

# Genomorientierte Bioinformatik - Report

## ExonSkipping

Malte Weyrich

OCTOBER 2024

---

**Exon Skipping Splicing Events** (*ES-SE*) beschreiben, wie co- oder posttranslational, manche Exons eines Transkripts durch das Spleißosom herausgeschnitten oder übersprungen werden, während in anderen Transkripten des selben Gens, diese weiterhin Teil der mRNA bleiben. Die *ES-SE* lassen sich anhand von **Gene Transfer Format** (*GTF*) files, also Genom Annotations Dateien ablesen und analysieren. Im Folgenden wird ein Programm zur Erkennung von allen *ES-SE* innerhalb eines Genoms anhand seiner Logik, Laufzeit und Ergebnisse analysiert, wobei nur *ES-SE* berücksichtigt werden, die protein-kodierende Transkripte betreffen. Das Programm wurde auf allen verfügbaren *GTF* Dateien in `/mnt/biosoft/praktikum/genprakt/gtfs/` ausgeführt. Der Source Code und alle dazugehörigen Komponenten sind auf [GitHub](#) zu finden.

---

# Contents

<b>1</b>	<b>ES-SE Definition</b>	<b>3</b>
<b>2</b>	<b>Java Programm</b>	<b>3</b>
2.1	Logik . . . . .	3
2.2	Korrektheit . . . . .	6
2.3	Laufzeit . . . . .	7
2.4	Benchmarking . . . . .	10
<b>3</b>	<b>Ergebnisse</b>	<b>11</b>
<b>A</b>	<b>Appendix Section</b>	<b>14</b>

## 1 – ES-SE Definition

In einem Gen kann jedes Transkript jeweils mehrere *ES-SE* haben. Ein *ES-SE* involviert immer jeweils mindestens eine **Splice Variant** (*SV*) und einen **Wild Type** (*WT*). Beide dieser Begriffe beziehen sich auf Transkripte eines Gens *G*. Ein *SV* ist ein Transkript  $T_{SV}$ , welches ein Intron *I* mit Startposition  $I_S$  und Endposition  $I_E$  besitzt, was gleichzeitig bedeutet, dass es in  $T_{SV}$  zwei Exons *A*, *B* gibt, die *I* flankieren. Somit endet *A* bei  $I_S - 1 = A_E$  und *B* startet bei  $I_E + 1 = B_S$ . Zudem ist die Position  $B_{pos} - A_{pos} = 1$ , wobei sich  $A_{pos}$  auf die Position von Exon *A* relativ gesehen zu allen anderen Exons von  $T_{SV}$  bezieht. Ein *WT* wäre nun ein weiteres Transkript  $T_{WT}$  des selben Gens *G*, welches ebenfalls zwei Exons *C*, *D* besitzt mit  $C_{pos} < D_{pos}$ , wobei  $C_E = I_S - 1$  und  $D_S = I_E + 1$ , jedoch gilt für *C*, *D*:  $D_{pos} - C_{pos} > 1$ . Dies bedeutet, dass die Exons von  $T_{WT}$  zwischen *C* und *D* in  $T_{SV}$  herausgespleißt wurden. Es kann pro Event mehrere *SV*'s und *WT*'s geben.

## 2 – Java Programm

### 2.1. Logik

Der Workflow der *JAR* lässt sich in drei Schritte aufteilen:

#### (A) Einlesen der *GTF* Datei und Initialisierung der Datenstruktur

Zum einlesen wird die *GTF* Datei zuerst nach relevanten Zeilen gefiltert, denn für uns sind momentan nur Zeilen relevant, die in der 3. Spalte entweder "*exon*" oder "*CDS*" stehen haben. Hierbei wird vermieden, die Methode *String.split("\t")* zu verwenden. Stattdessen wird in einem *for loop* jedes Zeichen einzeln betrachtet. Dabei werden Zeilen die mit einem "#" anfangen direkt übersprungen. Für alle anderen Zeilen werden die Anzahl der *tabs* gezählt und nach dem zweiten *tab*, werden alle darauf folgenden Zeichen zu einem *String* zusammen konkateniert, bis der dritte *tab* erreicht wurde. Falls der entstandenen  $\text{String} \in \{\text{"exon"}, \text{"CDS"}\}$ , wird die Zeile einer *ArrayList<String>* hinzugefügt, ansonsten wird mit der nächsten Zeile weiter gemacht. Diese Liste enthält am Ende alle relevanten Zeilen. Jede der relevanten Zeilen werden nun mit *String.split("\t")* in ein *String[] mainComponents* geschrieben. Die *attributeColumn* wird aus *mainComponents* extrahiert (auch als ein *String[]* Names *attributes*), indem man

*mainComponents[mainComponents.length - 1]* am ";" splitted.

