splicing graphs [1] based on annotations for a given organism. Annotations must describe a *gene model*, that is, they need to contain the following information:

- The exact exon/intron structure (i.e., genomic coordinates) for the known transcripts.
- The grouping of transcripts by gene.

Annotations need to be provided as a *TxDb* or *GRangesList* object, which is how a gene model is typically represented in Bioconductor.

The *SplicingGraphs* package defines the *SplicingGraphs* container for storing the splicing graphs together with the gene model that they are based on. Several methods are provided for conveniently access the information stored in a *SplicingGraphs* object. Most of these methods are described in this document.

The package also allows the user to assign RNA-seq reads to the edges of a *SplicingGraphs* object and to summarize them. This requires that the reads have been previously aligned to the exact same reference genome that the gene model is based on. RNA-seq data from an already published study is used to illustrate this functionality. In that study [2], the authors performed transcription profiling by high throughput sequencing of HNRNPC knockdown and control HeLa cells (*Human*).

2 Splicing graphs

Alternative splicing is a frequently observed complex biological process which modifies the primary RNA transcript and leads to transcript variants of genes. Those variants can be plentiful. Especially for large genes it is often difficult to describe their structure in a formal, logical, short and convenient way. To capture the full variety of splicing variants of a certain gene in one single data structure, Heber at al [6] introduced the term *splicing graph* and provided a formal framework for representing the different choices of the splicing machinery.

2.1 Definitions and example

For a comprehensive explanation of the splicing graph theory, please refer to [6, 1].

A splicing graph is a directed acyclic graph (DAG) where:

- Vertices (a.k.a. *nodes*) represent the splicing sites for a given gene. Splicing sites are ordered by their position from 5' to 3' and numbered from `1` to `n`. This number is the *Splicing Site id*. Splicing graphs are only defined for genes that have all the exons of all their transcripts on the same chromosome and strand. In particular, in its current form, the splicing graph theory cannot describe *trans-splicing* events.
- Edges are the exons and introns between splicing sites. Their orientation follows the 5' to 3' direction, i.e., they go from low to high *Splicing Site ids*.

Two artificial nodes that don't correspond to any splicing site are added to the graph: the *root* (`R`) and the *leaf* (`L`). Also artificial edges are added so that all the transcripts are represented by a path that goes from `R` to `L`. That way the graph is connected. Not all paths in the graph are necessarily supported by a transcript.

Figure 1 shows the splicing graph representation of all known transcript variants of Human gene CIB3 (Entrez ID 117286).

2.2 Details about nodes and edges

Splicing sites can be of the following types:

- *Acceptor* site (symbol: `-`);
- *Donor* site (symbol: `^`);
- *Transcription start* site (symbol: `[`);
- *Transcription end* site (symbol: `]`).

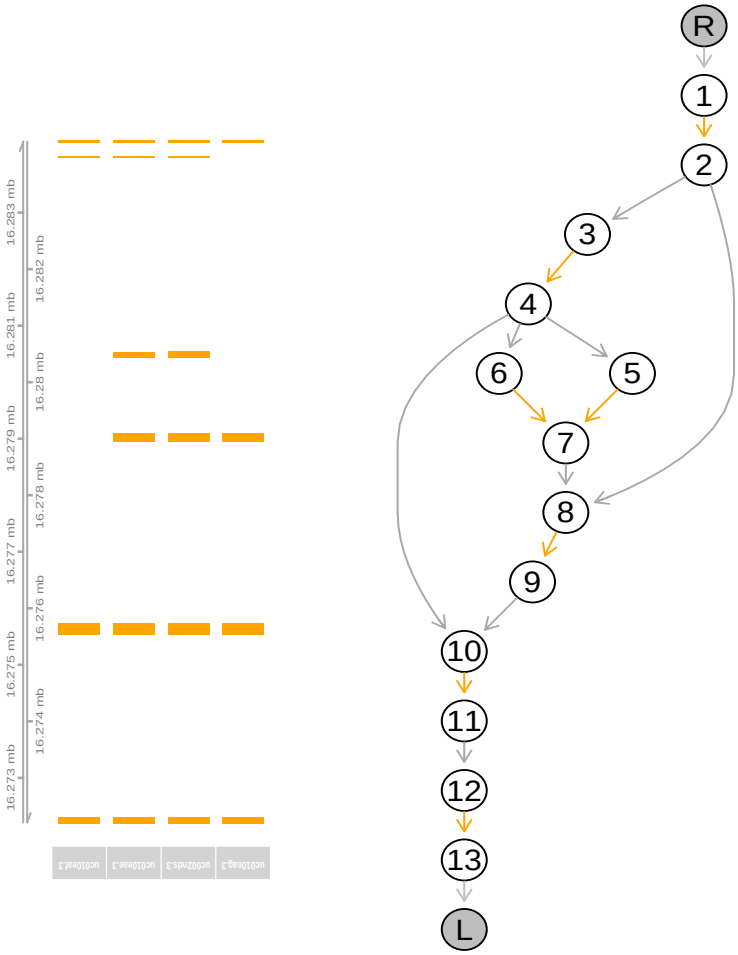

Figure 1: Splicing graph representation of the four transcript variants of gene CIB3 (Entrez ID 117286). Left: transcript representation. Right: splicing graph representation. Orange arrows are edges corresponding to exons.

The symbols associated with individual types of nodes shown in braces above are used for the identification of complete alternative splicing events. For more information about gathering the alternative splicing events and their associated codes, please refer to [6, 1].

In some cases a given splicing site can be associated with 2 types. The combinations of types for such sites is limited. Only *acceptor* (`-`) and *transcription start* (`[`), or *donor* (`^`) and *transcription end* (`]`), can occur as a pair of types. Such a dual-type node occurs when the start of an exon, which is not the first exon of a transcript, falls together with the transcription start site of another transcript of the same gene. An example of such a splicing graph is shown in Figure 2.

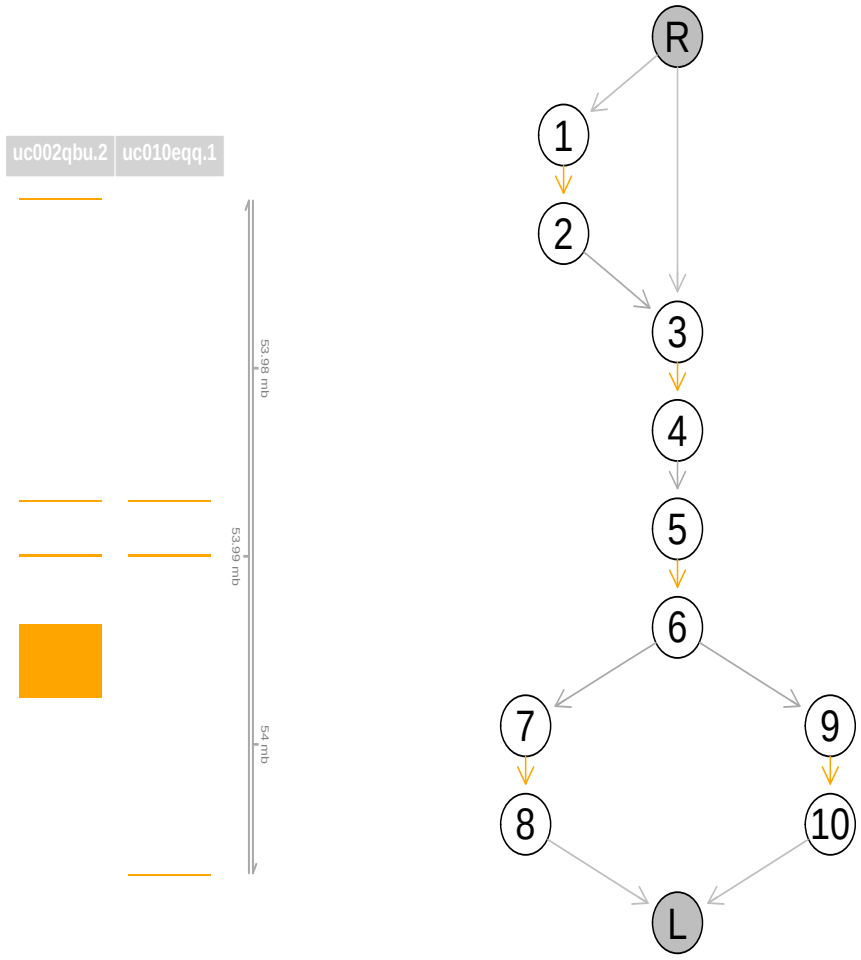

Figure 2: Splicing graph representation of the two transcript variants of Human gene ZNF813 (Entrez ID 126017). Node 3 is a (`-`,`[`) dual-type node: *acceptor* (`-`) for transcript `uc002qbu.2`, and *transcription start* (`[`) for transcript `uc010eqq.1`.

A similar case occurs when the end of an exon, which is not the last exon of a transcript, falls together with the transcription end site of another transcript of the same gene.

Whether a non-artificial edge represents an exon or an intron is determined by the types of its two flanking nodes:

- **Exon** if going from *acceptor* (-), or (-,[) dual-type, to *donor* (^), or (^,]) dual-type;
- **Intron** if the otherway around.

2.3 Uninformative nodes

Not all splicing sites (nodes) on the individual transcripts are alternative splicing sites. Therefore the initial splicing graph outlined above can be simplified by removing nodes of in- and out- degree equal to one because they are supposed to be non-informative in terms of alternative splicing. Edges associated with such nodes get sequentially merged with the previous ones and result in longer edges capturing adjacent exons/introns. The final splicing graph only contains nodes involved in alternative splicing. The result of removing uninformative nodes is illustrated in Figure 3.

