Genomorientierte Bioinformatik - Report

ExonSkipping

Malte Weyrich

November 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*) Dateien, 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

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} übersprungen 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 String $\in$ {"exon", "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 erst jetzt 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 *";"* teilt.

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*. Für jede erstellte *CodingDnaSequence* eines Transkripts wird ein neuer Eintrag in den zwei *HashMap*'s gemacht, wobei jeweils eine der Koordinaten als Schlüssel dient und auf das so eben erstellte Objekt abbildet. Zudem wird mit einer Zählvariable *int cdsCount* die Position der *CodingDnaSequence*'s innerhalb des Transkripts in dem *CodingDnaSequence* Objekt gespeichert. Wird ein neues Transkript erreicht, wird der *cdsCount* wieder zurückgesetzt.

(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. 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 n 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 $a < n.length$ beschreiben. Also

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

,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 m Zeilen muss**:

i. Die Zeile am *Tab* geteilt werden:

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

Michael 2016 , wobei n die Länge der Zeile ist.

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

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

von oben beschränken.

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

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

, 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:

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

v. Die *transcript_id* abgefragt werden, also

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

vi. Falls es sich um einen *"CDS"* Eintrag handelt, muss die *protein_id* abgefragt werden, also:

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

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

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

,da $m > n > e_l >= e$ 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:

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

, 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 *Cod-*

ingDnaSequence Objekten gespeicherte Position anzupassen. Dann werden für jedes Gen g die Introns generiert. Dies geschieht ebenfalls in

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

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 (B):

$$(B) \quad \mathcal{O}(\overbrace{2 \cdot (g \cdot g_t \cdot t_c)}^{\text{(8) \& (9)}} + \underbrace{g \cdot g_i \cdot g_t \cdot mSE}_{\text{getEvents()}}) = \mathcal{O}(g \cdot g_t \cdot t_c + g \cdot g_i \cdot g_t \cdot mSE) \tag{10}$$

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

Hat eine Komplexität von

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

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

Zusammenfassend also eine Gesamtkomplexität von:

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

, 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.

2.3. 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 *"ccdsid"* 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.

2.4. 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. Die zwei *JARs* (M und L) aus unserer Gruppe wurden jeweils 30 Mal ausgeführt und davon dann ein Durchschnitt errechnet.

In Abbildung 1 ist eindeutig zu sehen, wie das Einlesen und Initialisieren der Datenstruktur aus Schritt (A), die Dominante Komponente beider *JARs*, mit $6721ms$ und $26977ms$ bildet, während die Generierung der *ES-SE* lediglich $262ms$ und $498ms$ benötigt. Für die *Memory Allocations* wurde der in *IntelliJ* zur Verfügung gestellter *Profiler* verwendet. Wichtig ist hierbei der Unterschied zwischen *RAM* und *Memory Allocations*: Der *Profiler* erfasst, wie viel Speicher ein Programm insgesamt anfordert, auch wenn ein Großteil davon später wieder durch den *Garbage Collector* freigegeben wird. Der *Profiler* gibt also eher eine Obergrenze des Speicherverbrauchs an, nicht den exakten *RAM*-Verbrauch. Der Schritt (A) ist auch hier dominant und fordert in *JAR* M insgesamt $10.33GB$ an Speicher an. Von diesen $10.33GB$ werden alleine $6.72GB$ von der Methode *String.split()* gefordert. *JAR* L hingegen benötigt $65.32GB$ an Memory Allocations, wobei der Hauptteil dieses Volumens (ca. $60GB$) für die Methode *String.replaceAll()* aufgewendet wird. Für die Berechnung der *ES-SE* werden lediglich $288.11MB$ (M) und $296.91MB$ (L) in Anspruch

