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.

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} 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 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 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 *"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.

2.3. 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

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

,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 Zeilen muss:

i. Die Zeile am *Tab* geteilt werden:

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

, 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

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

,da $m > n > e_l >= e$ 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 *CodingDnaSequence* Objekten gespeicherte Position anzupassen. Dann werden für jedes Gen g die Introns generiert. Dies geschieht ebenfalls in

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

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 II:

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

(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.

2.4. 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.

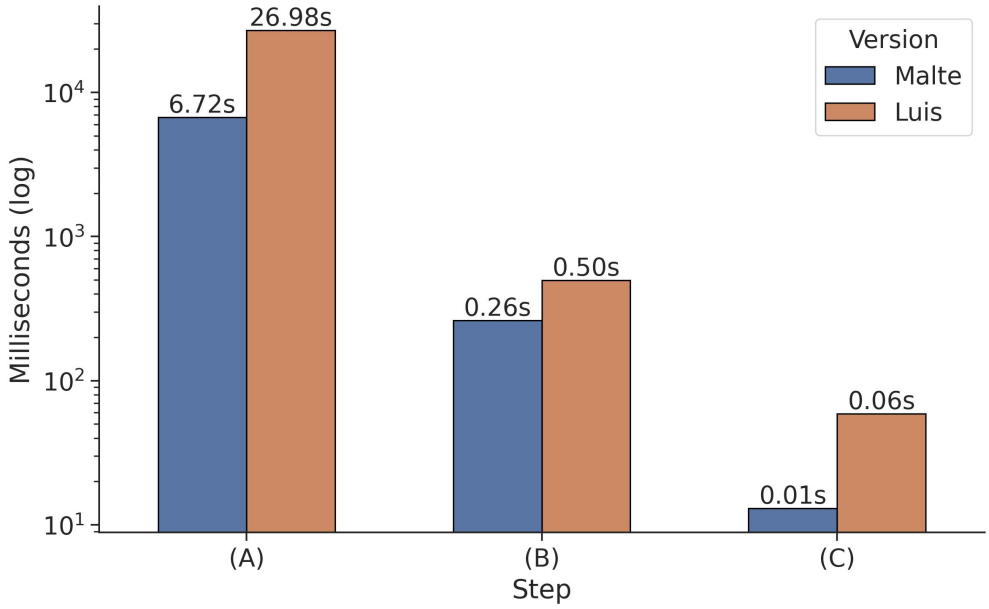

Figure 1 — Methoden Durchschnittslaufzeit der Schritte A, B, C nach Version in ms nach 30 facher Ausführung auf Hardware: *AMD Ryzen 7 PRO 4750U with Radeon Graphics (16) @ 1.700GHz*

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

A — Appendix Section

hm

Text goes here