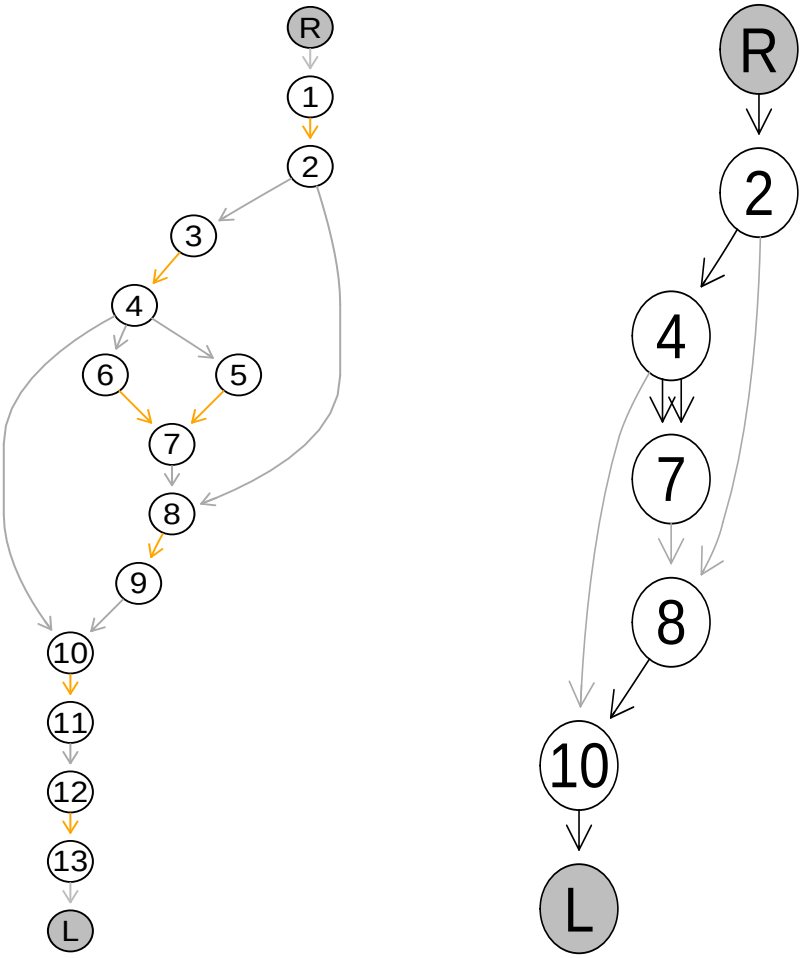

Figure 3: Nodes 1, 3, 5, 6, 9, 11, 12, and 13, are uninformative and can be removed. Left: before removal of the uninformative nodes. Right: after their removal.

3 Computing splicing graphs from annotations

3.1 Choosing and loading a gene model

The starting point for computing splicing graphs is a set of annotations describing a *gene model* for a given organism. In Bioconductor, a gene model is typically represented as a *TxDb* object. A few prepackaged *TxDb* objects are available as *TxDb.** annotation packages in the Bioconductor package repositories. Here the *TxDb.Hsapiens.UCSC.hg19.knownGene* package is used. If there is no prepackaged *TxDb* object for the genome/track that the user wants to use, such objects can easily be created by using tools from the *GenomicFeatures* package.

First we load the *TxDb.** package:

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
```

Creating the splicing graphs for all genes in the *TxDb* object can take a long time (up to 20 minutes or more). In order to keep things running in a reasonable time in this vignette, we restrict the gene model to the genes located on chromosome 14. Let's use the `isActiveSeq` getter/setter for this. By default, all the chromosomes in a *TxDb* object are "active":

```
> isActiveSeq(txdb)[1:25]
```

```
 chr1  chr2  chr3  chr4  chr5  chr6  chr7  chr8  chr9 chr10 chr11 chr12 chr13 chr14 chr15
 TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
chr16 chr17 chr18 chr19 chr20 chr21 chr22  chrX  chrY  chrM
 TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Next we set all the values in this named logical vector to `FALSE`, except the value for `chr14`:

```
> isActiveSeq(txdb)[-match("chr14", names(isActiveSeq(txdb)))] <- FALSE
> names(which(isActiveSeq(txdb)))
```

```
[1] "chr14"
```

3.2 Generating a *SplicingGraphs* object

Splicing graphs are computed by calling the `SplicingGraphs` function on the gene model. This is the constructor function for *SplicingGraphs* objects. It will compute information about all the splicing graphs (1 per gene) and store it in the returned object. By default, only genes with at least 2 transcripts are considered (this can be changed by using the `min.ntx` argument of `SplicingGraphs`):

```
> library(SplicingGraphs)
> sg <- SplicingGraphs(txdb)  # should take between 5 and 10 sec on
>                             # a modern laptop
> sg
```

```
SplicingGraphs object with 452 gene(s) and 1997 transcript(s)
```

sg is a *SplicingGraphs* object. It has 1 element per gene and `names(sg)` gives the gene ids:

```
> names(sg)[1:20]
```

```
 [1] "10001"     "100129075" "100309464" "10038"     "100505967" "100506412" "100506499"
 [8] "100529257" "100529261" "100874185" "100996280" "10175"     "10202"     "10243"
[15] "10278"     "1033"      "10379"     "10419"     "10484"     "10490"
```

`seqnames(sg)` and `strand(sg)` return the chromosome and strand of the genes:

```
> seqnames(sg)[1:20]
```

```
    10001 100129075 100309464     10038 100505967 100506412 100506499 100529257 100529261
    chr14     chr14     chr14     chr14     chr14     chr14     chr14     chr14     chr14
100874185 100996280     10175     10202     10243     10278      1033     10379     10419
    chr14     chr14     chr14     chr14     chr14     chr14     chr14     chr14     chr14
    10484     10490
    chr14     chr14
Levels: chr14
```

```
> strand(sg)[1:20]
```

```
    10001 100129075 100309464     10038 100505967 100506412 100506499 100529257 100529261
        -         -         +         +         +         +         -         -         +
100874185 100996280     10175     10202     10243     10278      1033     10379     10419
        -         +         -         +         +         -         +         +         -
    10484     10490
        -         -
Levels: + - *
```

```
> table(strand(sg))
```

```
  +   -   *
232 220   0
```

The number of transcripts per gene can be obtained with `elementNROWS(sg)`:

```
> elementNROWS(sg)[1:20]
```

```
    10001 100129075 100309464     10038 100505967 100506412 100506499 100529257 100529261
        5         2         2         4         2         4         2         3         6
100874185 100996280     10175     10202     10243     10278      1033     10379     10419
        2         2         2         4         6         3         4         2         6
    10484     10490
        4         3
```

`elementNROWS` is a core accessor for list-like objects that returns the lengths of the individual list elements.

At this point you might wonder why `elementNROWS` works on *SplicingGraphs* objects. Does this mean that those objects are list-like objects? The answer is yes. What do the list elements look like, and how can you access them? This is answered in the next subsection.

3.3 Basic manipulation of a *SplicingGraphs* object

The list elements of a list-like object can be accessed **one at a time** by subsetting the object with `[[` (a.k.a. double-bracket subsetting). On a *SplicingGraphs* object, this will extract the transcripts of a given gene. More precisely it will return an **unnamed** *GRangesList* object containing the exons of the gene grouped by transcript:

```
> sg[["3183"]]
```

```
GRangesList object of length 11:
[[1]]
GRanges object with 9 ranges and 5 metadata columns:
      seqnames            ranges strand |   exon_id   exon_name exon_rank start_SSid
         <Rle>         <IRanges>  <Rle> | <integer> <character> <integer>  <integer>
  [1]    chr14 21737457-21737638      - |    184118        <NA>         1          2
  [2]    chr14 21731470-21731495      - |    184116        <NA>         2          6
  [3]    chr14 21702112-21702388      - |    184113        <NA>         3         11
  [4]    chr14 21699156-21699231      - |    184112        <NA>         4         13
  [5]    chr14 21698478-21698525      - |    184109        <NA>         5         17
  [6]    chr14 21681119-21681276      - |    184107        <NA>         6         21
  [7]    chr14 21679969-21680082      - |    184105        <NA>         7         24
  [8]    chr14 21679565-21679725      - |    184103        <NA>         8         27
  [9]    chr14 21677296-21679465      - |    184100        <NA>         9         30
       end_SSid
      <integer>
  [1]         1
  [2]         5
  [3]         9
  [4]        12
  [5]        16
  [6]        18
  [7]        22
  [8]        25
  [9]        28
  -------
  seqinfo: 1 sequence from hg19 genome

[[2]]
GRanges object with 5 ranges and 5 metadata columns:
      seqnames            ranges strand |   exon_id   exon_name exon_rank start_SSid
         <Rle>         <IRanges>  <Rle> | <integer> <character> <integer>  <integer>
  [1]    chr14 21737457-21737638      - |    184118        <NA>         1          2
  [2]    chr14 21731470-21731495      - |    184116        <NA>         2          6
  [3]    chr14 21702237-21702388      - |    184114        <NA>         3         10
  [4]    chr14 21679565-21679672      - |    184102        <NA>         4         27
  [5]    chr14 21677296-21679465      - |    184100        <NA>         5         30
       end_SSid
      <integer>
  [1]         1
```

```
  [2]         5
  [3]         9
  [4]        26
  [5]        28
  -------
  seqinfo: 1 sequence from hg19 genome

...
<9 more elements>
```

The exon-level metadata columns are:

- `exon_id`: The original internal exon id as stored in the *TxDb* object. This id was created and assigned to each exon when the *TxDb* object was created. It's not a public id like, say, an Ensembl, RefSeq, or GenBank id. Furthermore, it's only guaranteed to be unique **within** a *TxDb* object, but not **across** *TxDb* objects.
- `exon_name`: The original exon name as provided by the annotation resource (e.g., UCSC, Ensembl, or GFF file) and stored in the *TxDb* object when it was created. Set to `NA` if no exon name was provided.
- `exon_rank`: The rank of the exon in the transcript.
- `start_SSid`, `end_SSid`: The *Splicing Site ids* corresponding to the *start* and *end* coordinates of the exon. (Please be cautious to not misinterpret the meaning of *start* and *end* here. See IMPORTANT NOTE below.) Those ids were assigned by the `SplicingGraphs` constructor.