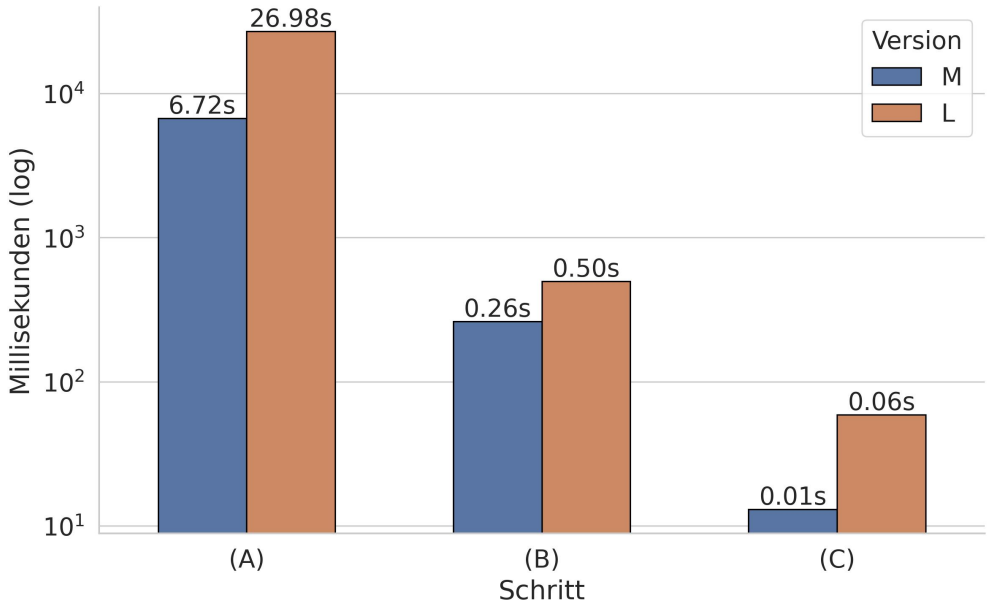

Figure 1 — Methoden Durchschnittslaufzeit der Schritte A, B, C in ms nach 30 facher Ausführung auf Hardware: *AMD Ryzen 7 PRO 4750U with Radeon Graphics (16) @ 1.700GHz*

genommen. Die starken Diskrepanzen in Schritt (A) liegen daran, dass die *JAR* L alle Zeilen mit dem *Parser* einließt und von diesen jedes mal die *attributes* vollständig in einer Datenstruktur abspeichert. Die Methode, die die *attributes* in L verarbeitet, ruft für jedes Attribut-Paar (*key*, *val*), die Methode *String.replaceAll("\"")* auf, welche teuer in Laufzeit und *Memory Allocations* ist. Da es in */mnt/biosoft/praktikum/genprakt/gtfs/Homo_sapiens.GRCh38.86.gtf* insgesamt 2.575.498 Einträge gibt und pro Entrag jeweils mehrere Attribute, wird *String.replaceAll()* extrem häufig aufgerufen. Zusätzlich werden die Attibute in *HashMap*'s gespeichert, was ein zusätzlicher Faktor ist. In *JAR* M werden die Zeilen wie in 2.1 beschreiben nur dann tatsächlich bearbeitet, wenn sie einen *"exon"* oder *"CDS"* Eintrag beinhalten, zudem werden die *attributes* effizienter verarbeitet. Somit ist der *Parser* aus M zwar schneller, jedoch weniger versatil für andere Probleminstanzen.

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 zum Spleißen kommt, in der gegebenen *GTF*-Datei keine

protein-kodierenden Transkripte mit Splicing gab. In der Hefe gibt es insgesamt sehr wenige Introns (ca. 300), verglichen mit (> 140.000) in dem Menschen (Juneau et al. 2007, p. 1525), was die Abwesenheit von *ES-SE* in *Saccharomyces_cerevisiae.R64-1-1.75.gtf* zusätzlich erklärt. Die *GTF*-Dateien und die dazugehörigen Ergebnisse werden ab jetzt mit den folgenden IDs aus Tabelle 1 bezeichnet:

Table 1 — Liste der verwendeten *GTF*-Dateien

ID	*GTF*-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	gencode.v10.annotation.gtf
h.gc.25	gencode.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:

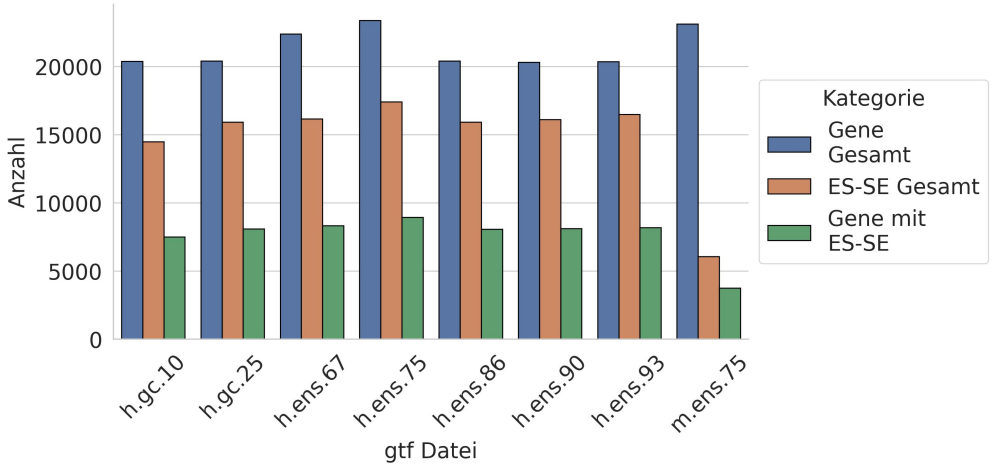

Figure 2 — Vergleich zwischen Gene Gesamt, Gene mit *ES-SE* und *ES-SE* Gesamt pro *GTF*

In Abbildung 2 sieht man, wie die Anzahl an Genen in den verschiedenen *GTF*-Dateien des Menschen in einem Intervall von $[20.320; 23.393]$ 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 *vier 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 vergleichsweise wenige *ES-SE*, obwohl es insgesamt fast genau so viele protein-kodierende Gene gibt (23.119) wie in *"h.ens.75"* (23.393).

