INSEGT

Ein Programm für annotationsbasierte Expressionanalyse von RNA-Seq Daten

Sabrina Krakau

Bachelorarbeit
22. Oktober 2009
Freie Universität Berlin
Bachelor of Sience in Bioinformatik

Betreuer:

Prof. Dr. Knut Reinert
Freie Universität Berlin
Algorithmische Bioinformatik

Dr. Stefan Haas
Max-Planck-Institut für Molekulare Genetik

Marcel Schulz
Max-Planck-Institut für Molekulare Genetik

Zusammenfassung

Neue Sequenzierverfahren ermöglichen die Erzeugung von Millionen von kurzen Sequenzfragmenten von expremierten RNAs in einer Zelle (RNA-Seq). In dieser Arbeit wird das Programm INSEGT entwickelt, das Alignments von RNA-Seq Reads (Single-End oder Paired-End) mit Hilfe von gegebenen Genannotationen effizient analysiert. Dabei können die Exon-, Transkript- und Genexpressionslevel der gegebenen Annotationen gemessen werden. Wenn Readalignments mehrere Exons überspannen, kann INSEGT aus diesen mögliche Exonkombinationen (Tupel) und deren Expressionslevel ermitteln. Dadurch können zusätzlich unbekannte Spleißvarianten aufgedeckt werden. Je nach Anforderungen des Benutzers können alle Tupel einer bestimmten Länge oder nur maximale Tupel gebildet werden. Schließlich wird INSEGT auf Single-End-Read und Paired-End-Read Datensätze vom Gehirn angewandt. Die Laufzeit und der Speicherverbrauch steigen linear mit der Größe des Datensatzes an und somit wäre INSEGT in der Lage Millionen von Reads zu analysieren.

Danksagung

Als erstes möchte ich ganz besonders meinem Betreuer Marcel Schulz für die gute Betreuung danken sowie für die zahlreichen Diskussionen und Verbesserungsvorschläge. Auf diese Weise hat die Arbeit Spaß gemacht.

Außerdem möchte ich David Weese danken, der mir in SeqAn spezifischen Problemen und Fragestellungen stets sehr geholfen hat.

Vielen Dank an meine Eltern, die mich auch aus weiter Entfernung unterstützt haben. Besonders an meine Mutter, die während des gesamten Zeitraums immer ein offenes Ohr für mich hatte. Desweiteren möchte ich meinen Freunden für die vielen netten Kleinigkeiten danken, die mir Motivation gegeben haben. Zuletzt noch ein Dankeschön an Ramesh, der mir selbst in besonders anstrengenden Zeiten geholfen hat entspannt zu bleiben.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde vorher nicht in einem anderen Prüfungsverfahren eingereicht und die eingereichte gedruckte Version entspricht der auf dem elektronischen Speichermedium. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Gliederung

Bilderverzeichnis

1 Bezeichnungen und Definitionen

1.1 Bezeichnungen

Mapping (Gen-) Kartierung.

Store Speicher.

Count Anzahl. Damit ist hier die Anzahl der gematchten Reads gemeint.

Parent Eltern. Hiermit ist ein übergeordneter Annotationsbereich bzgl. einer Annotation gemeint, z.B. das Gen von einem Exon.

1.2 Definitionen

Read Ein sequenzierter Abschnitt des Transkripts, wobei zunächst nur die Sequenz bekannt ist und nicht die Position innerhalb des Genoms.

Contig Bezeichnet eine durchgängige Sequenz, z.B. ein Chromosom.

gegappte Sequenz Sequenz, die Gaps aus einem Alignment enthält.

Matepair Zwei Reads, die durch Paired-End-Sequenzierung entstanden sind und somit in einem bestimmten Abstand zueinander stehen und von einem DNA/RNA-Fragment abstammen.

Matching Übereinstimmung, u.a. in Alignments.

Splicesite Stellen, an denen die prä-mRNA gespleißt wird.

Splicejunction Übergang zwischen zwei Exons in der mRNA, an dem ein Intron herausgespleißt wurde.

Coverage Abdeckung. Die Coverage eines DNA-Abschnitts bzgl. Reads ist die Anzahl der Reads, die auf genau diesen mappen.

Seed Hier bezogen auf BLAST: Kurze Alignments zwischen Tupeln der zu alignierenden Sequenz und der Referenzsequenz. Diese dienen anschließend als Basis für die Erweiterung zu längeren Alignments.

Map Hier die Bezeichnung einer Datenstruktur, die aus einem Container besteht, in dem Datenkombinationen gespeichert werden. Dabei werden jeweils einem Schlüsselwert bestimmte Daten zugeordnet, so dass damit ein effizienter Zugrif möglich ist.

2 Einleitung

2.1 Hintergrund

Alternatives Spleißen (AS) ist der Vorgang, bei dem aus einer prä-mRNA durch unterschiedliches Spleißen verschiedene mRNAs entstehen können. Dabei wird die prä-mRNA an den Splicesites so geschnitten, dass Introns entfernt werden. Diese Splicesites weisen bestimmte Motive auf, meist GT am 5'-Ende und AG am 3'-Ende des Exons. Es gibt vier verschiedene Mechanismen des AS: Bei der Intron-Retention wird ein vorhandenes Intron nicht herausgespleißt. Beim Exon-Skipping wird eine Akzeptorsite und damit das ganze Exon übersprungen. Daneben gibt es noch Formen, bei denen es mehrere mögliche Splicesites für ein Exon gibt. Dazu zählen alternative Donorsites und alternative Akzeptorsites (Abb. 1). AS ist somit von besonderer Bedeutung für die enorme funktionelle Komplexität von Genomen bei höher entwickelten eukaryotischen Organismen [1]. So kann ein einziges Gen viele unterschiedliche Proteine mit unterschiedlichen Funktionen kodieren. Diese können zu unterschiedlichen Zeitpunkten, in unterschiedlichen Gewebetypen oder auch in einem Gewebetyp simultan synthetisiert werden. Dies verdeutlicht, dass nicht allein die Anzahl der Gene in einem Organismus ausschlaggebend für die Komplexität ist, sondern vielmehr die Proteindiversität, welche durch AS entsteht [8]. Außerdem kann AS besonders wichtig für evolutionäre Prozesse sein, da unterschiedliche Protein-Domänen neu kombiniert werden und somit Proteine mit veränderten Funktionen entstehen können. Dabei werden Spezies spezifische Spleißvarianten zunächst nur in einer geringen Menge exprimiert, was ein hervorragendes sog. „test bed" für neue Proteine darstellt [1].

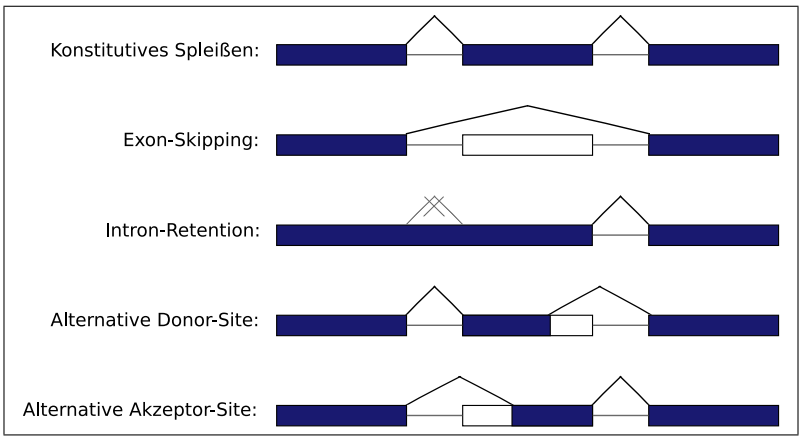

Abbildung 1: Alternative Spleißformen: Die dabei jeweils entstehende mRNA setzt sich aus den eingefärbten Bereichen zusammen.

Um DNA- und RNA-Sequenzen zu entschlüsseln und somit AS-Ereignisse zu identifizieren, wurden bisher Methoden wie EST (expressed-sequence-tag) und Sanger-Sequenzierung verwendet. Für die Gewinnung von ESTs wird mRNA zunächst in cDNA umgeschrieben und anschließend geklont. Diese Klone werden dann sequenziert. Allerdings sind die Kosten dafür relativ hoch und die Qualität ist u.a. durch bakterielles Klonen stark eingeschränkt [9]. Zur genomweiten Expressionsanalyse waren bisher Microarrays die Methode der Wahl. Dabei werden kurze bekannte DNA-Fragmente auf den Microarrays fixiert. Die zu untersuchenden mRNAs werden in cDNA umgeschrieben, markiert (z.B. mit Fluoreszenzfarbstoffen) und auf das

Array aufgetragen. Komplementäre Bereiche hybridisieren und können anschließend detektiert werden. Ein Nachteil bei dieser Methode ist ihre limitierte Sensitivität, so können schwach exprimierte Transkripte oft nicht detektiert werden [13]. Zudem können nur bereits bekannte Sequenzen aufgedeckt werden [13] und es entstehen Artefakte, z.B. durch Kreuz-Hybridisierungen bei untereinander ähnlichen Sequenzen [9].

Neue Next-Generation-Sequencing (NGS) Verfahren, wie z.B. die auf Illumina-Solexa basierende RNA-Seq, können mit geringen Kosten in kurzer Zeit mehrere Millionen von kurzen sequenzierten Reads erzeugen. Damit werden ganz neue Möglichkeiten in der Messung von Expressionsleveln und der Aufdeckung von bisher unbekannten DNA- bzw. RNA-Sequenzen eröffnet. Bei der RNA-Seq wird zunächst poly(A)-RNA extrahiert und mittels reverser Transkription in doppelsträngige cDNA umgeschrieben. Anschließend wird diese für die Illumina-Sequenzierung auf eine bestimmte Länge fragmentiert und schließlich sequenziert [11]. Die Sequenzierung kann sowohl von nur einem Ende (single-end) oder von beiden Enden (paired-end) stattfinden. In beiden Fällen werden meist nur die ersten 25-50 Basen sequenziert [14]. Bei der Paired-End-Methode werden beide Enden der cDNA-Sequenz abgelesen, so dass zwei Reads (ein Matepair) entstehen, die vom jeweils anderen Strang abstammen und deren Entfernung ungefähr bekannt ist. Matepairs bringen somit zusätzliche Informationen, da sie viel mehr abdecken, als ein einzelner Read [9]. Für beide Methoden besteht für anschließende Analysen oft das Problem, dass die Information, von welchem Strang ein Read abstammt, verloren geht. Deswegen muss dabei die Orientierung ggf. nachträglich rekonstruiert werden. Aligniert man die RNA-Seq Reads gegen ein Referenz-Genom, erhält man über die Coverage direkte Informationen über die jeweiligen Expressionslevel.