Mit diesen zwei Komponenten pro Zeile wird als erstes die *gene\_id* abgespeichert und überprüft, ob wir eine neue *gene\_id* erreicht haben. Falls ja, wird ein neues Gen erstellt. Für die darauf folgenden Zeilen wird überprüft, ob wir ein neues Transkript erreicht haben. Neue Transkripte werden in einer *ArrayList<Transkript>* des dazugehörigen Gens abgespeichert. Transkripte wiederum besitzen eine *ArrayList<CodingDnaSequence>* *cdsList* und zwei *HashMap<Integer, CodingDnaSequence>* *cdsStartIndices*, *cdsEndIndices*. Die Transkripte werden mit den dazugehörigen *CodingDnaSequence*'s befüllt, wobei für jede erstellte *CodingDnaSequence* die Start- und Endposition in den jeweiligen *HashMap*'s als Key auf das erstellte Objekt verweisen. Zudem wird mit einer Zählvariable *int cdsCount* die Position der *CodingDnaSequence*'s innerhalb des Transkripts in dem *CodingDnaSequence* Objekt gespeichert.

#### (B) Generieren der ES-SE

Zum generieren der ES-SE werden als erstes für alle in dem Genom abgespeicherten Gene, die dazugehörigen *Introns* errechnet und in einem *HashSet<Introns>* innerhalb des Gens abgespeichert. Dafür werden alle Transkripte eines Gens und deren *CodingDnaSequence*'s angeschaut. Die Introns werden dann mit jeweils zwei *CodingDnaSequence*'s berechnet (bei Genen die sich auf dem "-" Strang befinden, müssen zuerst die *cdsList*'s aller Transkripte invertiert werden und die Positionen der *CodingDnaSequence*'s neu berechnet werden. Das ist später relevant für die Identifikation der WT's):

```
// invert cdsList of transcripts
public void invertTranscripts() {
    for (int i = 0; i < transcripts.size(); i++) {
        Transcript currTranscript = transcripts.get(i);
        currTranscript.reversCdsList();
        // updating pos attribute of each cds
        for (int j = 0; j < currTranscript.getCdsList().size(); j++) {
            currTranscript.getCdsList().get(j).setPos(j);
        }
    }
}
```

```

}

// generating introns
for (Transcript transcript : transcripts) {
    for (int i = 0; i < transcript.getCdsList().size() - 1; i++) {
        int intronStart = transcript.getCdsList().get(i).getEnd() + 1;
        int intronEnd = transcript.getCdsList().get(i + 1).getStart() - 1;
        Intron intron = new Intron(intronStart, intronEnd);
        introns.add(intron);
    }
}
}

```

Anschließend wird für jedes Gen  $G$  über die Intron Liste iteriert. Für jedes Intron  $I$  müssen alle Transkripte von  $G$  nach *CodingDnaSequence*'s  $A, B$  durchsucht werden, die die Bedingung  $A_E + 1 = I_S$  und  $B_S - 1 = I_E$ . Dies kann mit Hilfe der zwei *HashMap<Integer, CodingDnaSequence>* Objekte durchgeführt werden. Zudem wir für jedes Intron  $I$  jeweils 4 leere *HashSet<String>*'s erstellt:

1. *SV\_INTORN*: enthält "*intronStart:intronEnd*" des momentanen Introns  $I$
2. *SV\_PROTS*: enthält die *proteinId* von *CodingDnaSequence*  $A$
3. *WT\_INTORN*: enthält alle "*intronStart:intronEnd*" Koordinaten, die zwischen  $A$  und  $B$  liegen
4. *WT\_PROTS*: enthält die *proteinId*'s von allen *CodingDnaSequence*'s die zwischen  $A$  und  $B$  liegen