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

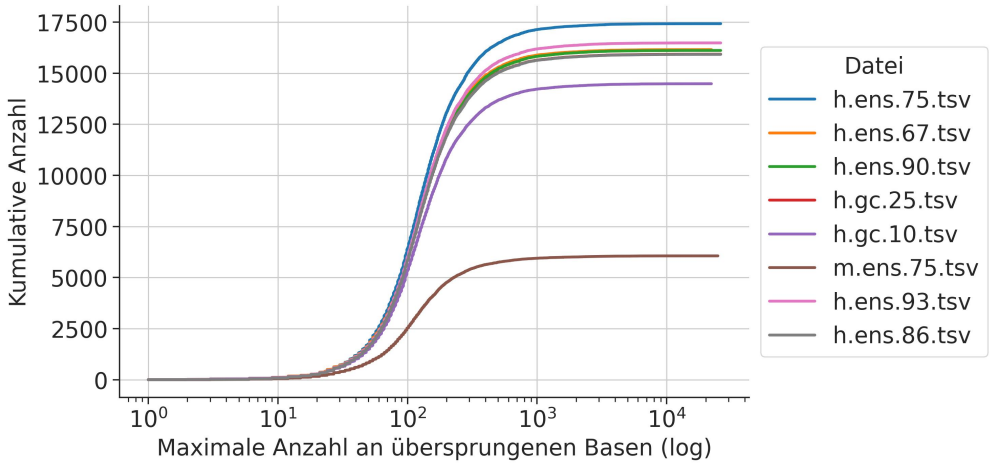

Figure 3 — Kumulative Verteilung der übersprungenen Basen pro *GTF*

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.

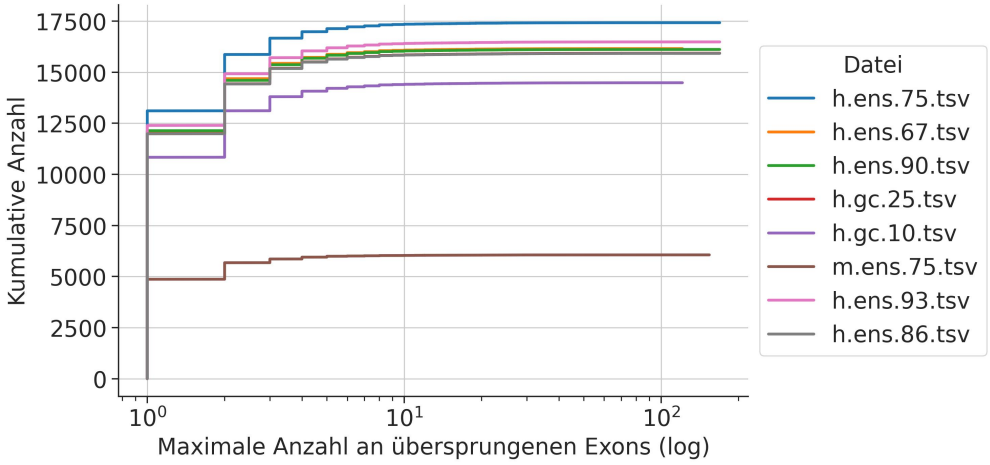

Figure 4 — Kumulative Verteilung der übersprungenen Exons pro *GTF*

Abbildung 4 zeigt die kumulative Verteilung der übersprungenen Exons pro *GTF*. 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.

Symbol	***ID***	***Basen***	***Symbol***	***ID***	***Exons***
TTN	ENSG00000155657	26.106	TTN	ENSG00000155657	169
Ttn	**ENSMUSG00000051747**	24.843	**Ttn**	**ENSMUSG00000051747**	154
TTN	ENSG00000283186	22.134	TTN	ENSG00000283186	121
MUC4	ENSG00000145113	12.875	NBPF20	ENSG00000203832.5	78
ADGRV1	ENSG00000164199	12.530	DYNC2H1	ENSG00000187240	70
DYNC2H1	ENSG00000187240	10.182	NBPF10	ENSG00000271425	70
Fsip2	**ENSMUSG00000075249**	9.659	NBPF10	ENSG00000163386	60
NBPF20	ENSG00000203832	9.573	ADGRV1	ENSG00000164199	59
FSIP2	ENSG00000188738	9.437	NBPF20	ENSG00000162825	56
XIRP2	ENSG00000163092	9.379	NBPF12	ENSG00000186275	52

Figure 5 — Rangliste an Genen mit höchster Anzahl an übersprungen Basen und Exons.