Der Nachteil der RNA-Seq ist, dass die Reads im Vergleich zu den EST-Reads mit mehreren Hundert Nukleotiden deutlich kürzer sind und die Fehlerrate im Vergleich zu Sanger höher ist. Im Vergleich zu der Microarray-Analyse hat sie den großen Vorteil, dass auch bisher unbekannte DNA-Sequenzen, wie z.B. Abberationen bei Krankheiten oder neue Spleißvarianten eines Genes identifiziert werden können. Auch hat sich die RNA-Seq als deutlich sensitiver herausgestellt und konnte bis zu 25 % mehr Gene als Microarray Experimente identifizieren [13].

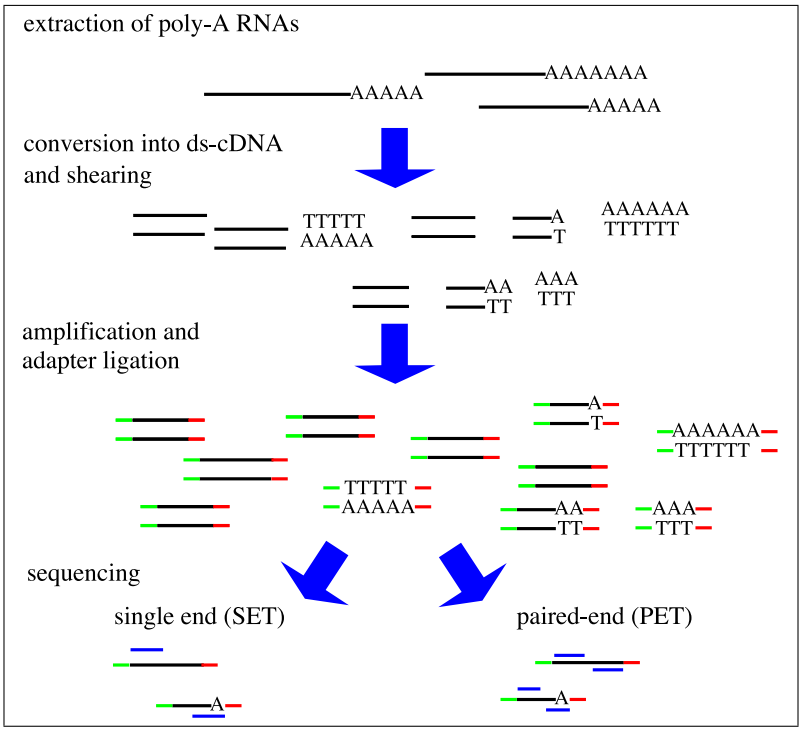

Abbildung 2: Ablauf des RNA-Seq Protokolls (für genauere Erläuterung siehe Text).

Die Analyse der RNA-Seq Daten kann dabei entweder mit Hilfe von gegebenen Annotationen oder durch *ab initio* Bestimmung erfolgen. Bei der Annotation basierten Analyse werden virtuell DNA-Fragmente aus verschiedenen Exon-Kombinationen gebildet, gegen welche die Reads gemappt werden [14]. Auf diese Weise werden Exon-Übergänge durch gemappte Reads nachgewiesen. *Ab initio* Verfahren sind sog. spliced Alignments und werden hauptsächlich für längere Reads und ESTs angewandt. Hierbei können Algorithmen verwendet werden, die mit modifizierten Smith-Waterman Algorithmen (dynamische Programmierung) arbeiten. Diese erlauben Introns und berücksichtigen zusätzlich Donor- und Akzeptorsites [2, 3]. Allerdings werden aufgrund der langen Laufzeit oft Heuristiken genutzt. Programme wie Exonerate [12] und EST2GENOM [10] arbeiten mit Heuristiken, die auf BLAST aufbauen. Dafür werden zunächst einzelne Seeds (siehe Kap. 1) ermittelt, die zu längeren Alignments erweitert und schließlich zusammengefügt werden. Introns und Splicesites werden dabei mit einberechnet.

Für kürzere Reads gibt es u.a. das Programm TopHat, welches ebenfalls Heuristiken verwendet. TopHat aligniert zunächst die Reads gegen das komplette Genom und ermittelt aus den gemappten Reads Consensus-Sequencen, welche die möglichen Exons darstellen. Aus diesen werden nun potentielle Splicejunctions und Transkripte erstellt, über die in einem zweiten Durchlauf die übrig gebliebenden Reads aligniert werden, um diese ggf. zu bestätigen. Um auch sehr kurze Exons zu finden, werden die Reads von TopHat vorher in kürzere Segmente geteilt, welche unabhängig voneinander aligniert werden [14]. Zudem gibt es bei TopHat die Möglichkeit, Informationen von bereits bekannten Splicesites einzubinden.

2.2 Motivation

Diese Bachelorarbeit beschäftigt sich mit der Entwicklung eines Programmes in C++ für die Zusammenführung von Informationen aus gegebenen Annotationen und Alignments. Um die Reads gegen das Genom zu alignieren, werden dabei *ab initio* Methoden benutzt. Über die Read-Coverage können direkt die Expressionslevel von Exons und Genen berechnet werden. Desweiteren werden Tupel von zusammen abgelesenen Annotationen gebildet und deren Coverage berechnet, so dass gleichzeitig eine Messung der unterschiedlichen Transkript-Expressionslevel stattfinden kann. Dies liefert wichtige Informationen über das Vorkommen von AS und bisher unbekannten Spleißvarianten. Zusätzlich können auch unbekannte RNA-Sequenzen über die alignierten Reads aufgedeckt werden

Ein weiterer Vorteil dieses Programms ist, dass mit nur einer Alignierung der Reads mehrere verschiedene Annotationssätze analysiert und deren Ergebnisse verglichen werden können. Ebenso können aber auch verschiedene Annotationssätze untereinander verglichen werden, indem anstelle von Reads Annotationen eingelesen werden und auf den jeweils anderen Annotationssatz gemappt werden.

Es gibt bereits das Programm ERANGE (Enhanced Read Analysis of Gene Expression), das u.a. RNA-Seq Reads gegen Annotationen mappt und damit Expressionslevel bestimmt [9]. ERANGE hat jedoch den Nachteil, dass es in Python geschrieben ist und eine sehr schlechte Laufzeit hat. Außerdem kann ERANGE keine Tupel von Exons bilden und somit auch keine Aussagen über alternative Spleißvarianten und deren Expressionslevel treffen.

3 Material und Methoden

3.1 Allgemeine Notationen

Bevor eine Aussage über die Gen-, die Exon- und die Transkriptexpression möglich ist, muss genau definiert werden, wann ein Read oder ein Matepair zu dem jeweiligen Objekt gezählt wird. Dabei werden folgende Notationen verwendet:

Es sei S $= s_1, \ldots, s_n$ die gegebene Contigsequenz, gegen die das Set von Reads $R_1, ..., R_l$ aligniert wurde. Jeder Read R_i kann auf mehrere Contigintervalle $I_1, \ldots, I_k \in R_i$ mappen, aufgrund von Introns. Jedes Intervall $I_j \in R_i$ für $j = 1, \ldots, k$ hat dabei mit I_j^s und I_j^e seine Start- und Endposition gegeben. Es gilt s, $e \in [1, n]$ und $s \leq e$. Ein Matepair M_m besteht aus zwei Reads R_{i1} und R_{i2}, die von demselben Transkript abstammen und einen bestimmten Abstand zueinander haben. Read R_{i1} liegt dabei bzgl. der Contigposition vor R_{i2}. Es sei das Set von Matepairs $M_1, \ldots, M_y$ gegeben.

Auf Annotationsebene sei ein Gen G mit den zugehörigen Exons $E_1, \ldots, E_m$ gegeben. Jedem Exon E_u ist eine Beginn- und Endposition durch E_u^s und E_u^s zugeordnet, wobei $s, e \in [1, n]$ gilt. Befindet sich das Exon auf dem Rückwärtsstrang, so gilt $e \leq s$. Der Übersicht halber wird hier von $s \leq e$ ausgegangen. Um Informationen über mögliche Transkripte zu erhalten, werden Exon-Tupel von allen Exons eines Gens gebildet, die mindestens durch einen Read verknüpft sind und mindestens die Länge zwei haben. Demnach gilt für die Menge der Tupel eines Gens:

$$T_{Single} = \{T_v | T_v \in \mathcal{P}(E_1, \ldots, E_m), E_1, \ldots, E_m \in G : u = 2, \ldots, m, E_{u-1}^e \leq E_u^s, |T_v| > 1\}.$$

Die Exons in einem Tupel T_v sind aufsteigend nach Contigpositionen geordnet.

Zur Aussage über das Vorkommen der jeweiligen Elemente (Intervall: I, Read: R, Matepair: M) in den verschiedenen Objekten (Exon: E, Gen: G, Tupel: T) dienen hier Indikatorvariablen. Diese werden entsprechend bezeichnet, bspw. mit $Exon_I(I_j, E_u)$ für ein Intervall bzgl. eines Exons. Die Indikatorvariable ist 1, wenn das Intervall I_j dem Exon E_u zugeordnet werden kann, ansonsten ist sie 0.