IMPORTANT NOTE: Please be aware that the *start* and *end* coordinates of an exon, like the *start* and *end* coordinates of a genomic range in general, are following the almost universal convention that *start* is <= *end*, and this **regardless of the direction of transcription**.

As mentioned previously, the *Splicing Site ids* are assigned based on the order of the site positions from 5' to 3'. This means that, for a gene on the plus (resp. minus) strand, the ids in the `start_SSid` metadata column are always lower (resp. greater) than those in the `end_SSid` metadata column.

However, on both strands, the *Splicing Site id* increases with the rank of the exon.

The `show` method for *GRangesList* objects only displays the inner metadata columns (which are at the exon level for an object like `sg[["3183"]]`). To see the outer metadata columns (transcript-level metadata columns for objects like `sg[["3183"]]`), we need to extract them explicitely:

```
> mcols(sg[["3183"]])
```

```
DataFrame with 11 rows and 2 columns
         tx_id        txpath
   <character> <IntegerList>
1   uc001vzw.3     1,2,5,...
2   uc001vzx.3     1,2,5,...
3   uc001vzy.3     1,2,5,...
4   uc001vzz.3     1,2,9,...
5   uc001waa.3     1,2,9,...
6   uc001wac.3     1,2,9,...
7   uc001wad.3     1,2,5,...
8   uc010ail.3     1,2,3,...
9   uc010tlq.2     1,2,5,...
10  uc010tlr.2  15,17,18,...
11  uc001wae.3     1,2,9,...
```

The transcript-level metadata columns are:

- `tx_id`: The original transcript id as provided by the annotation resource (e.g. UCSC, Ensembl, or GFF file) and stored in the *TxDb* object when it was created.
- `txpath`: A named list-like object with one list element per transcript in the gene. Each list element is an integer vector that describes the *path* of the transcript, i.e., the *Splicing Site ids* that it goes thru.

```
> mcols(sg[["3183"]])$txpath
```

```
IntegerList of length 11
[["uc001vzw.3"]] 1 2 5 6 9 11 12 13 16 17 18 21 22 24 25 27 28 30
[["uc001vzx.3"]] 1 2 5 6 9 10 26 27 28 30
[["uc001vzy.3"]] 1 2 5 6 9 11 12 14 16 17 18 21 22 24 25 27 28 30
[["uc001vzz.3"]] 1 2 9 11 12 13 16 17 18 21 22 24 25 27 28 30
[["uc001waa.3"]] 1 2 9 11 12 14 16 17 18 21 22 24 25 27 28 30
[["uc001wac.3"]] 1 2 9 11 12 13 16 17 18 19 23 24 25 27 28 30
[["uc001wad.3"]] 1 2 5 6 9 10 16 17 18 21 22 24 25 27 28 30
[["uc010ail.3"]] 1 2 3 4 5 6 7 8 9 11 12 14 16 17 18 21 22 24 25 27 28 30
[["uc010tlq.2"]] 1 2 5 6 9 11 12 14 20 21 22 24 25 27 28 30
[["uc010tlr.2"]] 15 17 18 21 22 24 25 29
...
<1 more element>
```

A more convenient way to extract this information is to use the `txpath` accessor:

```
> txpath(sg[["3183"]])
```

```
IntegerList of length 11
[["uc001vzw.3"]] 1 2 5 6 9 11 12 13 16 17 18 21 22 24 25 27 28 30
[["uc001vzx.3"]] 1 2 5 6 9 10 26 27 28 30
[["uc001vzy.3"]] 1 2 5 6 9 11 12 14 16 17 18 21 22 24 25 27 28 30
[["uc001vzz.3"]] 1 2 9 11 12 13 16 17 18 21 22 24 25 27 28 30
[["uc001waa.3"]] 1 2 9 11 12 14 16 17 18 21 22 24 25 27 28 30
[["uc001wac.3"]] 1 2 9 11 12 13 16 17 18 19 23 24 25 27 28 30
[["uc001wad.3"]] 1 2 5 6 9 10 16 17 18 21 22 24 25 27 28 30
[["uc010ail.3"]] 1 2 3 4 5 6 7 8 9 11 12 14 16 17 18 21 22 24 25 27 28 30
[["uc010tlq.2"]] 1 2 5 6 9 11 12 14 20 21 22 24 25 27 28 30
[["uc010tlr.2"]] 15 17 18 21 22 24 25 29
...
<1 more element>
```

The list elements of the `txpath` metadata column always consist of an even number of *Splicing Site ids* in ascending order.

The transcripts in a *GRangesList* object like `sg[["3183"]]` can be plotted with `plotTranscripts`:

```
> plotTranscripts(sg[["3183"]])
```

The resulting plot is shown on figure 4.

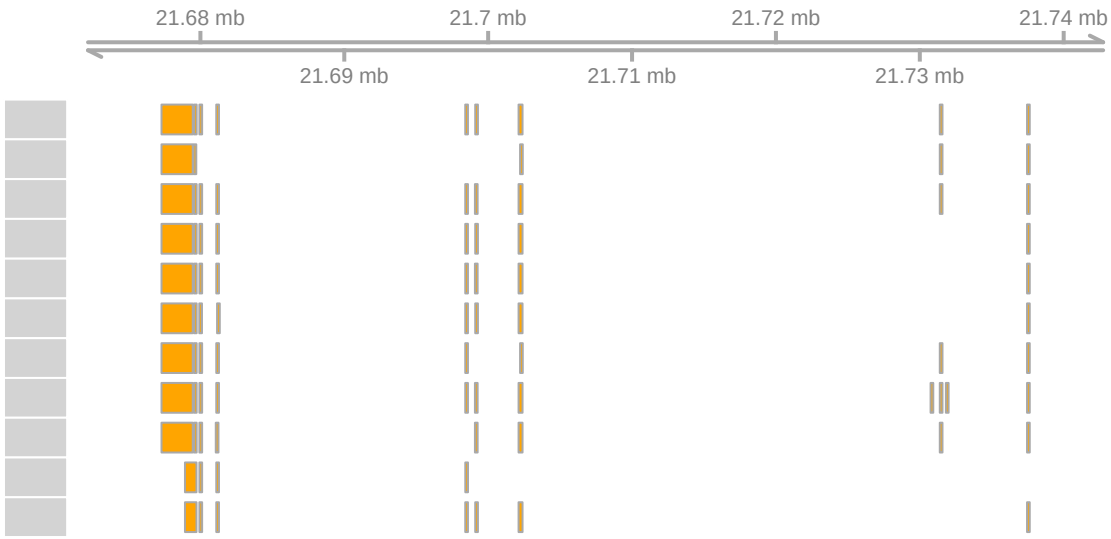

Figure 4: The 11 transcripts of gene HNRNPC (Entrez ID 3183).

SplicingGraphs objects, like most list-like objects, can be unlisted with `unlist`. This will extract the transcripts of all the genes and return them as a **named** *GRangesList* object. The names on the object are the gene ids:

```
> ex_by_tx <- unlist(sg)
> head(names(ex_by_tx))
```

```
[1] "10001"     "10001"     "10001"     "10001"     "10001"     "100129075"
```