Nun gibt es zwei Möglichkeiten:

- i.  $A_E + 1 = I_S$  und  $B_S - 1 = I_E$  und  $B_{pos} - A_{pos} = 1$
- ii.  $A_E + 1 = I_S$  und  $B_S - 1 = I_E$  und  $B_{pos} - A_{pos} > 1$

Falls  $i$  eintrifft, handelt es sich um ein *SV* und es wird die *proteinId* von *CodingDnaSequence*  $A$  in *SV\_PROTS* aufgenommen. Ansonsten werden bei Fall *ii* alle *proteinId*'s der *CodingDnaSequence*'s zwischen  $A$  und  $B$  zu *WT\_PROTS* und alle Introns zwischen  $A$  und  $B$  zu *WT\_INTORN* hinzugefügt. Dabei werden ebenfalls Werte wie *min/max\_skipped\_exon/bases* berechnet:

```

// add all introns of WT to WT_INTRON and all cdsids/prot_ids to WT_prots
int skippedBases = 0;
for (int i = cdsFront.getPos() ; i < cdsBehind.getPos(); i++) {
    int wtIntronStart = cdsList.get(i).getEnd() + 1;
    int wtIntronEnd = cdsList.get(i+1).getStart();
    WT_INTRON.add(wtIntronStart + ":" + wtIntronEnd);

    // like this i add many ids twice but that's fine :)
    WT_PROTS.add(cdsFront.getId());
    WT_PROTS.add(cdsBehind.getId());

    if (i > cdsFront.getPos() && i < cdsBehind.getPos()) {
        // we are in a cds that was skipped
        // → get end - start + 1 = length → add to skipped bases
        skippedBases += cdsList.get(i).getEnd()
                        - cdsList.get(i).getStart() + 1;
    }
}
}

```

Ein *ES-SE* wird nur in die *ArrayList<String> events* aufgenommen, falls es für das momentane Intron *I* mindestens einen *WT* gab.

### (C) Erstellen der *<out>.tsv* Datei

Die *ArrayList<String> events* enthält nun alle *ES-SE* als *String* in bereits korrekter Formatierung. In einem *for loop* wird die Lösung Zeile für Zeile in ein *out.tsv* geschrieben.

## 2.2. Korrektheit

In der Einleseroutine werden alle relevanten Zeilen verarbeitet und das Genom korrekt Initialisiert, sofern die Struktur von dem *GTF* den **offiziellen Konventionen** folgt und die jeweiligen *CodingDnaSequence*'s in korrekter Reihenfolge (je nach *"-"/"+* Strang) vorliegen. Zudem werden, falls es keine *"protein\_id"* für eine gegebene Zeile gibt, nach der *"ccsid"* gesucht und falls es diese

nicht gibt, wird die *"protein\_id"* mit *"NaN"* überschrieben. So werden alle Zeilen, die *"CDS"* in ihrer dritten Spalte stehen haben, genutzt, um das Genom aufzufüllen. Für das Errechnen der *ES-SE* in Schritt (B) gilt folgendes: Alle möglichen Introns in einem Gen werden überprüft und für jedes Intron werden alle Transkripte des jeweiligen Gens auf bei  $I_S - 1$  endende und bei  $I_E + 1$  startende *CodingDnaSequence*'s  $A, B$  abgefragt. Für jeden *SV* oder *WT* Kandidaten wird anschließend geschaut, ob es zwischen  $A$  und  $B$  weitere *CodingDnaSequence*'s gibt und je nach dem ein *ES-SE* entdeckt oder nicht. So ist das Programm unter der Annahme, dass die *GTF* Datei fehlerfrei ist, korrekt.

### 2.3. Laufzeit

Die Laufzeitanalyse wird in die drei Segmente aus 2.1 unterteilt.