3.1.1 Zugehörigkeiten eines Reads

Um zu ermitteln, ob ein Read R_i einem Exon E_u zugehörig ist, werden dessen Intervalle $I_1, .., I_k \in R_i$ überprüft. Ein Intervall I_j ist einem Exon E_u zugehörig, wenn die Indikatorvariable $Exon_I$ an entsprechender Stelle 1 ist.

$$Exon_I(I_j, E_u, k) = \begin{cases} 1 & \text{if } k = 1 \wedge E_u^s \leq I_j^s \leq I_j^e \leq E_u^e \\ 1 & \text{else if } j = 1 \wedge E_u^s \leq I_j^s \leq I_j^e = E_u^e \\ 1 & \text{else if } 1 < j < k \wedge E_u^s = I_j^s \leq I_j^e = E_u^e \\ 1 & \text{else if } j = k \wedge E_u^s = I_j^s \leq I_j^e \leq E_u^s \\ 0 & \text{else} \end{cases} \tag{1}$$

Für den einfachen Fall, dass der Read aus nur einem Intervall besteht, so wird der Read zugeordnet, wenn das Intervall innerhalb der Exongrenzen E_u^s und E_u^e liegt (siehe (1), Fall 1). Handelt es sich jedoch um mehrere Intervalle, so müssen die inneren Intervallgrenzen zusätzlich bündig mit den Exongrenzen abschließen. Das hat den Hintergrund, dass hier nur Reads gezählt werden sollen, die mit bekannten Annotationen erklärt werden können. Das erste Intervall wird nur für seine Endposition I_j^e auf Übereinstimmung mit der Exon-Endposition E_u^e überprüft (siehe (1), Fall 2). Alle mittleren Intervalle werden für I_j^s und I_j^e überprüft (siehe (1), Fall 3). Das letzte Intervall wird entsprechend nur für seine Startposition getestet (siehe (1), Fall 4). Trifft keiner der Fälle zu, so wird die Indikatorvariable auf 0 gesetzt (siehe (1), Fall 5).

Daraus folgt, dass es pro Read höchstens ein Intervall geben kann, welches einem Exon zugehörig ist. Für den Read R_i bzgl. des Exons E_u gilt also:

$$Exon_R(R_i, E_u) = \begin{cases} 1 & \text{if } \exists\, I_j \text{ for } j \in [1, k] : Exon_I(I_j, E_u, k) = 1 \\ 0 & \text{else} \end{cases} . \tag{2}$$

Da die Reads von Transkripten abstammen, müssen alle Intervalle der Reads in zugehörigen Annotationsbereichen des Gens mappen. Daraus ergibt sich die Zugehörigkeit eines Reads R_i mit seinen Intervallen $I_1, \ldots, I_k \in R_i$ zu einem Gen G_g, welches aus den Exons $E_1, \ldots, E_m$ besteht, folgendermaßen:

$$Gen_R(R_i, G_g) = \begin{cases} 1 & \text{if } \forall I_j \in R_i \text{ for } u \in [1, m] : \exists E_u : Exon_I(I_j, E_u, k) = 1 \\ 0 & \text{else} \end{cases} . \tag{3}$$

Für das Programm INSEGT können zwei Arten von Tupeln für T_{Single} gebildet werden, max-Tupel oder n-Tupel. Ein Read R_i deckt ein n-Tupel $T_v \in T_{Single}$ bestehend aus den Exons $E_1, \ldots, E_n \in T_v$ ab, wenn der Read allen Exons des Tupels zugeordnet werden kann. Der Read ist mit seinen Intervallen $I_1, .., I_k \in R_i$ gegeben, wobei er auch n-Tupel abdecken kann, die weniger als k Exons enthalten. Die einzelnen gemappten Readintervalle $I_r, .., I_{r+n-1}$ mit $r \in [1, k-n+1]$ müssen aufeinanderfolgend sein.

$$n\text{-}Tupel_R(R_i, T_v) = \begin{cases} 1 & \text{if } \exists r \in [1, k-n+1], \forall E_u \in T_V : Exon_I(I_{r+u-1}, E_u) = 1 \\ 0 & \text{else} \end{cases} \tag{4}$$

max-Tupel sind Tupel, die von dem gesamten Read abgedeckt werden. Jedes Intervall des Reads muss in das Tupel mappen, so dass für die gemappten Intervalle $I_1, .., I_k \in R_i$ gilt:

$$max\text{-}Tupel_R(R_i, T_v) = \begin{cases} 1 & \text{if } \forall E_u \in T_v : Exon_I(I_u, E_u) = 1 \\ 0 & \text{else} \end{cases} . \tag{5}$$

3.1.2 Zugehörigkeiten eines Matepairs

Ein Matepair M_m, bestehend aus den Reads R_{i1} und R_{i2}, ist einem Exon E_u zugehörig, wenn beide seine Reads diesem Exon zugeordnet werden können (siehe (6)). Es gehört zu einem Gen G_g, wenn beide Reads entsprechend diesem Gen zugeordnet werden können (siehe (7)).

$$Exon_M(M_m, E_u) = \begin{cases} 1 & \text{if } Exon_R(R_i1, E_u) = 1 \wedge Exon_R(R_{i2}, R_u) = 1 \\ 0 & \text{else} \end{cases} \tag{6}$$

$$Gen_M(M_m, G_g) = \begin{cases} 1 & \text{if } Gen_R(R_{i1}, G_g) = 1 \wedge Gen_R(R_{i2}, G_g) = 1 \\ 0 & \text{else} \end{cases} \tag{7}$$

Für Matepairs wird die Tupelmenge T_{Single} so erweitert, dass ein Tupel auch durch zwei Matepairreads abgedeckt werden kann. Dabei wird ein Separator $\mathcal{S}$ eingeführt, der den Matepair-Übergang in einem Tupel markiert ($\mathcal{S}$ wird im folgenden auch mit "^" bezeichnet). Mit diesem Separator kann nun jedes Tupel auch durch Matepairs abgedeckt werden, zwischen denen eine genomische Lücke liegt. Für jedes Tupel der Länge z aus T_{Single} gibt es genau $z-1$ mögliche Matepairtupel, welche durch unterschiedliche Matepair-Übergänge entstehen. Um die Menge aller Matepairtupel, die aus einem einfachen Tupel aus T_{Single} entstehen können, zu definieren, wird zunächst ein Operator *concat* eingeführt. Dieser konkateniert jeweils zwei unterschiedliche Exonteilmengen des Tupels und bildet die Vereinigungen aller möglichen Konkatenationen. Damit kann kein Exon in diesem Tupel doppelt vorkommen. Das hat den Vorteil, dass z.B. der Fall *E1^E1,E2* nicht vorkommt und Matepairs, welche diese Exonkombination abdecken mit unter dem Eintrag *E1^E2* gezählt werden können.

$$concat(T_v, z) = \bigcup_{i=2}^{z} (E_1, \ldots, E_{i-1} \mathcal{S} E_i, \ldots, E_z) \tag{8}$$

Die Menge aller Matepairtupel für ein Gen wird mit T_{Pair} bezeichnet und setzt sich wie folgt zusammen:

$$T_{Pair} = \{concat(T_v, z) | T_v \in T_{Single}\} \ . \tag{9}$$

Zusätzlich wird noch ein Operator *cut* definiert, der für ein Matepairtupel jeweils die beiden Teilmengen T_1 und T_2 zurückgibt, aus denen das Tupel zusammengesetzt wurde. Dabei muss es nicht so sein, dass T_1 und T_2 Elemente aus T_{Single} sind, weil diese auch aus nur einem Exon bestehen können.

$$cut(E_1, \ldots, E_{i-1} \mathcal{S} E_i, \ldots, E_z) = \{(T_1, T_2) | T_1 = E_1, \ldots, E_{i-1} \wedge T_2 = E_i, \ldots, E_z\} \tag{10}$$

Ein Matepair M_m besteht aus den Reads R_{i1} und R_{i2}. Es deckt ein Matepairtupel T_v ab, wenn die durch den Separator getrennten Teilmengen $T_1, T_2 \subset T_v$ jeweils von einem der beiden Reads abgedeckt werden. Dabei wird die Teilmenge T_1 von Read R_{i1} abgedeckt und die Teilmenge T_2 von Read R_{i2}, weil die Exonpositionen von T_1 im Contig kleiner sind als die von T_2. Es kann vorkommen, dass das letzte Readintervall von R_{i1} und das erste Readintervall von R_{i2} im selben Exon E_u mappen. Dieses Exon ist entweder in der Teilmenge T_1 oder in der Teilmenge T_2 enthalten, weil diese disjunkt sind. Hier wird festgelegt, dass solche Reads nur dann dem Matepairtupel T_v zugeordnet werden, wenn $E_u \in T_1$ gilt. Damit wird sichergestellt, dass der Read nicht für mehrere Tupel gleichzeitig gezählt wird. Außerdem können in INSEGT auf diese Weise Matepairs, die zwar denselben Exons zugehörig sind, sich aber in ihrem Übergang darin unterscheiden, ob der zweite Read auch noch in das letzte Exon des ersten Reads mappt, unter einem Eintrag gezählt werden. Entsprechend der unterschiedlichen Tupelarten für einzelne Reads werden hier ebenfalls n-Tupel und max-Tupel gebildet.