Because each element in the object represents a transcript (and not a gene), the names are not unique! This means that trying to subset the object by name (e.g. with `ex_by_tx["3183"]` or `ex_by_tx[["3183"]]`) is probably a bad idea because this will only select the first element with that name. When the names on a vector-like object `x` are not unique, a safe way to select all the elements with some given names is to do something like `x[names(x) %in% c("name1", "name2")]`. For example, to select all the transcripts from genes 10001 and 100129075:

```
> ex_by_tx[names(ex_by_tx) %in% c("10001", "100129075")]
```

```
GRangesList object of length 7:
$`10001`
GRanges object with 8 ranges and 5 metadata columns:
      seqnames            ranges strand |   exon_id   exon_name exon_rank start_SSid
         <Rle>         <IRanges>  <Rle> | <integer> <character> <integer>  <integer>
  [1]    chr14 71067333-71067384      - |    186281        <NA>         1          2
  [2]    chr14 71064335-71064494      - |    186280        <NA>         2          4
  [3]    chr14 71063328-71063419      - |    186279        <NA>         3          6
  [4]    chr14 71060013-71060095      - |    186278        <NA>         4          8
  [5]    chr14 71059597-71059705      - |    186276        <NA>         5         11
  [6]    chr14 71057983-71058098      - |    186273        <NA>         6         15
  [7]    chr14 71052473-71052500      - |    186272        <NA>         7         17
```

```
  [8]    chr14 71050957-71051660      - |    186271        <NA>         8         19
       end_SSid
      <integer>
  [1]         1
  [2]         3
  [3]         5
  [4]         7
  [5]        10
  [6]        14
  [7]        16
  [8]        18
  -------
  seqinfo: 1 sequence from hg19 genome

$`10001`
GRanges object with 8 ranges and 5 metadata columns:
      seqnames            ranges strand |   exon_id   exon_name exon_rank start_SSid
         <Rle>         <IRanges>  <Rle> | <integer> <character> <integer>  <integer>
  [1]    chr14 71067333-71067384      - |    186281        <NA>         1          2
  [2]    chr14 71064335-71064494      - |    186280        <NA>         2          4
  [3]    chr14 71063328-71063419      - |    186279        <NA>         3          6
  [4]    chr14 71060013-71060095      - |    186278        <NA>         4          8
  [5]    chr14 71059597-71059726      - |    186277        <NA>         5         11
  [6]    chr14 71057983-71058098      - |    186273        <NA>         6         15
  [7]    chr14 71052473-71052500      - |    186272        <NA>         7         17
  [8]    chr14 71050957-71051660      - |    186271        <NA>         8         19
       end_SSid
      <integer>
  [1]         1
  [2]         3
  [3]         5
  [4]         7
  [5]         9
  [6]        14
  [7]        16
  [8]        18
  -------
  seqinfo: 1 sequence from hg19 genome

...
<5 more elements>
```

3.4 Extracting and plotting graphs from a *SplicingGraphs* object

The edges (resp. nodes) of the splicing graph of a given gene can be extracted with the `sgedges` (resp. `sgnodes`) function. An important caveat is that this can only be done for one gene at a time, or, said otherwise, these functions only work on a *SplicingGraphs* object of length 1. Here is where subsetting with `[` (a.k.a. single-bracket subsetting) comes into play.

Using `[` on a *SplicingGraphs* object returns a *SplicingGraphs* object containing only the selected genes:

```
> sg[strand(sg) == "-"]


SplicingGraphs object with 220 gene(s) and 957 transcript(s)
```

```
> sg[1:20]

SplicingGraphs object with 20 gene(s) and 68 transcript(s)

> tail(sg)  # equivalent to 'sg[tail(seq_along(sg))]'

SplicingGraphs object with 6 gene(s) and 19 transcript(s)

> sg["3183"]

SplicingGraphs object with 1 gene(s) and 11 transcript(s)
```

Let's extract the splicing graph edges for gene HNRNPC (Entrez ID 3183):

```
> sgedges(sg["3183"])

DataFrame with 41 rows and 5 columns
           from          to   sgedge_id ex_or_in                                tx_id
    <character> <character> <character> <factor>                  <CharacterList>
1             R           1    3183:R,1          uc001vzw.3,uc001vzx.3,uc001vzy.3,...
2             1           2    3183:1,2       ex uc001vzw.3,uc001vzx.3,uc001vzy.3,...
3             2           5    3183:2,5       in uc001vzw.3,uc001vzx.3,uc001vzy.3,...
4             5           6    3183:5,6       ex uc001vzw.3,uc001vzx.3,uc001vzy.3,...
5             6           9    3183:6,9       in uc001vzw.3,uc001vzx.3,uc001vzy.3,...
...         ...         ...         ...      ...                                   ...
37           20          21  3183:20,21       ex                          uc010tlq.2
38            R          15   3183:R,15                                   uc010tlr.2
39           15          17  3183:15,17       ex                          uc010tlr.2
40           25          29  3183:25,29       ex                uc010tlr.2,uc001wae.3
41           29           L   3183:29,L                        uc010tlr.2,uc001wae.3
```

The *DataFrame* object returned by `sgedges` has 1 row per edge. Its columns are explained below.

Let's extract the splicing graph nodes for that gene:

```
> sgnodes(sg["3183"])

 [1] "R"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14" "15" "16"
[18] "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28" "29" "30" "L"
```

The character vector returned by `sgnodes` contains the node ids, that is, `R` and `L` for the *root* and *leaf* nodes, and the *Splicing Site ids* for the other nodes. The node ids are always returned in ascending order with `R` and `L` being always the first and last nodes, respectively.

The *DataFrame* object returned by `sgedges` has the following columns:

- `from`, `to`: The 2 nodes connected by the edge.
- `sgedge_id`: A *global edge id* of the form `gene_id:from,to`.
- `ex_or_in`: The type of the edge, i.e., exon, intron, or no type if it's an artificial edge.
- `tx_id`: The ids of the transcripts that support the edge.

Alternatively the edges and ranges of all the genes can be extracted with `sgedgesByGene`:

```
> edges_by_gene <- sgedgesByGene(sg)
```

In this case the edges are returned in a *GRangesList* object where they are grouped by gene. `edges_by_gene` has the length and names of `sg`, that is, the names on it are the gene ids and are guaranteed to be unique. Let's look at the edges for gene 3183:

```
> edges_by_gene[["3183"]]
```

```
GRanges object with 37 ranges and 5 metadata columns:
       seqnames            ranges strand |        from          to   sgedge_id ex_or_in
          <Rle>         <IRanges>  <Rle> | <character> <character> <character> <factor>
   [1]    chr14 21737457-21737638      - |           1           2    3183:1,2       ex
   [2]    chr14 21731496-21737456      - |           2           5    3183:2,5       in
   [3]    chr14 21731470-21731495      - |           5           6    3183:5,6       ex
   [4]    chr14 21702389-21731469      - |           6           9    3183:6,9       in
   [5]    chr14 21702112-21702388      - |           9          11   3183:9,11       ex
   ...      ...               ...    ... .         ...         ...         ...      ...
  [33]    chr14 21702389-21730759      - |           8           9    3183:8,9       in
  [34]    chr14 21681204-21699116      - |          14          20  3183:14,20       in
  [35]    chr14 21681119-21681203      - |          20          21  3183:20,21       ex
  [36]    chr14 21698478-21698532      - |          15          17  3183:15,17       ex
  [37]    chr14 21678927-21679725      - |          25          29  3183:25,29       ex
                                    tx_id
                              <CharacterList>
   [1] uc001vzw.3,uc001vzx.3,uc001vzy.3,...
   [2] uc001vzw.3,uc001vzx.3,uc001vzy.3,...
   [3] uc001vzw.3,uc001vzx.3,uc001vzy.3,...
   [4] uc001vzw.3,uc001vzx.3,uc001vzy.3,...
   [5] uc001vzw.3,uc001vzy.3,uc001vzz.3,...
   ...                                 ...
  [33]                           uc010ail.3
  [34]                           uc010tlq.2
  [35]                           uc010tlq.2
  [36]                           uc010tlr.2
  [37]                uc010tlr.2,uc001wae.3
  -------
  seqinfo: 1 sequence from hg19 genome
```

The edge-level metadata columns are the same as the columns of the *DataFrame* object returned by `sgedges`. An important difference though is that the artificial edges (i.e., edges starting from the root node (`R`) or ending at the leaf node (`L`)) are omitted!

Finally, to plot a given splicing graph:

```
> plot(sg["3183"])
> plot(sgraph(sg["3183"], tx_id.as.edge.label=TRUE))
```

The resulting plots are shown on figure 5.

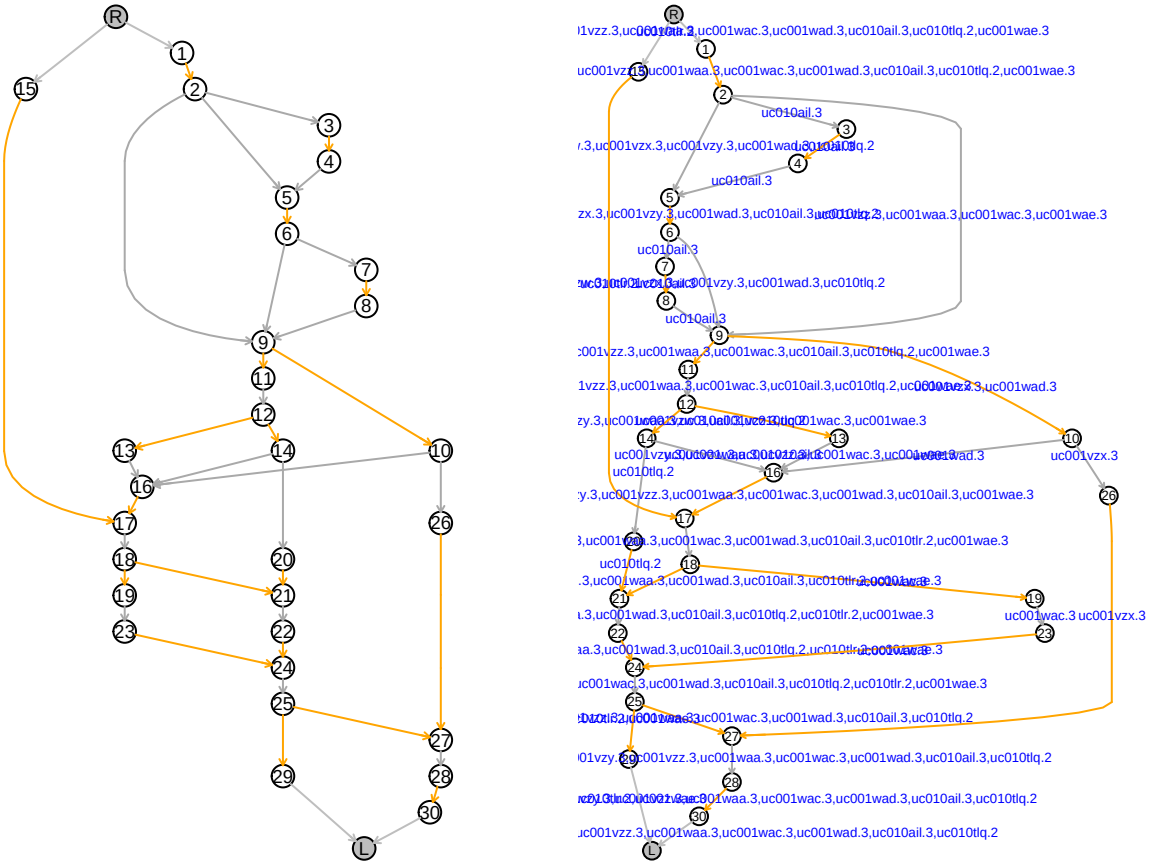

Figure 5: Splicing graph representation of gene HNRNPC (Entrez ID 3183). Left: unlabelled edges. Right: edge labelled with the transcript ids (or names).