#### (A) Einlesen der *GTF* Datei und Initialisierung der Datenstruktur

Für eine *GTF* Datei mit  $m$  Zeilen benötigt die Selektion der relevanten Zeilen schon mal mindestens  $m$  Vergleiche, da jede Zeile überprüft werden muss. Für jede Zeile wird ein Substring ab dem zweiten *Tab* bis zum dritten *Tab* erstellt (außer bei Kommentaren, diese werden übersprungen). Die Anzahl der Vergleiche pro Zeile ist kleiner als die Länge der Zeile (da wir ab dem dritten *Tab* abbrechen) und lässt sich als  $a < m.length$  beschreiben. Also

$$\mathcal{O}(m \cdot a) \implies \mathcal{O}(m) \quad (1)$$

,da  $a$  eine Konstante ist.

Bei einer *GTF* Datei mit  $m$  validen Zeilen (d.h. jede Zeile hat entweder einen *"exon"* oder *"CDS"* Eintrag) bleiben nach dem Filtern  $m$  Zeilen übrig. Für jede dieser Zeilen muss:

- i. Die Zeile am *Tab* geteilt werden:

$$\mathcal{O}(n) \quad (2)$$

, wobei  $n$  die Länge der Zeile ist.

- ii. Die letzte Komponente aus i. am ; geteilt werden: lässt sich ebenfalls mit

$$\mathcal{O}(n) \quad (3)$$

von oben beschränken.

- iii. Die *gene\_id* aus den Attributen aus ii. mit *String parseAttributes(String[] attributeEntries, String attributeName)* abfragen, also:

$$\mathcal{O}(e \cdot e_l) \quad (4)$$

, wobei  $e$  die Länge des *attributeEntries* Arrays ist und  $e_l$  die Länge des längsten Eintrags in *attributeEntries* ist, da wir im Worst Case über alle Einträge in *attributeEntries* iterieren ( $e$ ) und für jeweils jeden Eintrag mindesten vier String Operationen (*trim()*, *indexOf()*, zwei mal *substring()* und eine Vergleichsoperation (*equals()*) aufrufen, welche maximal  $e_l$  viele Operationen benötigen. Also  $\mathcal{O}(e \cdot 5 \cdot e_l) = \mathcal{O}(e \cdot e_l)$ . Für jedes weitere Vorkommen von *parseAttributes()* wird diese Komplexität angenommen.

- iv. Ein neues Gen initialisiert werden und der *gene\_name* abgefragt werden (falls ein neues Gen erreicht wurde), also:

$$\mathcal{O}(1 + e \cdot e_l) \quad (5)$$

- v. Die *transcript\_id* abgefragt werden, also

$$\mathcal{O}(e \cdot e_l) \quad (6)$$

- vi. Falls es sich um einen "CDS" Eintrag handelt, muss die *protein\_id* abgefragt werden, also:

$$\mathcal{O}(e \cdot e_l) \quad (7)$$

und in Konstanter Zeit ggf. neue Objekte erstellt, oder auf bereits existierende Objekte zugreifen, um ein neues *CodingDnaSequence* Objekt zu erstellen.

Insgesamt hat die Einleseroutine also eine Komplexität von

$$(A) \quad \mathcal{O}(m \cdot 2 \cdot n \cdot 4 \cdot (e \cdot e_l)) = \mathcal{O}(m \cdot n \cdot e \cdot e_l) \in \mathcal{O}(m^2).$$

,da  $m > n > e_l \geq e$  und somit  $\mathcal{O}(m^2)$  eine valide obere Schranke darstellt.

## (B) Generieren der ES-SE



Sei  $g$  die Anzahl an Genen in unserem Genom. Da wir vom Worst Case ausgehen sagen wir, dass sich jedes Gen auf dem "-" Strang befindet. So muss zuerst in jedem Transkript jedes Gens die *cdsList* invertiert werden. Dies geschieht in:

$$\mathcal{O}(g \cdot g_t \cdot 2 \cdot t_c) = \mathcal{O}(g \cdot g_t \cdot t_c) \quad (8)$$