Ein n-Tupel aus T_{Pair} besteht aus den Teilmengen T_1 und T_2, die beide genau n Exons enthalten. Das Matepair M_m ist dem n-Tupel $T_v \in T_{Pair}$ zugehörig, wenn Read R_{i1} in das n-Tupel T_1 und Read R_{i2} in das n-Tupel T_2 mappt.

$$n\text{-}Tupel_M(M_m, T_v) = \begin{cases} 1 & \text{if } T_1, T_2 \in cut(T_v), n = |T_1| = |T_2| : \\ & \quad n\text{-}Tupel_R(R_{i1}, T_1) = 1 \wedge \\ & \quad n\text{-}Tupel_R(R_{i2}, T_2) = 1 \\ 0 & \text{else} \end{cases} \tag{11}$$

Ein max-Tupel aus T_{Pair} besteht aus den Teilmengen T_1 und T_2. Das max-Tupel $T_v \in T_{Pair}$ wird von dem Matepair M_m abgedeckt, wenn das max-Tupel T_1 von dem Read R_{i1} und das max-Tuple T_2 von dem Read R_{i2} abgedeckt wird (siehe (13),

Fall 1). Für den Fall, dass beide Reads in ein gemeinsames Exon mappen, ist die Teilmenge T_2 kein max-Tupel für den Read R_{i2} mehr, weil das erste Exon in das der Read mappt in T_2 fehlt. Deshalb wird zusätzlich geprüft, ob das max-Tuple T_3, welches aus dem letzten Exon von T_1 folgend von allen Exons aus T_2 gebildet wird, von dem Read R_{i2} abgedeckt wird (siehe (13), Fall 2). Dafür wird ein Operator *lastExon* eingeführt, der das letzte Exon aus einem Tupel zurückgibt.

$$lastExon(T_v) = E_{\max\{u|E_u \in T_v\}} \tag{12}$$

$$max\text{-}Tupel_M(M_m, T_v) = \begin{cases} 1 & \text{if } T_1, T_2 \in cut(T_v): \\ & \quad max\text{-}Tupel_R(R_{i1}, T_1) = 1 \wedge max\text{-}Tupel_R(R_{i2}, T_2) = 1 \\ 1 & \text{else if } T_1, T_2 \in cut(T_v), T_3 = lastExon(T_1) \cup T_2: \\ & \quad max\text{-}Tupel_R(R_{i1}, T_1) = 1 \wedge max\text{-}Tupel_R(R_{i2}, T_3) = 1 \\ 0 & \text{else} \end{cases} \tag{13}$$

Desweiteren werden für Matepairs kumulative n-Tupel eingeführt. Ein kumulatives n-Tupel aus T_{Pair} besteht aus den Teilmengen T_1 und T_2, wobei diese jeweils aus bis zu n Exons bestehen können. Die Teilmengen T_1 und T_2 können auch unterschiedlich groß sein. Das kumulative n-Tupel $T_v \in T_{Pair}$ wird von dem Matepair M_m abgedeckt, wenn das n_1-Tupel T_1 von Read R_{i1} und das n_2-Tupel T_2 von Read R_{i2} abgedeckt wird.

$$kum.\ n\text{-}Tupel_M(M_m, T_v) = \begin{cases} 1 & \text{if } T_1, T_2 \in cut(T_v), n_1 = |T_1|, n_2 = |T_2|: \\ & \quad n_1\text{-}Tupel_R(R_{i1}, T_1) = 1 \wedge n_2\text{-}Tupel_R(R_{i2}, T_2) = 1 \\ 0 & \text{else} \end{cases} \tag{14}$$

3.1.3 Ausgleich von Ungenauigkeiten in den Intervallen

Bei der Überprüfung der Intervallgrenzen (siehe Kap. 3.1.1), um ein Readintervall eindeutig einem Exon zuzuordnen, wird eine gewisse Abweichung unter einem gegebenen Grenzwert erlaubt. Der Standardwert beträgt 5 bp. Dadurch werden mögliche Artefakte aus dem Alignment ausgeglichen. Zudem werden kleine Abweichungen bei den Splicesites toleriert. So gibt es beispielsweise sog. NAGNAG-Spleißmotive [1], bei denen sowohl das erste AG als auch das zweite AG als Splicesite dienen kann. Die zugehörigen Exons unterscheiden sich nur um wenige Basen und werden unter derselben Annotation geführt.

3.1.4 Expressionslevel

Die Expressionslevel der verschiedenen Objekte ergeben sich aus der Anzahl der zugehörigen Reads, wobei Matepairreads zusammen immer nur einmal gezählt werden. Dafür wird der Matepaircount ggf. von dem Count der Reads für Exons und Gene wieder abgezogen (siehe (15), (16)). Für ein gegebenes Set von Reads $R_1, ..., R_l$ und Matepairs $M_1, \ldots, M_y$ berechnen sich die Counts wie folgt.

$$Count(E_u) = \sum_{i=1}^{l} Exon_R(R_i, E_u) - \sum_{m=1}^{y} Exon_M(M_m, E_u) \tag{15}$$

$$Count(G_g) = \sum_{i=1}^{l} Gen_R(R_i, G_g) - \sum_{m=1}^{y} Gen_M(M_m, G_g) \tag{16}$$

Für Tuple aus T_{Single} werden alle zugehörigen Reads gezählt. Abhängig von den Forderungen des Programmbenutzers werden diese für alle n-Tupel oder für alle max-Tupel aus T_{Single} oder für alle kumulativen n-Tupel gezählt. Für kumulative n-Tuple wird einfach der Count für alle n-Tuple aus T_{Single} der Länge $2, \ldots, n$ ermittelt.

$$Count_{n\text{-}Tupel}(T_v) = \sum_{i=1}^{l} n\text{-}Tupel_R(R_i, T_v) \tag{17}$$

$$Count_{max\text{-}Tupel}(T_v) = \sum_{i=1}^{l} max\text{-}Tupel_R(R_i, T_v) \tag{18}$$

Bei Tupeln aus T_{Pair} werden alle zugehörigen Matepairs gezählt. Dabei werden entweder alle n-Tupel, alle max-Tupel oder alle kumulativen n-Tupel aus T_{Pair} ermittelt.

$$Count_{n\text{-}Tupel}(T_v) = \sum_{m=1}^{y} n\text{-}Tupel_M(M_m, T_v) \tag{19}$$

$$Count_{max\text{-}Tupel}(T_v) = \sum_{m=1}^{y} max\text{-}Tupel_M(M_m, T_v) \tag{20}$$

$$Count_{kum.\ n\text{-}Tupel}(T_v) = \sum_{m=1}^{y} kum.\ n\text{-}Tupel_M(M_m, T_v) \tag{21}$$

Der Count an sich sagt allerdings noch relativ wenig über die Expression aus, da er von der Anzahl der insgesamt verwendeten Reads und der Länge des zu untersuchenden Objekts abhängt. Um vergleichbare Ergebnisse zu produzieren, werden diese normalisiert, indem die Länge der Annotation und Anzahl der Reads miteinberechnet werden. Die Messung findet in RPKM (reads per kilobase of exon model

per million mapped reads) statt [9]. Dabei wird die Anzahl der in den betreffenden Bereich gemappten Reads pro Kilobase pro eine Million insgesamt verwendeter Reads angegeben:

$$RPKM = \frac{10^9 \cdot Count}{N \cdot L} \ . \tag{22}$$

N ist die Anzahl aller verwendeter Reads. L ist die Länge des betreffenden Bereichs, für den das Expressionslevel bestimmt werden soll. Für ein Gen ist die Länge die Summe aller zugehörigen Exonlängen.

3.2 SeqAn

Die Grundlage für diese Arbeit stellt SeqAn (Sequence Analysis), eine effiziente, generische C ++ Library für Sequenzanalysen dar. Eine der wichtigsten Datenstrukturen von SeqAn sind Strings zur Speicherung von Sequenzen, bestehend aus DNA oder anderen Alphabeten [5]. Darauf aufbauend gibt es StringSets, welche wiederum Strings von Strings sind. Diese sind besonders speichereffizient, da sie im Gegensatz zu Matrizen nur so viel Speicher für die einzelnen Strings in Anspruch nehmen, wie auch benötigt wird. Sie können aus mehreren unterschiedlich langen Strings bestehen.

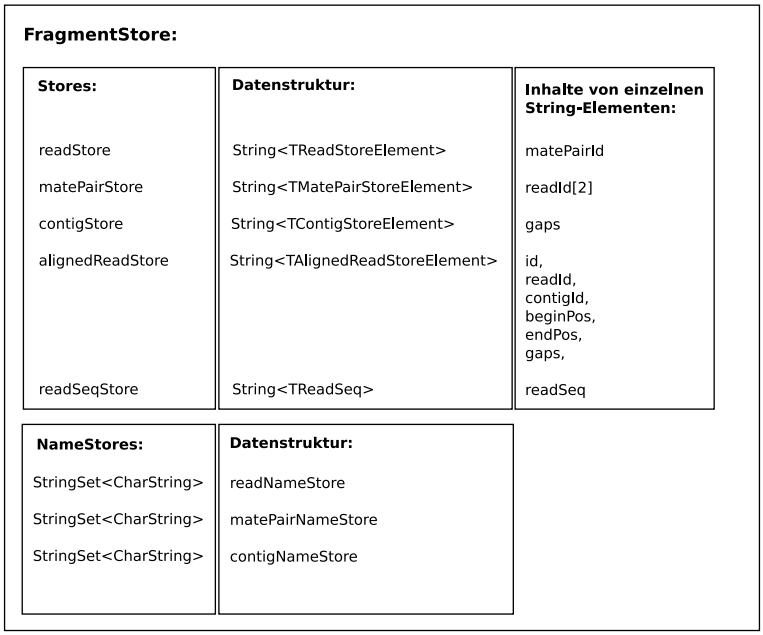

Abbildung 3: SeqAn *FragmentStore*: Dieser enthält die Stores für Reads, Matepairs, Contigs und deren Alignments sowie die zugehörigen Name-Stores. Abgebildet sind nur die in diesem Kontext wichtigen Strukturen.