4 Splicing graph bubbles

4.1 Some definitions

The definition of a bubble provided in Sammeth's paper [1] is quite obfuscated. Here we try to give a simpler one. We start by introducing the notion of a valid path between 2 nodes of a splicing graph: a "valid path" between 2 nodes s and t ($s < t$) of a splicing graph is a path followed by at least 1 transcript (see the `txpath` function in previous section for how to extract the transcript paths).

Let's illustrate this by looking at the splicing graph for gene 7597 (Entrez ID). This gene is made of 3 transcripts: `uc001xhg.3`, `uc001xhc.3`, and `uc001xhf.3`. Its splicing graph is shown on figure 6.

Path `{2,11,12,15,16,L}` in that graph is followed by transcript `uc001xhc.3` and thus is valid. However path `{2,11,12,13,14,L}` is not followed by any known transcript and thus is not valid.

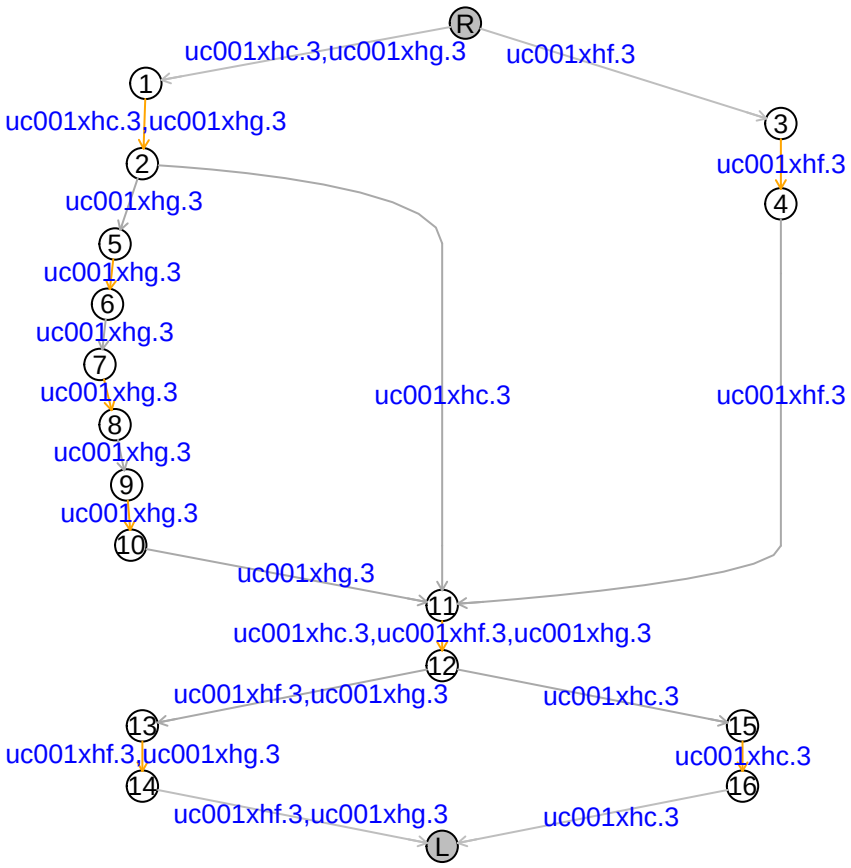

Figure 6: Splicing graph representation of gene 7597 (Entrez ID). Valid paths between nodes `2` and `L` are: `{2,5,6,7,8,9,10,11,12,13,14,L}` (followed by transcript `uc001xhg.3`), and `{2,11,12,15,16,L}` (followed by transcript `uc001xhc.3`). Paths `{2,5,6,7,8,9,10,11,12,15,16,L}` or `{2,11,12,13,14,L}` are not followed by any known transcript and thus are not valid paths.

Now the definition of a bubble: there is a "bubble" between 2 nodes s and t ($s < t$) of a splicing graph if (a) there are at least 2 distinct valid paths between s and t, and (b) no other nodes than s and t are shared by all the valid paths between them. Nodes s and t are said to be the "source" and "sink" of the bubble. The number of distinct valid paths between s and t is the "dimension" of the bubble.

For example, there is a bubble between nodes `R` and `11` of gene 7597 (see figure 6) because the 3 valid paths between the 2 nodes and these paths share those 2 nodes only. The dimension of this bubble is 3. However, there is no bubble between nodes `2` and `L` because the 2 valid paths between the 2 nodes also share nodes `11` and `12`.

The set of all valid paths that form a bubble describes a "complete alternative splicing event" (a.k.a. "complete AS event", or simply "complete event"). The individual valid paths that form the bubble are sometimes called the "variants" of the bubble or event.

4.2 Computing the bubbles of a splicing graph

Like with the `sgedges` and `sgnodes` functions, the bubbles can only be computed for one gene (or graph) at a time. Let's compute the bubbles for gene 7597 (Entrez ID) (represented on figure 6):

```
> bubbles(sg["7597"])
```

```
DataFrame with 3 rows and 7 columns
       source        sink         d                           partitions
  <character> <character> <integer>                       <CharacterList>
1           R          11         3 {uc001xhc.3},{uc001xhg.3},{uc001xhf.3}
2           2          11         2              {uc001xhc.3},{uc001xhg.3}
3          12           L         2    {uc001xhf.3,uc001xhg..,{uc001xhc.3}
                          paths                 AScode            description
                <CharacterList>           <character>            <character>
1 {1,2},{1,2,5,6,7,8,9,10},{3,4} 1[2^,1[2^5-6^7-8^9-1..                     NA
2              {},{5,6,7,8,9,10}         0,1-2^3-4^5-6^           skip 3 exons
3                {13,14},{15,16}              1-2],3-4] 2 alternative last e..
```

The *DataFrame* object returned by `bubbles` has 1 row per bubble in the graph and the following columns:

- `source`, `sink`: The source and sink of the bubble.
- `d`: The dimension of the bubble.
- `partitions`, `paths`: The latter contains the valid paths between the source and sink of the bubble (note that the source and sink nodes are not reported in the paths). The former contains the list of transcripts associated with each of these paths.
- `AScode`, `description`: The AS code (Alternative Splicing code) and its verbose description. More on these in the next subsection.

Bubbles can be nested, i.e., contain (or be contained in) other bubbles of smaller (or higher) dimensions. For example, in gene 7597, the bubble between nodes `R` and `11` (1st bubble in the *DataFrame* object returned by `bubbles`) is of dimension 3 and contains the smaller bubble between nodes `2` and `11` (2nd bubble in the *DataFrame* object returned by `bubbles`) which is of dimension 2.

Bubbles can also overlap without one containing any of the others.

4.3 AS codes

See [1] for how AS codes are generated.

Let's illustrate this by looking at how the AS code for the 2nd bubble in gene 7597 is obtained. In 3 steps:

1. The paths reported for this bubble are `{}` and `{5,6,7,8,9,10}`. All the nodes involved in these paths are re-numbered starting at `1` so nodes `5`, `6`, `7`, `8`, `9`, `10` become `1`, `2`, `3`, `4`, `5`, `6`, respectively.
2. Then, for each path, a string is generated by putting together the new node numbers and node types in that path. By convention, this string is set to `0` if the path is empty. So paths `{}` and `{5,6,7,8,9,10}` generate strings `0` and `1-2^3-4^5-6^`, respectively.
3. Finally, the strings obtained previously are sorted lexicographically and concatenated together separated by commas (`,`). This gives the final AS code: `0,1-2^3-4^5-6^`.