, wobei  $g_t$  die größte Anzahl an Transkripten von  $g$  ist und  $t_c$  die längste *cdsList* eines Transkripts ist. Die Konstante 2 kommt zustande, da zuerst die *cdsList* mit *Collections.reverse()* umgekehrt wird ( $\mathcal{O}(g_t)$ ) und dann nochmals in  $\mathcal{O}(g_t)$  durchlaufen wird, um die in den *CodingDnaSequence* Objekten gespeicherte Position anzupassen. Dann werden für jedes Gen  $g$  die Introns generiert. Dies geschieht ebenfalls in

$$\mathcal{O}(g \cdot g_t \cdot t_c) \quad (9)$$

Die *ES-SE* werden in der *getEvents()* Methode berechnet. Dafür müssen für alle Gene alle Introns und alle Transkripte überprüft werden. Also schon mal  $\mathcal{O}(g \cdot g_i \cdot g_t)$ . Hier ist  $g_i$  die größte Anzahl an Introns von allen Genen und  $g_t$  wieder die größte Anzahl an Transkripten aller Gene. Alle anderen Operationen sind konstant in ihrer Komplexität, da bei ihnen lediglich bereits existente Werte in *HashMaps* oder Objekten abgefragt werden. Nur falls ein *WT* entdeckt wird, wird in einem *for loop* über die *CodingDnaSequence*'s zwischen  $C$  und  $D$  (siehe 1. ES-SE Definition) iteriert. Sei  $mSE$  (= *maxSkippedExons*) also die von allen *ES-SE* eines Genoms maximale Anzahl an übersprungenen Exons, so wäre die gesamte Komplexität von II:

$$(B) \quad \mathcal{O}(2 \cdot (g \cdot g_t \cdot t_c) + g \cdot g_i \cdot g_t \cdot mSE) = \mathcal{O}(\underbrace{g \cdot g_t \cdot t_c}_{\substack{\text{invertTranscripts() \\ \& generateIntrons()}} + \underbrace{g \cdot g_i \cdot g_t \cdot mSE}_{\text{getEvents()}}) \quad (10)$$

### (C) Erstellen der <out>.tsv Datei

Hat eine Komplexität von

$$(C) \quad \mathcal{O}(E) \quad (11)$$

, wenn  $E$  die Menge aller *ES-SE* ist.

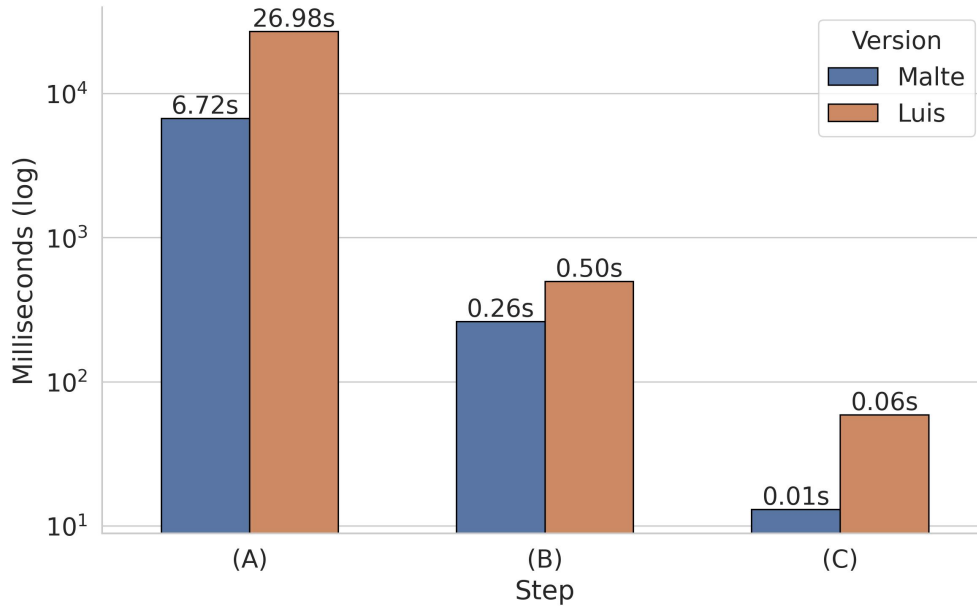
Zusammenfassend also eine Gesamtkomplexität von:

$$(A) + (B) + (C) = \mathcal{O}(m^2 + g \cdot g_t \cdot t_c + g \cdot g_i \cdot g_t \cdot mSE + E) \in \mathcal{O}(m^2) \quad (12)$$

, da  $m^2 > g \cdot g_t \cdot t_c$  und  $m^2 > g \cdot g_i \cdot g_t \cdot mSE + E$ . Das bedeutet, dass die Kosten der Gesamtoperation im Wesentlichen durch das Einlesen und Strukturieren der Daten (Teil A) dominiert werden, wenn man davon ausgeht, dass  $m$  in der Praxis größer als  $g$ ,  $g_t$ ,  $t_c$ ,  $g_i$  und  $mSE$  ist.