Diese Arbeit baut auf den bereits in SeqAn implementierten *FragmentStore* und dessen Funktionen auf. Er enthält u.a. die verschiedenen Contigs, die daran alignierten Reads und deren Alignments, welche in weiteren assoziierten Stores gespeichert werden (Abb. 3). Diese Stores bestehen aus den beschriebenen Strings bzw. StringSets. Abgesehen vom *AlignedReadStore* sind alle Stores und ihre zugehörigen

Name-Stores so aufgebaut, dass jeder Eintrag an der zu der jeweiligen ID passenden Position gespeichert ist. So ist ein schneller Zugriff möglich und eine zusätzliche Speicherung der Id wird gespart. Beim *AlignedReadStore* ist dies nicht möglich, da dieser je nach Bedarf sortiert werden kann. Die Orientierung des Reads bzgl. des Contigs kann an der Beginn- und Endposition abgelesen werden. Um Speicherplatz zu sparen, werden die Alignments nicht vollständig sondern nur in einer sog. *Gap-Anchor*-Struktur (String von *Gap-Anchors*) gespeichert. Ein *Gap-Anchor* besteht aus den Positionen in der ungegappten und in der gegappten Contig-Sequenz jeweils hinter einem Gap:

$$\begin{array}{rl} \text{ungegappte Sequenzposition:} & \binom{1}{2}\ \binom{3}{6}\ . \\ \text{gegappte Sequenzposition:} & \end{array}$$

Damit ist indirekt die Anzahl der Gaps vor diesen Positionen gegeben. Der erste *Gap-Anchor* gibt in dem obigen Beispiel an, dass Position 1 der ungegappten Sequenz und Position 2 der gegappten Sequenz auf ein Gap folgen. Da es sich um den ersten *Gap-Anchor* handelt, ist genau ein Gap vor diesen Positionen eingefügt worden. Der zweite *Gap-Anchor* sagt aus, dass insgesamt 3 Gaps (6 - 3) vor den jeweiligen Positionen vorkommen. Zieht man das erste Gap davon ab, so kommt man auf eine Anzahl von genau zwei Gaps. Im *AlignedReadStore* werden dabei nur die Gaps, die in dem jeweiligen Read eingefügt wurden sowie zusätzlich die Start- und die Endposition in der gegappten Contigsequenz gespeichert. Für das Contig werden im *ContigStore* separat Gaps gespeichert, allerdings nur einmal für alle alignierten Reads zusammen. Dies geschieht, indem jedesmal, wenn ein neuer Read aligniert wird, die Contiggaps so erweitert werden, dass alle Alignments damit erfüllt werden.

In SeqAn gibt es bereits Funktionen, welche Alignments, z.B. im SAM (Sequence Alignment/Map) Format einlesen sowie eine Funktion, die die zugehörigen Contigs im FASTA Format einliest.

3.3 Intervallbäume

Für die Suche von bestimmten Punkten in gegebenen Intervallen eignen sich Intervallbäume. Ein Intervallbaum ist ein binärer Baum, dessen Knoten jeweils einen Punkt aus dem Gesamtintervall enthalten (Abb. 4). Dabei sind immer alle Punkte im linken Teilbaum kleiner als im rechten Teilbaum. Zusätzlich enthält jeder Knoten zwei Listen mit Intervallen, die den zugehörigen Punkt enthalten. Dabei sind die Punkte so gewählt, dass jedes Intervall in nur genau einem Knoten auftaucht. Daraus schließt sich, dass alle Intervalle mit einem rechten Intervallwert, der kleiner als der aktuelle Punkt ist, auch im linken Teilbaum zu suchen sind und umgekehrt. In der ersten Liste sind alle Intervalle aufsteigend nach dem linken Intervallwert sortiert, in der zweiten alle aufsteigend bzgl. des rechten Wertes. Somit ist eine effiziente Suche nach Intervallen, die einen gegebenen Punkt enthalten, in einer Laufzeit von $\mathcal{O}(\log n)$ möglich, wobei n die Anzahl aller Intervalle ist.

Diese Struktur ist bereits als sog. *IntervalTree* mit entsprechender Suchfunktion in SeqAn implementiert [6].

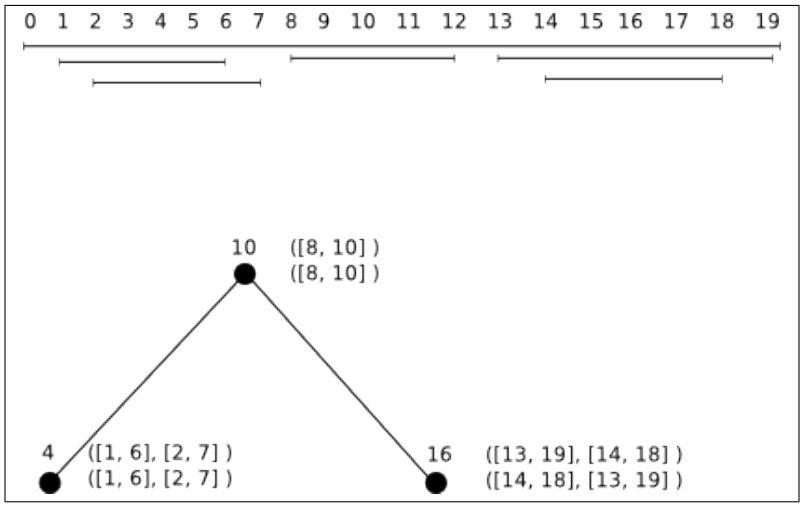

Abbildung 4: Intervalle mit zugehörigem Intervallbaum

3.4 Verwendete Formate

Die Eingabe der Annotationen in das Programm erfolgt im GFF (General Feature Format). In diesem werden allgemein Gene, Exons und andere Strukturen von DNA, RNA und Proteinen dargestellt. Dabei müssen folgende Spezifikationen für das Programm INSEGT eingehalten werden (Abb. 5):

In der ersten Spalte steht der Sequenzname, in der 4. und 5. Spalte stehen ggf. Start- und Endposition und in der 7. Spalte steht die Orientierung. Der Name der Annotation und ggf. der Name der übergeordneten Annotation (z.B. das Gen bzgl. eines Exons) befinden sich in der 9. Spalte. Dabei steht vor dem eigentlichen Namen der Annotation ein "ID=" und ggf. vor dem Elternnamen ein "ParentID=". Sofern beide Namen vorhanden sind, werden sie durch ein ";" ohne Leerzeichen getrennt.

Exoneinträge können sowohl mit, als auch ohne Gennamen vorkommen, jedoch müssen die Start- und die Endposition angegeben sein. Da die Orientierung separat verwaltet wird, ist die Startposition immer kleiner als die Endposition. Geneinträge können hingegen ohne Start- und Endposition vorkommen, sofern zugehörige Exoneinträge vorhanden sind. Die Reihenfolge der Einträge spielt keine Rolle.

```
<Sequenzname> < > < > <Start> <Ende> < > <Strang> < > <ID;ParentID>

chr1          .   .   200     1000    .   +         .   ID=ENSG1
chr1          .   .   300     450     .   +         .   ID=ENSG1-1;ParentID=ENSG1
chr1          .   .   700     600     .   -         .   ID=ENSG2-1;ParentID=ENSG2
```

Abbildung 5: Beispiel für ein .gff-File. Spalten, welche für dieses Programm nicht von Bedeutung sind, wurden durch ein "< >" bzw. "." markiert.

Die Alignments der gegen eine Contigsequenz alignierten Reads werden im SAM Format eingegeben [7], die zugehörige Contigsequenz im FASTA Format.

3.5 Verwendete Daten

Als Annotationssatz für alle Experimente wurde ein Datensatz von 213948 Exon-Annotationen des menschlichen Genoms erzeugt, basierend auf ENSEMBL 46 [13]. Für alle Experimente wurde Genomversion HG18 verwendet.

Für die interne Verwaltung der Alignments im *FragmentStore* wurden die Contigsequenzen benötigt, um ggf. Gaps einzufügen, so dass alle Alignments von verschiedenen Reads gegen ein Contig konsistent sind. Da die Contigsequenz selbst nicht für das Programm INSEGT benutzt wird, wurden als provisorische Lösung nur die Contignamen mit einer Pseudosequenz AA eingelesen, um Speicherplatz und Laufzeit zu reduzieren. Das funktioniert, weil die *Gap-Anchor*-Sequenzen der Reads und der Contigs unabhängig von der Sequenz erstellt werden.

Für die Alignments wurden unterschiedliche Datensätze mit verschiedenen Größen und Eigenschaften verwendet. Die Orientierung der Reads ist immer bekannt. Zunächst wurden Datensätze von Reads erstellt, die von mRNA aus Gehirnzellen gewonnen wurden. Dabei wurden Illumina 32 bp Single-End-Reads [15] benutzt, sowie Illumina 32 bp Paired-End-Reads (nicht publizierte Daten des MPI für molekulare Genetik). Die Single-End-Reads wurden mit RazerS [16] und den folgenden Parametern aligniert: *-i 92 -pa –max-hits 15 -rr 99*. Die Paired-End-Reads wurden mit ELAND [4] mit Standardparametern aligniert, es wurden nur Alignments behalten, bei denen beide Reads eines Paares genomweit eindeutig mappen. Bei beiden Alignments wurde nur gegen HG18 gemappt und Splicejunctions nicht berücksichtigt.

Um die Bildung der verschiedenen Tupelgrößen zu testen, wurde ein Datensatz mit langen Transkripten erstellt. Dabei wurden aus der Ensembl Datenbank Version 53 für jedes Gen von Chromosom 18 bis zu zwei Transkripte heruntergeladen. Anschließend wurden diese mit Exonerate2.2 [12] und den Parametern *-m est2genome –percent 97 –geneseed 500 –bestn 1* aligniert. Dafür wurden Splicesites berücksichtigt.

Die verschiedenen Ausgabeformate der Aligner wurden ins SAM Format umgewandelt.