Verbose descriptions (in the English language) of the AS codes are obtained by looking them up through the `ASCODE2DESC` predefined object. `ASCODE2DESC` is a named character vector that contains the verbose descriptions for the AS codes of the 50 most frequent patterns of internal complete events found in Human as reported in Table 1 of Sammeth's paper [1].

```
> codes <- bubbles(sg["7597"])$AScode
> data.frame(AScode=codes, description=ASCODE2DESC[codes], row.names=NULL)
```

```
                       AScode              description
1 1[2^,1[2^5-6^7-8^9-10^,3[4^                    <NA>
2              0,1-2^3-4^5-6^             skip 3 exons
3                   1-2],3-4] 2 alternative last exons
```

```
> codes <- bubbles(sg["10202"])$AScode
> data.frame(AScode=codes, description=ASCODE2DESC[codes], row.names=NULL)
```

```
                                AScode               description
1                            1[2^,3[4^ 2 alternative first exons
2                               1-,2-   2 alternative acceptors
3 0,1-2^3-4^6-7^8-9],1-2^3-5^6-7^8-9]                       <NA>
4                               1^,2^      2 alternative donors
```

`ASCODE2DESC` only contains the 50 code descriptions reported in Table 1 of Sammeth's paper [1]. Therefore, looking up codes through `ASCODE2DESC` will inevitably return an `NA` for some codes. For example, even simple codes like `0,1[2^` or `0,1-2^3-4]` (which mean "skip the first exon" or "skip the last 2 exons", respectively) are not in Sammeth's table, because, as its title indicates, only **internal** complete events are reported in that table (a complete event, or bubble, being considered "internal" when its source and sink nodes are not `R` or `L`, or, equivalently, when its code has no `[` or `]` in it).[1]

[1] Despite its title ("50 Most Frequent Patterns of Internal Complete Events Found in Human"), Table 1 of Sammeth's paper [1] actually contains AS codes that seem to correspond to complete events that are not internal. For example, the 16th AS code in the table is `1*2^,3*4^` and its description is `"2 alternative first exons"`, which indicates that the event is internal. This raises at least 2 questions: why is it in the table? and why does the code contain `*` instead of `[`? A third question could be: why exclude complete events that are not internal in the first place?

4.4 Tabulating all the AS codes for Human chromosome 14

To extract all the bubbles from all the genes on Human chromosome 14 and tabulate their codes, we do the following:

```
> AScode_list <- lapply(seq_along(sg), function(i) bubbles(sg[i])$AScode)
> names(AScode_list) <- names(sg)
> AScode_table <- table(unlist(AScode_list))
> AScode_table <- sort(AScode_table, decreasing=TRUE)
> AScode_summary <- data.frame(AScode=names(AScode_table),
+                              NbOfEvents=as.vector(AScode_table),
+                              Description=ASCODE2DESC[names(AScode_table)])
> nrow(AScode_summary)
```

```
[1] 956
```

```
> head(AScode_summary, n=10)
```

```
        AScode NbOfEvents                                                  Description
1       0,1-2^        454                                                  skip 1 exon
2        1-,2-        146                                    2 alternative acceptors
3        1^,2^         88                                       2 alternative donors
4       0,1^2-         53                                            retain 1 intron
5    1[2^,3[4^         51                                  2 alternative first exons
6       0,1[2^         29                                                       <NA>
7   0,1-2^3-4^         28                                               skip 2 exons
8    1^3-4],2]         23 alternative poly-adenylation site in the retained last intron
9    1[2^4-,3[         18           alternative transcription start in the first intron
10 0,1-2^,3-4^         15                         include 0 or 1 of 2 alternative exons
```

Amongst the 956 distinct complete events we observe on Human chromosome 14, the most frequent one is described as "skip 1 exon". At this point it should be mentioned that not all splicing events have descriptions in the English language since there are a variety of different events.

Another interesting question we can ask now is how many complete events is observed in each gene. To answer this question we first need to count the number of bubbles within each gene.

```
> nb_bubbles_per_gene <- elementNROWS(AScode_list)
```

Below the genes with the most observed splicing events are shown.

```
> head(sort(nb_bubbles_per_gene, decreasing=TRUE))
```

```
  4287  23224 317749  80017   5265  57862
    68     39     27     25     20     20
```

Next we want to see which genes show the highest heterogeneity of splicing events. We check for the number of unique splicing events for each gene.

```
> nb_unique_bubbles_per_gene <- elementNROWS(unique(CharacterList(AScode_list)))
```

Below the genes with the highest heterogeneity in observed splicing events are shown

```
> head(sort(nb_unique_bubbles_per_gene, decreasing=TRUE))
```

```
  4287  23224 317749  80017   3183  57096
    56     29     23     20     18     18
```

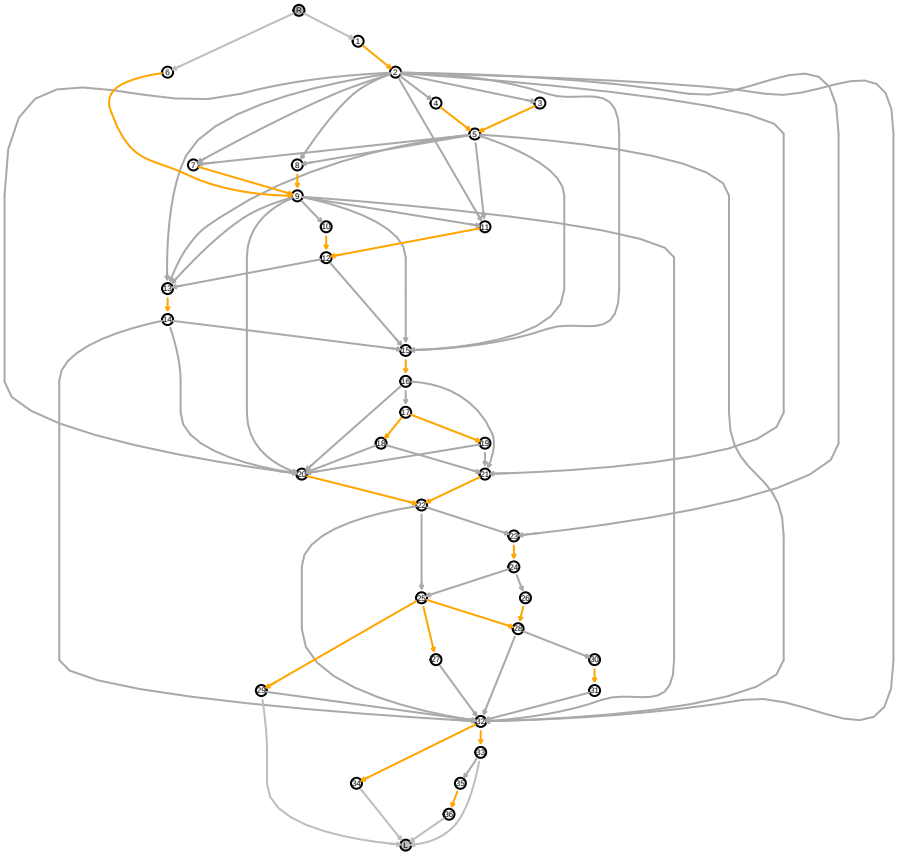

Figure 7: Splicing graph representation of gene 4287 (Entrez ID). This is the gene with the most observed splicing events on Human chromosome 14.

5 Counting reads

In this section, we illustrate how RNA-seq reads can be assigned to the exonic and intronic edges of a *SplicingGraphs* object, and how these assignments can be summarized in a table of counts. The reads we're going to assign are stored as BAM files available in the *RNAseqData.HNRNPC.bam.chr14* package. They've

been aligned to the *hg19* reference genome from UCSC, which is also the reference genome of the gene model in our *SplicingGraphs* object `sg`. As a general rule, the aligned reads and the annotations should always be based on the same reference genome.

Let's start by loading the *RNAseqData.HNRNPC.bam.chr14* package:

```
> library(RNAseqData.HNRNPC.bam.chr14)
> bam_files <- RNAseqData.HNRNPC.bam.chr14_BAMFILES
> names(bam_files)  # the names of the runs
```

```
[1] "ERR127306" "ERR127307" "ERR127308" "ERR127309" "ERR127302" "ERR127303" "ERR127304"
[8] "ERR127305"
```

Another important question about RNA-seq reads is whether they are single- or paired-end. This can be checked with the `quickBamFlagSummary` utility from the *Rsamtools* package:

```
> quickBamFlagSummary(bam_files[1], main.groups.only=TRUE)
```

```
                                group |    nb of |    nb of | mean / max
                                   of |  records |   unique | records per
                              records | in group |   QNAMEs | unique QNAME
All records..................... A |   800484 |   393300 | 2.04 / 10
  o template has single segment.... S |        0 |        0 |   NA / NA
  o template has multiple segments. M |   800484 |   393300 | 2.04 / 10
      - first segment.............. F |   400242 |   393300 | 1.02 / 5
      - last segment............... L |   400242 |   393300 | 1.02 / 5
      - other segment.............. O |        0 |        0 |   NA / NA

Note that (S, M) is a partitioning of A, and (F, L, O) is a partitioning of M.
Indentation reflects this.
```

Doing this for the 8 BAM files confirms that all the reads in the files are paired-end. This means that they should be loaded with the `readGAlignmentPairs` function from the *GenomicAlignments* package. As a quick example, we load the reads located on the first 20 million bases of chromosome 14:

```
> param <- ScanBamParam(which=GRanges("chr14", IRanges(1, 20000000)))
> galp <- readGAlignmentPairs(bam_files[1], param=param)
> length(galp)  # nb of alignment pairs
```

```
[1] 3092
```

```
> galp
```

```
GAlignmentPairs object with 3092 pairs, strandMode=1, and 0 metadata columns:
         seqnames strand   :            ranges  --            ranges
            <Rle>  <Rle>   :         <IRanges>  --         <IRanges>
```

```
    [1]    chr14      +   : 19069583-19069654  -- 24932002-24932073
    [2]    chr14      -   : 19363755-19363826  -- 19363738-19363809
    [3]    chr14      +   : 19369799-19369870  -- 19369828-19369899
    [4]    chr14      +   : 19411097-19411168  -- 19411108-19411179
    [5]    chr14      +   : 19411459-19411530  -- 19411582-19411653
    ...      ...    ... ...               ... ...               ...
 [3088]    chr14      -   : 19988540-19988611  -- 19988439-19988510
 [3089]    chr14      -   : 19988549-19988620  -- 19988436-19988507
 [3090]    chr14      -   : 19988549-19988620  -- 19988443-19988514
 [3091]    chr14      -   : 19988550-19988621  -- 19988443-19988514
 [3092]    chr14      -   : 19988551-19988622  -- 19988439-19988510
 -------
 seqinfo: 93 sequences from an unspecified genome
```

The result is a *GAlignmentPairs* object with 1 element per alignment pair (alignment pair = pair of BAM records). Although paired-end reads could also be loaded with `readGAlignments` (also from the *GenomicRanges* package), they should be loaded with the `readGAlignmentPairs`: this actually pairs the BAM records and returns the pairs in a *GAlignmentPairs* object. Otherwise the records won't be paired and will be returned in a *GAlignments* object.