#### 2.4. Benchmarking

Für das Benchmarking wird jeweils `/mnt/biosoft/praktikum/genprakt/gtfs/Homo_sapiens.GRCh38.86.gtf` verwendet, da sie die größte *GTF* Datei mit *1.4GB* ist.



**Figure 1 – Methoden Durchschnittslaufzeit der Schritte A, B, C in ms nach 30 facher Ausführung auf Hardware: AMD Ryzen 7 PRO 4750U with Radeon Graphics (16) @ 1.700GHz**

In 1 ist eindeutig zu sehen, wie das Einlesen und Initialisieren der Datenstruktur aus Schritt (A), die Dominante Komponente mit *6721ms* bildet, während die Generierung der *ES-SE* lediglich *262ms* benötigt. Für die Memory Allocations wurde der in *IntelliJ* zur Verfügung gestellter *Profiler* verwendet. Der Schritt (A) ist auch hier dominant und beansprucht insgesamt *10.33GB* an Speicher. Von diesen *10.33GB* werden alleine *6.72GB* (65.18%) für die Methode *String.split()*

benötigt. Für die Berechnung der *ES-SE* werden lediglich 288.11MB in Anspruch genommen.

### 3 — Ergebnisse

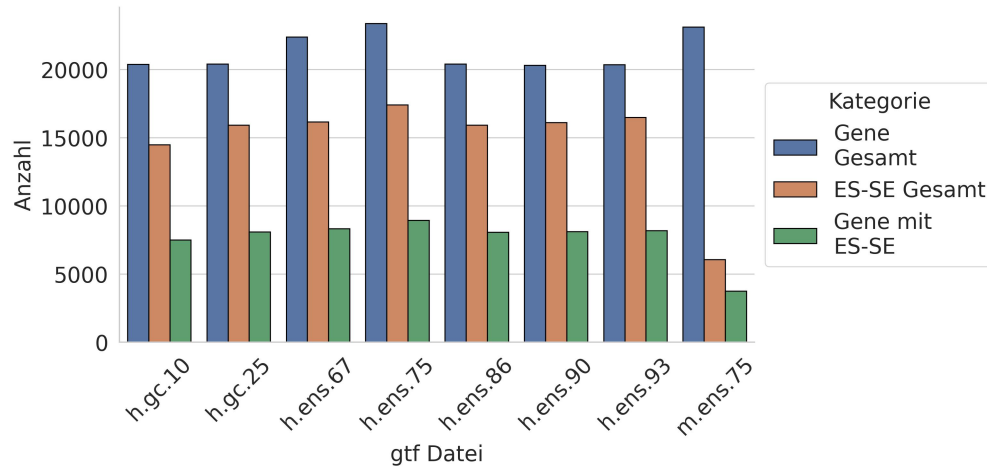
Für die Analyse wurde die *GTF* Datei `/mnt/biosoft/praktikum/genprakt/gtfs/Saccharomyces_cerevisiae.R64-1-1.75.gtf` ausgelassen, da es hier zu keinen *ES-SE* gekommen ist. Die Ursache dafür ist wahrscheinlich, dass es, obwohl es in der Hefe auch zu Splicing kommt, in der gegebenen *GTF* Datei keine protein-kodierenden Transkripte mit Splicing gab, oder allgemein keine *ES-SE* aufgetreten sind. Zusätzlich werden die *GTF* Dateien und die dazugehörigen Ergebnisse mit den folgenden IDs bezeichnet:

**Table 1 — Liste der verwendeten *GTF* Dateien**

ID	<i>GTF</i> Datei
h.ens.67	Homo_sapiens.GRCh37.67.gtf
h.ens.75	Homo_sapiens.GRCh37.75.gtf
h.ens.86	Homo_sapiens.GRCh38.86.gtf
h.ens.90	Homo_sapiens.GRCh38.90.gtf
h.ens.93	Homo_sapiens.GRCh38.93.gtf
m.ens.75	Mus_musculus.GRCm38.75.gtf
h.gc.10	encode.v10.annotation.gtf
h.gc.25	encode.v25.annotation.gtf

Als erstes wird die Anzahl an aller protein-kodierenden Gene einer *GTF* mit der Anzahl an Genen mit *ES-SE* und der Gesamtanzahl an *ES-SE* verglichen:

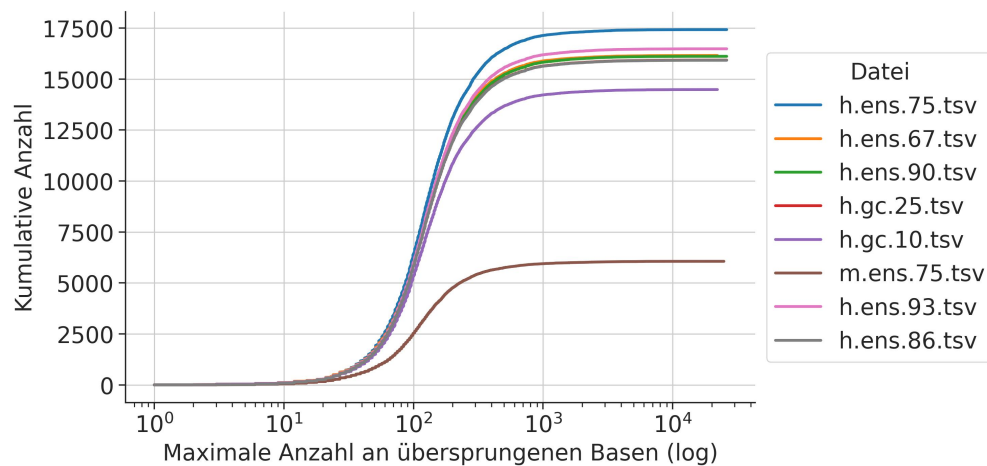
In Abbildung 2 sieht man, wie die Anzahl an Genen in den verschiedenen *GTF* Dateien des Menschen in einem Intervall von [20320; 23393] variieren. Dies liegt daran, dass die *GTF* Dateien jeweils von unterschiedlichen Assemblies und Annotations Versionen stammen, welche beide einen starken Einfluss auf die resultierende *GTF* haben (und somit auch auf das Ergebnis der *JAR*). Wie zu erwarten, hat bei den zum Menschen zugehörigen *GTF* Dateien, die mit den meisten Genen auch die meisten *ES-SE*. Die Dateien "*h.gc.25*", "*h.ens.67*", "*h.ens.86*", "*h.ens.90*", "*h.ens.93*" haben alle eine sehr ähnliche Verteilung der "*ES-SE Gesamt*" und "*Gene mit ES-SE*" Kategorie, obwohl die *GTF* Datei von "*h.ens.67*" mehr Gene beinhaltet, als die drei anderen *GTF* Dateien. "*h.gc.10*" scheint am wenigsten *Gene mit ES-SE* und "*ES-SE Gesamt*" zu besitzen. Bei der Maus wiederum gibt es



**Figure 2 – Vergleich zwischen Gene Gesamt, Gene mit *ES-SE* und *ES-SE* Gesamt pro *GTF***

vergleichsweise wenige *ES-SE*, obwohl es insgesamt fast genau so viele protein-kodierende Gene gibt (23119) wie in "*h.ens.75*" (23393).