4 Design und Implementation

4.1 Ziele

Das Ziel dieser Arbeit ist es, ein Programm zu erstellen, welches in kurzer Zeit die Informationen von hunderten Millionen Reads verarbeiten kann, um u.a. Aussagen über die verschiedenen Expressionslevel zu treffen. Es sollen sowohl Informationen von Single-End-Reads als auch von Paired-End-Reads verarbeitet werden können. Für die gegebenen Annotationen sollen die gemappten Reads gezählt werden, so dass Aussagen über Exon-Expressionslevel und auch über Gen-Expressionslevel möglich sind. Zusätzlich ist geplant, durch Tupelbildung von durch Reads verknüpften Annotationen, alternative Spleißvarianten eines Gens und deren unterschiedliche Expressionslevel zu bestimmen. Dabei ist wichtig, dass diese Tupel möglichst speichereffizient verwaltet werden. Bei der Verwendung von Paired-End-Reads wird die darin enthaltene Information ebenfalls genutzt werden, um Rückschlüsse auf die Zusammenstellung der Tranksripte zu ermöglichen. Außerdem sollen unbekannte exprimierte Bereiche aufgedeckt werden können.

4.2 Konzept

Das Programm erhält den Namen INSEGT, welcher für „INtersecting SEcond Generation sequencing daTa with annotation" steht. Zu Beginn wird erst einmal auf das grobe Konzept des Programmes eingegangen, welches auch in Abbildung 6 dargestellt ist. Diese enthält zusätzlich Verweise auf die nachfolgenden Kapitel, in denen die jeweiligen Teilbereiche detaillierter beschrieben werden.

Als erstes werden die Annotationsbereiche aus dem GFF-File eingelesen (siehe Kap. 4.3). Um diese zu verwalten, wird der SeqAn-*FragmentStore* um einen *AnnotationStore* und einen *AnnotationNameStore* erweitert. Diese Stores werden äquivalent zu den bereits vorhandenen Stores gebildet (siehe Kap. 3.2). Sie bestehen aus Strings von den jeweiligen Elementen. Im *AnnotationStore* werden die Contig-Id, die ggf. gegebene Parent-Id sowie die Start- und die Endposition der Annotation gespeichert. Die Parent-Id ist nötig, damit für einen Read sowohl der Count der gemappten Annotation als auch der Count der zugehörigen Parent-Annotation hochgesetzt werden kann. Es ist immer eine 2-Level Hierarchie gegeben, die aus Kindern und zugehörigen Parents besteht. Anhand der Start- und Endposition kann direkt die Orientierung im Genom ablesen werden. Ist die Startposition größer als die Endposition, so befindet sich das betreffende Element auf dem Rückwärtsstrang und umgekehrt. Im *AnnotationNameStore* werden an entsprechender Position die Namen der einzelnen Annotationsbereiche verwaltet. Dadurch, dass die Id der*AnnotationStore*-Elemente indirekt durch den Index im String gegeben ist, ist ein direkter Zugriff möglich. Auf Basis des *AnnotationStores* werden dann *IntervalTreeStores* gebaut, in denen die Annotationsbereiche in Intervallbäumen (siehe Kap. 3.3) verwaltet werden.

Die gegebenen Alignments zwischen den Reads und dem Genom werden in dem *AlignedReadStore*, welcher Teil des *FragementStores* ist, gespeichert. Weil die konkreten Contigpositionen der Alignments benötigt werden, werden mittels der *GapAnchors* die Aligmentintervalle berechnet und im *AlignIntervalsStore* gespeichert (siehe Kap. 4.4).

Anhand der Alignmentintervalle werden dann die Ids der entsprechenden Annotationsbereiche, in die der Read mappt, aus dem *IntervalTreeStore* gewonnen (siehe Kap. 4.6). Anschließend werden diese noch nach bestimmten Kriterien selektiert (siehe Kap. 4.7).

Für die Verwaltung der dabei erzielten Resultate werden drei weitere Stores eingeführt. Im *ReadAnnoStore* werden die Ids der gemappten Annotationen für die einzelnen Reads gespeichert (siehe Kap. 4.8). Der *AnnoCountStore* enthält die Counts der Annotationsbereiche (siehe Kap. 4.9). Und im *TupelCountStore* werden Tupel von zusammen abgelesenen Annotationsbereichen und deren Counts gespeichert (siehe Kap. 4.10). Zu guter Letzt werden die Ergebnisse entsprechend dieser Stores ausgegeben (siehe Kap. 4.12).

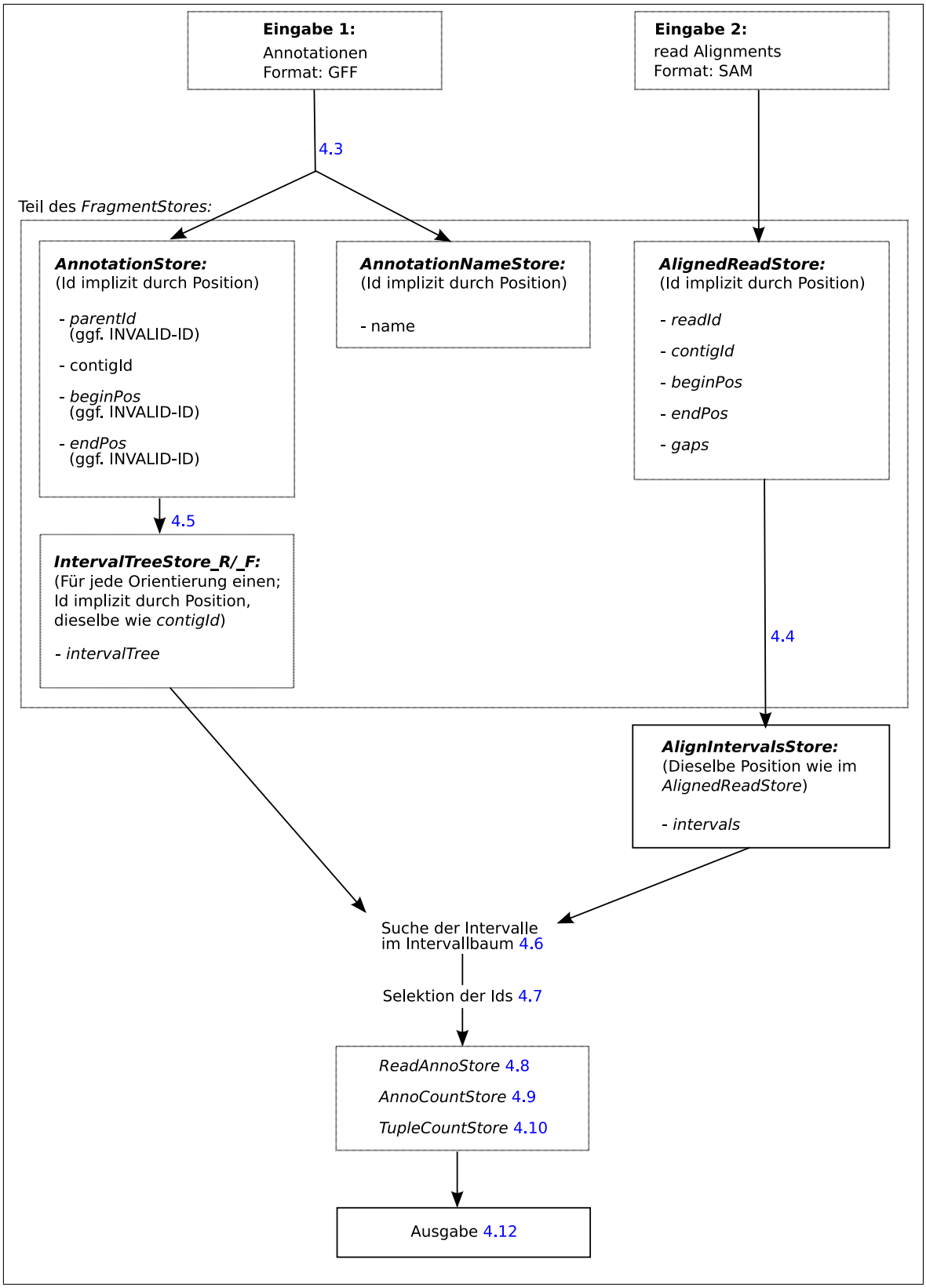

Abbildung 6: Konzept: Es sind die für das Programm verwendeten und neu erstellten Datenstrukturen abgebildet sowie deren Verknüpfungen untereinander. Die den einzelnen Teilbereichen zugeordneten Nummern geben jeweils das Kapitel an, in dem eine genauere Beschreibung stattfindet.

4.3 Einlesen der Annotationsbereiche aus einem GFF-File in den *AnnotationStore*

Zunächst wird eine Funktion benötigt, um die gegebenen Annotationen aus dem GFF-File in den *AnnotationNameStore* und den *AnnotationStore* einzulesen. Dabei wird die Reihenfolge der Einträge beibehalten. Es werden zwei Durchläufe durch das GFF-File benötigt.

Beim ersten Durchlauf werden nur die Namen der Annotationen und ggf. die Parent-Namen gelesen, welche in den *AnnotationNameStore* eingetragen werden. Um später an die Parent-Ids für die *AnnotationStore*-Einträge zu gelangen, wird eine Map (siehe Kap. 1) angelegt, in der alle Parent-Namen und die zugehörigen *AnnotationStore*-Ids gespeichert werden. Ist ein Parent-Name noch nicht in der Map vorhanden, wird ein Eintrag in den *AnnotationNameStore* zwischen geschoben (vor dem eigentlichen Eintrag) und in die Map eingefügt. Annotationen ohne Parent-Name werden zunächst als Parents behandelt und ggf. in die Map eingefügt. So wird verhindert, dass Namen, die später noch einmal als Parent-Name vorkommen, doppelt eingefügt werden.