5.1 Assigning reads to the edges of a *SplicingGraphs* object

We can use the `assignReads` function to assign reads to the the exonic and intronic edges of `sg`. The same read can be assigned to more than one exonic or intronic edge. For example, a junction read with 1 junction (i.e., 1 `N` in its CIGAR) can be assigned to an intron and its 2 flanking exons, and this can happen for one or more transcripts, from the same gene or from different genes.

We're going to loop on the BAM files to assign the reads from 1 run at a time. When we load the files, we want to filter out secondary alignments, reads not passing quality controls, and PCR or optical duplicates. This is done by preparing a *ScanBamParam* object that we'll pass to `readGAlignmentPairs` (see `ScanBamParam` in the *Rsamtools* package for more information about this):

```
> flag0 <- scanBamFlag(isSecondaryAlignment=FALSE,
+                      isNotPassingQualityControls=FALSE,
+                      isDuplicate=FALSE)
> param0 <- ScanBamParam(flag=flag0)
```

Loop:

```
> ## The following loop takes about 7 minutes on a modern laptop/desktop...
> for (i in seq_along(bam_files)) {
+     bam_file <- bam_files[i]
+     cat("Processing run ", names(bam_file), " ... ", sep="")
+     galp <- readGAlignmentPairs(bam_file, use.names=TRUE, param=param0)
+     sg <- assignReads(sg, galp, sample.name=names(bam_file))
+     cat("OK\n")
+ }
```

The assignments to the exonic and intronic edges of a given gene can be retrieved by extracting the edges grouped by transcript for that gene. This is achieved by calling `sgedgesByTranscript` on the subsetted

SplicingGraphs object. `sgedgesByTranscript` is similar to `sgedgesByGene` (introduced in Section 3), except that it groups the edges by transcript instead of by gene. By default the edge-level metadata columns are the same as with `sgedgesByGene` so we use `with.hits.mcols=TRUE` to get the additional metadata columns containing the assignments:

```
> edges_by_tx <- sgedgesByTranscript(sg["3183"], with.hits.mcols=TRUE)
> edge_data <- mcols(unlist(edges_by_tx))
> colnames(edge_data)
```

```
 [1] "from"           "to"             "sgedge_id"      "ex_or_in"       "tx_id"
 [6] "ERR127306.hits" "ERR127307.hits" "ERR127308.hits" "ERR127309.hits" "ERR127302.hits"
[11] "ERR127303.hits" "ERR127304.hits" "ERR127305.hits"
```

There is 1 *hits* column per run (or sample). They appear in the order they were created. Each of them is a *CharacterList* object that contains the reads from that run (or sample) that were assigned to the individual edges. Let's look at the *hits* column for the first run (`ERR127306`):

```
> head(edge_data[ , c("sgedge_id", "ERR127306.hits")])
```

```
DataFrame with 6 rows and 2 columns
       sgedge_id                                                ERR127306.hits
     <character>                                               <CharacterList>
3183    3183:1,2 ERR127306.14143335,ERR127306.23395242,ERR127306.26790344,...
3183    3183:2,5 ERR127306.14143335,ERR127306.23395242,ERR127306.26790344,...
3183    3183:5,6    ERR127306.329023,ERR127306.9471404,ERR127306.11461405,...
3183    3183:6,9    ERR127306.329023,ERR127306.9471404,ERR127306.11461405,...
3183   3183:9,11  ERR127306.6268951,ERR127306.12789886,ERR127306.13671541,...
3183  3183:11,12
```

5.2 How does read assignment work?

[TODO: Complete this subsection.]

For the purpose of explaining how reads from run `ERR127306` were assigned to the edges of gene 3183, we load the reads that are in the region of that gene and contain at least 1 junction:

```
> param <- ScanBamParam(flag=flag0, which=range(unlist(sg[["3183"]])))
> reads <- readGAlignmentPairs(bam_files[1], use.names=TRUE, param=param)
> junction_reads <- reads[njunc(first(reads)) + njunc(last(reads)) != 0L]
```

and we plot the genomic region chr14:21675000-21702000:

```
> plotTranscripts(sg[["3183"]], reads=junction_reads, from=21675000, to=21702000)
```

The resulting plot is shown on figure 8.

Figure 9 is a zoom on genomic region chr14:21698400-21698600 that was obtained with:

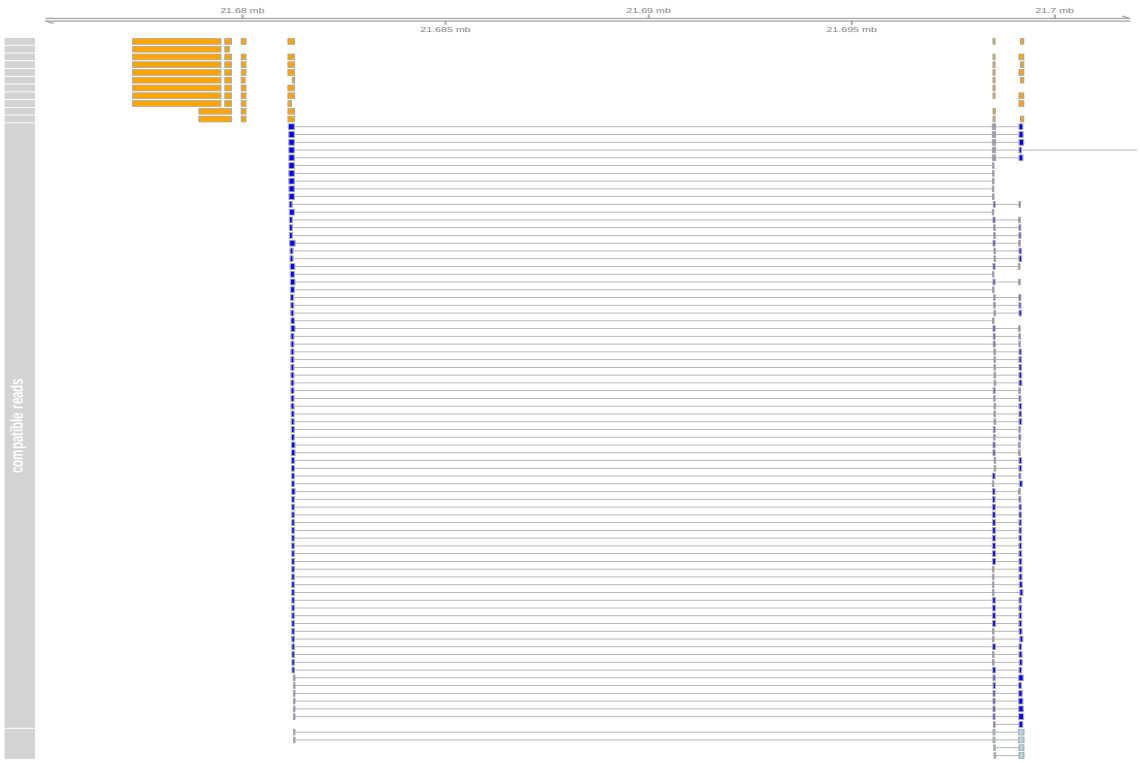

Figure 8: Plot of the genomic region chr14:21675000-21702000 showing the 11 transcripts of gene HNRNPC (Entrez ID 3183) and the reads in that region. Reads that are "compatible" with the splicing of a transcript are plotted in dark blue. Incompatible reads are in light blue.

```
> plotTranscripts(sg[["3183"]], reads=junction_reads, from=21698400, to=21698600)
```