Unter Einbezug der Abbildungen 3 und 4, sind die selben vier Trends zu beobachten, die sich in Abbildung 2 bereits andeuten:

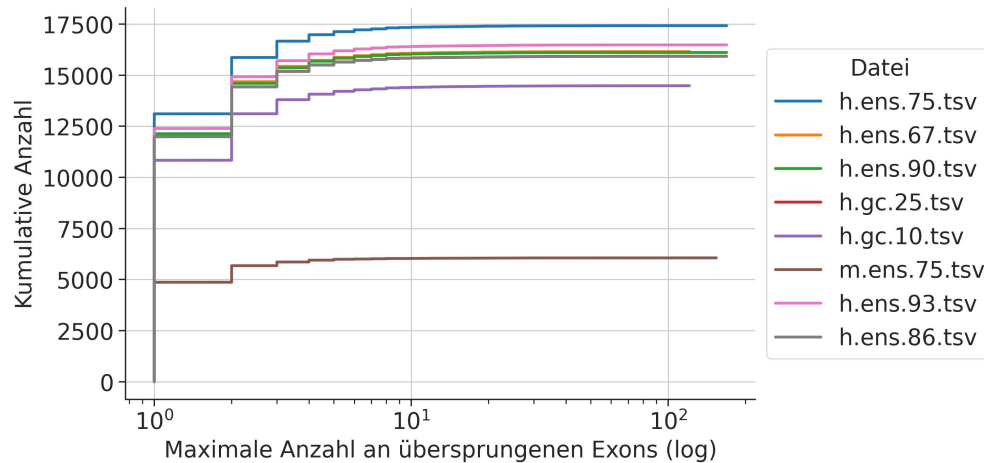


**Figure 3 – Kumulative Verteilung der übersprungenen Basen pro *GTF***

Abbildung 3 zeigt die kumulative Verteilung der übersprungenen Basen pro *GTF*. Alle Kurven zeigen einen charakteristischen S-förmigen Verlauf, was auf ein ähnliches grundlegendes Muster im *ES-SE* Verhalten hindeutet. Es bilden sich hauptsächlich zwei Plateaus aus: Ein höheres Plateau bei etwa 15.000-17.500 übersprungenen Basen für die vom Mensch stammenden *GTF* Dateien und ein niedrigeres Plateau bei etwa 6.000 übersprungenen Basen für die Maus. Die zwei Ausreißer

("h.ens.75" und "h.gc.10") des höheren Plateaus sind wieder auf die jeweils größte und kleinste Anzahl an Genen mit *ES-SE* innerhalb der Humanen *GTF* Dateien zurückzuführen.

Der steilste Anstieg der Kurven erfolgt im Bereich zwischen 100 und 1.000 übersprungenen Basen, was darauf hindeutet, dass die meisten *ES-SE* in diesem Größenbereich stattfinden. Die logarithmische Skalierung der x-Achse verdeutlicht, dass die *ES-SE* über mehrere Größenordnungen hinweg auftreten, von einzelnen Basen bis hin zu mehreren tausend Basen.



**Figure 4 – Kumulative Verteilung der übersprungenen Exons pro *GTF***

Abbildung 4 zeigt die kumulative Verteilung der übersprungenen Exons pro *GTF*. Auch hier zeigen sich die 4 Trends aus 3 und 2. Die meisten *ES-SE* betreffen lediglich ein oder zwei Exons und werden mit zunehmender Anzahl an übersprungenen Exons immer weniger. Diese Verteilung unterstreicht die biologische Relevanz von *Single-Exon-Skipping* als häufigstem Mechanismus im alternativen Spleißen und zeigt gleichzeitig, dass komplexere *ES-SE* mit mehreren Exons zwar vorkommen, aber deutlich seltener sind.

**Figure 5 – Übersprungene Basen**

<i>gene_id</i>	<i>skipped_bases</i>
ENSG00000155657	26106
ENSG00000155657.25	26106
<b>ENSMUSG000000051747</b>	24843
ENSG00000283186	22134
ENSG00000155657.16	22134
ENSG00000283186.1	22134
ENSG00000145113	12875
ENSG00000145113.16	12875
ENSG00000145113.21	12875
ENSG00000164199	12530

**Figure 6 – Übersprungene Exons**

<i>gene_id</i>	<i>skipped_exons</i>
ENSG00000155657	169
ENSG00000155657.25	169
<b>ENSMUSG000000051747</b>	154
ENSG00000283186	121
ENSG00000155657.16	121
ENSG00000283186.1	121
ENSG00000203832.5	78
ENSG00000187240	70
ENSG00000271425	70
ENSG00000187240.8	70

## A – Appendix Section

hm

Text goes here