Beim zweiten Durchlauf durch das GFF-File, wird gleichzeitig über den *AnnotationNameStore* und den *AnnotationStore* iteriert. Ist der aktuelle Parent-Name aus dem GFF-File identisch mit dem aktuellen Namen aus dem *AnnotationNameStore*, so wird zunächst ein Eintrag für die Parent-Id im *AnnotationStore* getätigt. Erst danach wird der eigentliche Eintrag gemacht. Das bringt den Vorteil, dass für ein Exon nicht explizit ein Geneintrag im GFF-File vorhanden sein muss. Sofern ein Parent-Name vorhanden ist, wird dieser automatisch generiert. Stimmt jedoch der aktuelle Name im *AnnotationNameStore* mit dem aktuellen Annotations Namen im GFF-File überein, so wird nur dieser Eintrag gemacht. Die *AnnotationStore*-Parent-Ids werden jeweils aus der Map gewonnen. Gleichzeitig werden ggf. im *AnnotationStore* die Positionen des Parents auf `INVALID_ID` gesetzt. Dadurch wird später sicher gestellt, dass für diesen Eintrag die Reads nur indirekt über passende Exons gezählt werden. Sind bspw. für ein Gen keine Exon-Annotationen vorhanden, auf welche die Reads mappen, so soll für dieses Gen der Count auch nicht hochgesetzt werden.

Aufgrund der Beibehaltung der Reihenfolge ist ein schnelles Einlesen möglich. Würden die Einträge im *AnnotationNameStore* z.B. lexikographisch sortiert werden, so würden die Parent-Ids in der Map nicht mehr stimmen und müssten erneut ermittelt werden. Ebenso wäre anschließend jedesmal eine Binärsuche notwendig, um an die dem Annotationsnamen entsprechende *AnnotationStore*-Id zu kommen.

Die Anforderungen an das GFF-File (siehe Kap. 3.4) sind relativ gering, da das Einlesen möglichst flexibel gehalten wurde. So spielt die Reihenfolge, in der die Einträge und die zugehörigen Parent-Einträge auftauchen, keine Rolle. Die Einträge müssen auch nicht nach Contig sortiert sein. Ebenso ist es irrelevant, ob für ein eingetragenes Gen, das nur über seine Exons gezählt werden soll, Positionen angegeben sind oder nicht. Wie schon erwähnt werden die Positionen gelöscht, sobald ein Kindereintrag auftaucht. Folglich werden Annotationen ohne Parent und mit angegebenen Positionen nur dann direkt gezählt, wenn für sie keine Kindereinträge vorhanden sind.

4.4 Umwandlung der SeqAn *Gap-Anchor*-Struktur in eine Intervallstruktur

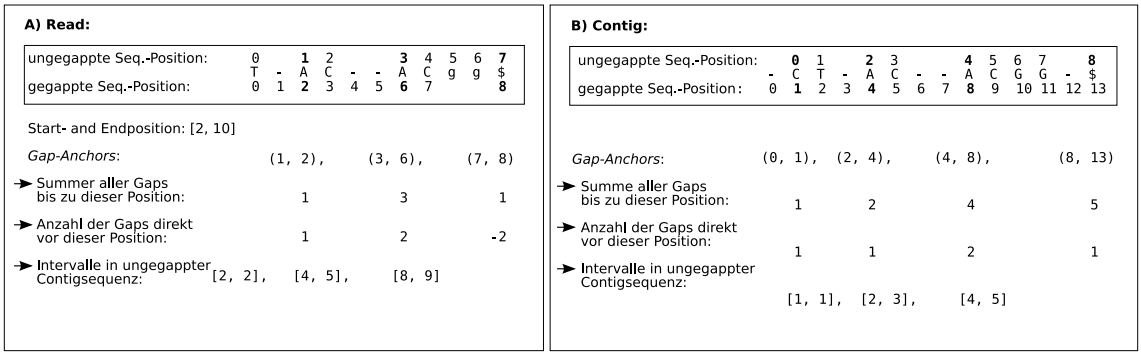

Abbildung 7: Beispiel für die Extraktion der Intervalle aus einem Alignment anhand der zugehörigen *Gap-Anchor*-Sequenzen. A) Berechnung der Intervalle in der gegappten Contigsequenz: Mittels der *Gap-Anchors* sind indirekt die Positionen und Längen der Gaps angegeben. Daraus kann man die Intervalle des Reads in seiner gegappten Sequenz ermitteln. Anschließend werden diese Intervalle durch Addition der Startposition auf die gegappte Contigsequenz projiziert. B) Berechnung der Intervalle in der ungegappten Contigsequenz: Anhand der Gaps können zunächst die einzelnen Intervalle des Contigs selbst in der gegappten Sequenz berechnet werden. Bildet man nun die Schnittmenge dieser Intervalle mit den Intervallen des Reads und zieht die Anzahl der vorher auftretenden Gaps ab, so erhält man die gesuchten Alignmentintervalle.

Wie bereits erwähnt, werden die Alignments in SeqAn in einer *Gap-Anchor*-Struktur gespeichert (siehe Kap. 3.2). Da aber für jeden Read überprüft werden muss, ob er in die gebenen Annotationsbereiche des Contigs fällt, werden konkrete Positionen der Alignments benötigt. Deshalb wird die *Gap-Anchor*-Struktur zunächst in ein Intervallformat umgewandelt, welches die einzelnen gematchten Exonbereiche darstellt. Dafür werden im ersten Schritt die Intervalle des alignierten Reads in der gegappten Contigsequenz berechnet (Abb. 7, A)). Anhand von jeweils zwei aufeinanderfolgenden *Gap-Anchors* wird die Anzahl der Gaps zwischen den gegebenen Readpositionen berechnet. Somit kann gleichzeitig auf die Readintervalle selbst zurückgeschlossen werden. Je nach Orientierung werden die Readintervalle mittels der Start- bzw. Endpostion des Reads, auf die gegappte Contigsequenz projiziert. Im zweiten Schritt werden aus diesen Intervallen die Positionen in der ungegappten Contigsequenz berechnet (Abb. 7, B). Dies geschieht, indem zunächst anhand der Contig-*Gap-Anchors* die Contigintervalle bzgl. der gegappten Sequenz ermittelt werden. Anschließend werden die Aligmentintervalle aus den Schnittmengen der Read- und der Contigintervalle berechnet und auf die ungegappte Contigsequenz projiziert. Aufgrund der entsprechend aufsteigenden Reihenfolge der Intervalle können diese Berechnungen in einem Durchgang erfolgen. Zum Schluss werden alle Intervalle, deren Distanz unter einem bestimmten Grenzwert liegt, vereinigt. Damit werden einfache Gaps ignoriert, so dass nur noch Intervalle vorliegen, die durch Introns getrennt sind.

Das Resultat für ein Alignment wird in einem String von Intervallen gespeichert. Diese Liste wird im *AlignIntervalsStore* verwaltet (Abb. 6), der äquivalent zu den anderen Stores aus einem String besteht. Dabei enstpricht die Position im *AlignIntervalsStore* der im *AlignedReadStore*. Damit kann von beiden Stores direkt auf die Information im jeweils anderen Store zugegriffen werden. Auf diese Weise wird auch

die Speicherung der zugehörigen Read- und Contig-Ids im *AlignIntervalsStore* gespart.

4.5 Bildung eines *IntervalTreeStores*

Um die passenden Annotationen aus dem *AnnotationStore* für die Alignmentintervalle eines Reads zu finden, wird eine schnelle Suche benötigt. Dafür werden die SeqAn *IntervalTrees* (siehe Kap. 3.3) verwendet.

Für den Fall, dass die Orientierungen der Reads im Genom bekannt sind, werden auf Basis des *AnnotationStores* zunächst zwei *IntervalTreeStores* (Abb. 6) gebildet. Sie enthalten für jedes Contig aus dem *ContigStore* einen *IntervalTree*, an der dem *ContigStore* entsprechenden Position. Ein *IntervalTree* ist jeweils aus den Annotationen des Vorwärtsstranges und einer aus denen des Rückwärtsstranges aufgebaut. Dadurch ist für ein gegebenes Alignment, für welches gleichzeitig die Contig-Id und die Orientierung bekannt sind, ein direkter Zugriff auf den jeweils benötigten *IntervallTree* möglich.

Für den Fall, dass für die Reads nicht bekannt ist von welchem Strang sie ursprünglich abstammen (siehe Kap. 2.1), wird für jedes Contig nur ein Baum erstellt, in dem alle Intervalle von beiden Strängen enthalten sind. Somit wird die Suche in zwei Intervallbäumen gespart. Dieser Fall muss vom Benutzer explizit angegeben werden, weil INSEGT ansonsten die Orientierung der Reads automatisch anhand der Beginn- und Endposition der Alignments berechnet.

Da die weitere Prozessierung der Ergebnisse einen direkten Zugriff auf den *AnnotationStore* verlangt, werden für jedes Intervall im *IntervalTree* zusätzlich die entsprechenden *AnnotationStore*-Ids gespeichert.

4.6 Nutzung des *IntervalTreeStores* zur effizienten Suche von passenden Annotationen

Für jeden alignierten Read können nun die gemachten Annotationen in dem zu der Orientierung und dem Contig passenden *IntervalTree* gesucht werden. Dabei wird für jedes Intervall dessen Start- und Endposition separat gesucht (siehe Kap. 4). Um Artefakte aus dem Alignment auszugleichen, wird hier für die Suche mit einem gewissen Toleranzwert gerechnet. Damit wird das gesuchte Alignmentintervall verkleinert, so dass sicher gestellt wird, dass alle Annotationen gefunden werden. Anschließend wird die Schnittmenge der dabei jeweils ermittelten *AnnotationStore*-Ids gebildet. Für jedes Intervall des Reads liegen damit alle gematchten Annotationen vor.