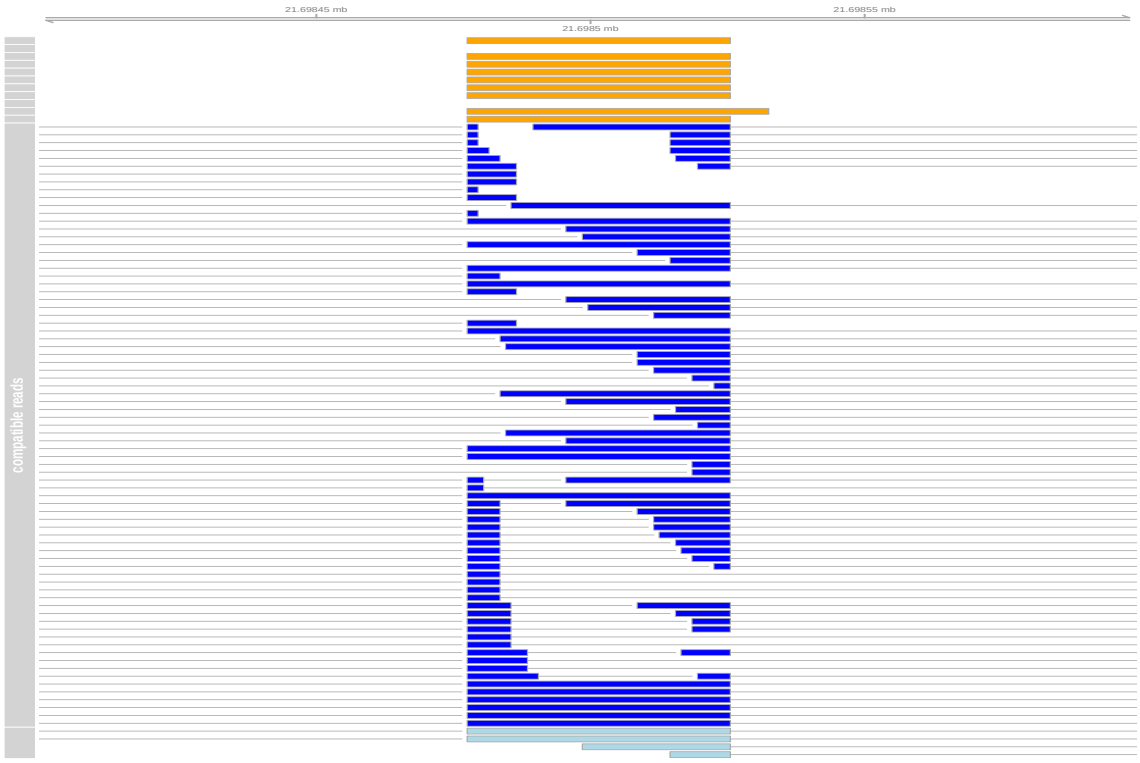

Figure 9: Zoom on genomic region chr14:21698400-21698600 of figure 8.

[TODO: To be continued.]

5.3 Counting and summarizing the assigned reads

We use `countReads` to count the number of reads assigned to each (non-artificial) edge in `sg` for each run:

```
> sg_counts <- countReads(sg)
```

The returned object is a *DataFrame* with one row per unique splicing graph edge and one column of counts per run. Two additional columns contain the splicing graph edge ids and the type of edge (exon or intron):

```
> dim(sg_counts)
```

```
[1] 13483    10
```

```
> head(sg_counts[1:5])
```

```
DataFrame with 6 rows and 5 columns
    sgedge_id ex_or_in ERR127306 ERR127307 ERR127308
  <character> <factor> <integer> <integer> <integer>
1   10001:1,2       ex         7         7         1
2   10001:2,3       in         7         7         1
3   10001:3,4       ex        32        41        29
4   10001:4,5       in        20        37        26
5   10001:5,6       ex        37        66        51
6   10001:6,7       in        23        36        31
```

Total number of hits per run:

```
> sapply(sg_counts[-(1:2)], sum)
```

```
ERR127306 ERR127307 ERR127308 ERR127309 ERR127302 ERR127303 ERR127304 ERR127305
   632444    728648    691039    642086    565818    598047    610521    654658
```

[TODO: Suggest some possible downstream statistical analysis that take the output of countReads() as a starting point.]

6 Session Information

All of the output in this vignette was produced under the following conditions:

```
> sessionInfo()
```

```
R Under development (unstable) (2024-10-21 r87258)
Platform: x86_64-pc-linux-gnu
Running under: Ubuntu 24.04.1 LTS

Matrix products: default
BLAS:   /home/biocbuild/bbs-3.21-bioc/R/lib/libRblas.so
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C               LC_TIME=en_GB
 [4] LC_COLLATE=C               LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C                  LC_ADDRESS=C
[10] LC_TELEPHONE=C             LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

time zone: America/New_York
tzcode source: system (glibc)

attached base packages:
[1] grid      stats4    stats     graphics  grDevices utils     datasets  methods
[9] base

other attached packages:
 [1] RNAseqData.HNRNPC.bam.chr14_0.43.0      SplicingGraphs_1.47.0
 [3] Rgraphviz_2.51.0                        graph_1.85.0
```

```
 [5] GenomicAlignments_1.43.0               Rsamtools_2.23.0
 [7] Biostrings_2.75.0                      XVector_0.47.0
 [9] SummarizedExperiment_1.37.0            MatrixGenerics_1.19.0
[11] matrixStats_1.4.1                      TxDb.Hsapiens.UCSC.hg19.knownGene_3.2.2
[13] GenomicFeatures_1.59.0                 AnnotationDbi_1.69.0
[15] Biobase_2.67.0                         GenomicRanges_1.59.0
[17] GenomeInfoDb_1.43.0                    IRanges_2.41.0
[19] S4Vectors_0.45.0                       BiocGenerics_0.53.0

loaded via a namespace (and not attached):
  [1] DBI_1.2.3                bitops_1.0-9             deldir_2.0-4
  [4] gridExtra_2.3            httr2_1.0.5              biomaRt_2.63.0
  [7] rlang_1.1.4              magrittr_2.0.3           biovizBase_1.55.0
 [10] compiler_4.5.0           RSQLite_2.3.7            png_0.1-8
 [13] vctrs_0.6.5              ProtGenerics_1.39.0      stringr_1.5.1
 [16] pkgconfig_2.0.3          crayon_1.5.3             fastmap_1.2.0
 [19] backports_1.5.0          dbplyr_2.5.0             utf8_1.2.4
 [22] rmarkdown_2.28           UCSC.utils_1.3.0         bit_4.5.0
 [25] xfun_0.48                zlibbioc_1.53.0          cachem_1.1.0
 [28] jsonlite_1.8.9           progress_1.2.3           blob_1.2.4
 [31] DelayedArray_0.33.0      BiocParallel_1.41.0      jpeg_0.1-10
 [34] parallel_4.5.0           prettyunits_1.2.0        cluster_2.1.6
 [37] VariantAnnotation_1.53.0 R6_2.5.1                 stringi_1.8.4
 [40] RColorBrewer_1.1-3       rtracklayer_1.67.0       rpart_4.1.23
 [43] Gviz_1.51.0              Rcpp_1.0.13              knitr_1.48
 [46] base64enc_0.1-3          Matrix_1.7-1             nnet_7.3-19
 [49] igraph_2.1.1             tidyselect_1.2.1         dichromat_2.0-0.1
 [52] rstudioapi_0.17.1        abind_1.4-8              yaml_2.3.10
 [55] codetools_0.2-20         curl_5.2.3               lattice_0.22-6
 [58] tibble_3.2.1             KEGGREST_1.47.0          evaluate_1.0.1
 [61] foreign_0.8-87           BiocFileCache_2.15.0     xml2_1.3.6
 [64] pillar_1.9.0             filelock_1.0.3           checkmate_2.3.2
 [67] generics_0.1.3           RCurl_1.98-1.16          ensembldb_2.31.0
 [70] hms_1.1.3                ggplot2_3.5.1            munsell_0.5.1
 [73] scales_1.3.0             glue_1.8.0               lazyeval_0.2.2
 [76] Hmisc_5.2-0              tools_4.5.0              interp_1.1-6
 [79] BiocIO_1.17.0            data.table_1.16.2        BSgenome_1.75.0
 [82] XML_3.99-0.17            latticeExtra_0.6-30      colorspace_2.1-1
 [85] GenomeInfoDbData_1.2.13  htmlTable_2.4.3          restfulr_0.0.15
 [88] Formula_1.2-5            cli_3.6.3                rappdirs_0.3.3
 [91] fansi_1.0.6              S4Arrays_1.7.0           dplyr_1.1.4
 [94] AnnotationFilter_1.31.0  gtable_0.3.6             digest_0.6.37
 [97] SparseArray_1.7.0        rjson_0.2.23             htmlwidgets_1.6.4
[100] memoise_2.0.1            htmltools_0.5.8.1        lifecycle_1.0.4
[103] httr_1.4.7               bit64_4.5.2
```

References

[1] Michael Sammeth. Complete Alternative Splicing Events Are Bubbles in Splicing Graphs *Computational Biology*, **16**, 1117-1140, 2010.

[2] Kathi Zarnack and Julian Konig and Mojca Tajnik and Inigo Martincorena and Sebastian Eustermann and Isabelle Stevant and Alejandro Reyes and Simon Anders and Nicholas M. Luscombe and Jernej Ule Direct Competition between hnRNP C and U2AF65 Protects the Transcriptome from the Exonization of Alu Elements *Cell*, 2012, Europe PMC 23374342.

[3] LI, Bo and Dewey, Colin. RSEM: accurate transcript quantification from RNA-Seq data with or without a reference genome *BMC Bioinformatics*, **12**(1), 323, 2011.

[4] Roberts, Adam and Pimentel, Harold and Trapnell, Cole and Pachter, Lior. Identification of novel transcripts in annotated genomes using RNA-Seq *BIOINFORMATICS*, 2011

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

[6] Heber, Steffen and Alekseyev, Max and Sze, Sing-Hoi and Tang, Haixu and Pevzner, Pavel A. Splicing graphs and EST assembly problem *Bioinformatics*, **18**(suppl 1), S181-S188, 2002.