Die Tabellen in Abbildung 5 zeigen die Top-10 Gene basierend auf der Anzahl ihrer Basen (links) und Exons (rechts). Bei Genen mit identischen Transkripten, die sich nur durch Suffix-Annotationen aus verschiedenen GTF-Dateien unterscheiden (z.B. *ENSG00000155657* und

ENSG00000155657.25), wurde nur ein Repräsentant beibehalten, da diese Duplikate biologisch dasselbe Gen repräsentieren und sich lediglich in ihrer Versions-/Quellenkennzeichnung unterscheiden. Die Ränge in den Tabellen werden vor allem von Genen aus dem Menschen eingenommen, lediglich zwei Gene der **Maus** (***"Ttn"*** und ***Fsip2***) konkurrieren mit den anderen Genen der Rangliste. Dabei ist *"TTN"* in beiden Kategorien auf dem ersten Platz mit über 26000 übersprungenen Basen und fast 170 übersprungenen Exons. *"TTN"* und ***"Ttn"*** sind eng verwandte Gene, welche beide für das Protein *Titin* in den jeweiligen Organismen verantwortlich sind. *Titin* wiederum hat eine sehr wichtige Rolle in der Muskelkontraktion und dient zur Stabilisierung und Flexibilität der Sarkomere (UniProt Consortium 2024b). Zudem sind *"TTN"* und ***"Ttn"*** jeweils mit 34.350 (UniProt Consortium 2024c) und 35.213 (UniProt Consortium 2024a) extrem lange Proteine, was die hohe Position in der Rangliste erklärt. Ebenfalls aus der Tabelle hervorgehend ist, dass eine hohe Anzahl an übersprungenen Exons nicht unbedingt mit der Anzahl an übersprungenen Basen zusammenhängen muss. Während das Gen *"MUC4"* mit 12.875 Basen auf Rang vier der linken Tabelle steht, fehlt in der rechten Tabelle von ihm jede Spur. In den Tabellen ist auch auffällig, dass es oft das gleiche ***Gen Symbol*** mit unterschiedlicher ***ID*** und Werten gibt. Zum Beispiel ist *"TTN"* in beiden Tabellen jeweils zwei mal mit unterschiedlichen Werten gelistet. Diese Unterschiede entstehen durch die Verwendung von sechs verschiedenen *GTF*-Dateien für das menschliche Genom, die unterschiedliche Versionen und Qualitätsstufen der Genom Annotation repräsentieren. So kann ein Gen wie *"TTN"* in einer neueren oder detaillierteren *GTF*-Datei mehr annotierte Exons oder eine präzisere Basenzahl aufweisen als in einer älteren oder weniger umfassenden Annotation.

References

Juneau, Kara, Curtis Palm, Molly Miranda, and Ronald W. Davis. 2007. "High-density yeast-tiling array reveals previously undiscovered introns and extensive regulation of meiotic splicing." *Proceedings of the National Academy of Sciences* 104 (5): 1522–1527. https://doi.org/10.1073/pnas.0610354104. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.0610354104. https://www.pnas.org/doi/abs/10.1073/pnas.0610354104. (Cited on page 12).

Michael. 2016. *What's the complexity of Java's String.split function?* Software Engineering Stack Exchange. Accessed on: 2024-02-11. https://softwareengineering.stackexchange.com/questions/331909/whats-the-complexity-of-javas-string-split-function. (Cited on page 7).

UniProt Consortium. 2024a. *UniProtKB - A2ASS6 (TITIN_MOUSE) - Sequence & Isofroms.* [Accessed 01-Nov-2024]. https://www.uniprot.org/uniprotkb/A2ASS6/entry#sequences. (Cited on page 15).

———. 2024b. *UniProtKB - Q8WZ42 (TITIN_HUMAN).* [Accessed 01-Nov-2024]. https://www.uniprot.org/uniprotkb/Q8WZ42/entry. (Cited on page 15).

———. 2024c. *UniProtKB - Q8WZ42 (TITIN_HUMAN) - Sequence & Isofroms.* [Accessed 01-Nov-2024]. https://www.uniprot.org/uniprotkb/Q8WZ42/entry#sequences. (Cited on page 15).