4.7 Selektion der gematchten *AnnotationStore*-Ids

Bei der Suche in den Intervallbäumen werden alle Annotation-Ids ausgegeben, in welche die Alignmentintervalle des Reads matchen (siehe Kap. 4.6). Allerdings gibt es Fälle, in denen ersichtlich ist, dass der Read eigentlich von einem anderen Transkript abstammt und dieser Bereich nur zufällig mit einem anderen Annotationsbereich überlappt. Gerade bei längeren Reads, die mehrere Annotationbereiche hintereinander abdecken, müssen unpassende Annotationen ausselektiert werden (siehe Kap. 3.1.1). So kann eine Annotation ausgeschlossen werden, wenn die inneren Intervallgrenzen nicht mit den Grenzen der Annotation übereinstimmen. In solch einem

Fall muss es einen anderen Annotationsbereich geben, aus dem der Read hervorgegangen ist (Abb. 8, E5 und E4). Um Artefakte in den Alignments auszugleichen muss dabei zusätzlich mit einem gewissen Toleranzwert gerechnet werden. Ist keine passende Annotation vorhanden, so wird eine `INVALID_ID` in die selektierten Ids eingefügt (Abb. 8, 2. Intervall). Damit wird später deutlich, dass es an dieser Stelle einen bisher unbekannten Contigbereich geben muss, der exprimiert wird und genauer untersucht werden kann. Da bei der Erstellung der Alignmentintervalle alle Bereiche, die nicht durch ein Intron getrennt sind, zusammengefügt wurden, ist die Überprüfung der Annoatationen relativ simpel. Jedes Intervall entspricht daher einem Exonbereich, so dass nur noch die Start- und Endpositionen verglichen werden müssen.

Es gibt allerdings bei mehreren überlappenden Annotationen auch den Fall, dass es mehr als eine valide Annotation für ein Intervall gibt (Abb. 8, E7 und E9). Aufgrunddessen werden die selektierten Ids in einem String von Strings verwaltet, damit diese Informationen nicht verloren gehen. Das Resultat ist ein String, indem jeder Eintrag einem Readintervall entspricht und aus einem weiteren String von Ids besteht.

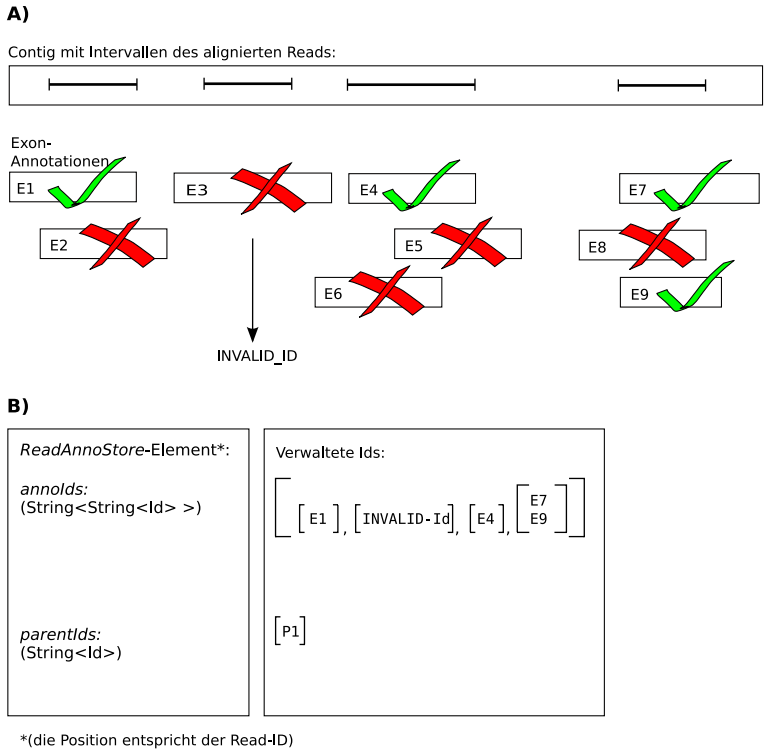

Abbildung 8: A) Selektion der *AnnotationStore*-Ids abhängig davon, ob die Intervalle des Alignments mit den Grenzen des AnnotationStore-Eintrags übereinstimmen. Für bisher unbekannte Bereiche wird eine `INVALID_ID` eingefügt. B) Verwaltung im *ReadAnnoStore*: Die gematchten Ids werden in einem String von Strings gespeichert. Für den Fall, dass es mehrere passende Annotations-Bereiche für ein Read-Intervall gibt.

4.8 Bildung des *ReadAnnoStores*

Für die Verwaltung der selektierten gematchten Annotationen eines Reads wird der *ReadAnnoStore* eingeführt. Dieser besteht aus einem String, in dem die Positionen der einzelnen Elemente jeweils den Read-Ids entsprechen (Abb. 8, B)). Die Annotation-Ids für einen bestimmten Read befinden sich an derselben Position wie der zugehörige Eintrag im *ReadStore*. Dabei bestehen die einzelnen Elemente des *ReadAnnoStores* aus den gematchten *AnnotationStore*-Ids und den entsprechenden Parent-Ids (Abb. 8, B)). Die Parent-Ids können einfach aus dem *AnnotationStore* abgelesen werden (Abb. 6). Weil nur Annotationen gezählt werden sollen, die mit dem Read übereinstimmen (siehe Kap. 3.1.1), werden nur die Parent-Ids gespeichert, in die alle Intervalle des Reads mappen. Für ihre Verwaltung reicht ein einfacher String, während die eigentlich gematchten Ids in einem String von Strings gespeichert werden. Damit wird in der zweiten Dimension die Speicherung von mehreren Ids von überlappenden Bereichen möglich.

Bei der Ausgabe für die einzelnen Reads sollen zusätzlich auch die Annotationen ausgegeben werde, in deren Parents der Read nicht komplett matcht, ohne dass diese vorher gezählt werden. Für Readintervalle, für die es demnach keine passende Annotation mit passender Parent-Id gibt, wird damit die Information über unbekannte gematchte RNA-Bereiche ausgegeben. Deshalb werden alle einzeln gematchten Annotation-Ids gespeichert, unhabhängig davon, ob der Read laut Definition (siehe Kap. 3.1.1) der Parent-Id zugehörig ist oder nicht.

4.9 Bildung des *AnnoCountStores*

Um aus den Informationen der einzelnen Reads nun an Informationen über die verschiedenen Expressionslevel der Annotationen zu gelangen, wird der *AnnoCountStore* gebildet. Dieser wird äquivalent zum *AnnotationStore* (siehe Kap. 4.2) aufgebaut, wobei an entsprechender Position der Count der jeweiligen Annotation gespeichert wird.

Dafür wird für jeden Read aus dem *ReadAnnoStore* anhand der dort verwalteten Ids der entsprechende Count um eins erhöht. Dies geschieht, indem für jede Parent-Id der entsprechende Count und der Count für zugehörige direkt gematchte Annotation-Ids um eins hoch gesetzt wird. Somit werden nur die Annotationen gezählt, die mit dem Read komplett übereinstimmen (siehe Kap. 3.1.1). Bei Matepairs, die in denselben Annotationsbereich matchen, wird dabei der Count nur einmal erhöht. Gleichzeitig werden Matepairs für eine Parent-Id und deren Kinder nur dann gezählt, wenn beide Reads in diesen Parent matchen. Andernfalls würden Parent-Ids gezählt werden, deren Annotationen den Read nicht erklären. Damit diese Matepair-Informationen abgeglichen werden können, wird der *AnnoCountStore* erst dann gebildet, wenn der *ReadAnnoStore* vollständig ist.

4.10 Bildung des *TupleCountStores*

Um nicht nur eine Aussage über die einzelnen Exons zu erhalten sondern auch über deren Kombinationen (siehe Kap. 3.1.1) in verschiedenen Transkripten und deren Häufigkeit wird der *TupelCountStore* eingeführt. Dieser verwaltet die Exonkombinationen an der Position entsprechend der ersten Id aus dem jeweiligen Tupel. Es werden zwei Sorten von Tupeln verwaltet. Zum einen die Exontupel, die durch Verknüpfungen über einzelne Reads entstehen und zum anderen solche, die zusätzlich die Verknüpfungen über Matepairs enthalten (siehe Kap. 3.1 und 3.1.1). Für Matepairtupel wird der Übergang zwischen den beiden Reads mit einem "^" markiert. Als Datenstruktur für die Tupel dient dabei wieder ein String von Strings, wobei diesmal die zweite Dimension jeweils ein Exontupel, bestehend aus den Ids, enthält (Abb. 9, B)). Ein jeweils zweiter String enthält an entsprechender Position den Count.

Die Bildung des *TupelCountStores* erfolgt mit Hilfe des *ReadAnnoStores* (siehe Kap. 4.8). Dabei wird für jeden Eintrag im *ReadAnnoStore*, also für jeden Read, wenn möglich, als erstes eine Liste mit allen möglichen Tupeln gebildet. Je nachdem, was der Programmbenutzer fordert, werden dabei alle max-Tupel, alle n-Tupel oder alle kumulativen n-Tupel (alle Tupel bis zu einer Länge n) berechnet (siehe Kap. 3.1.1). max-Tupel sind alle Exonkombinationen über die gesamte Readlänge. Das heißt, es kann nur durch mehrere mögliche überlappende Annotationen für ein Readintervall zu der Bildung von mehreren Tupeln kommen. Bei n-Tupeln und kumulativen n-Tupeln entstehen verschiedene Tupel, zum einen durch mehrere valide Annotationen und zum anderen durch unterschiedliche Teilmengen der Readintervalle mit ihren zugeordneten Annotationen (Abb. 9, B) z.B. ^E5 und ^E2, E5). Aufgrund der Struktur im *ReadAnnoStore* (siehe Kap. 4.8) kann die Berechnung relativ einfach durch unterschiedliche Kombination der dort gespeicherten Ids in der zweiten Dimension erfolgen.

Damit nur valide Tupel gebildet werden, erfolgt die Tupelbildung anhand der gegebenen Parent-Ids, so dass keine Kinder unterschiedlicher Parents kombiniert werden. Zusätzlich wird für Matepairs äquivalent zu der Bildung des *AnnoCountStores* (siehe Kap. 4.9) geprüft, ob beide Matepairreads in diesem Parent mappen. Ist dies nicht der Fall, so wird das Tupel nicht erstellt. Weil bei der Ermittlung der Parent-Ids bereits geprüft wurde, ob alle Intervalle des Reads in die Parentannotation mappen (siehe Kap. 4.8), können keine unbekannten Bereiche für diese Parent-Ids vorkommen. Somit muss keine Prüfung auf `INVALID_IDs` stattfinden.

Für jedes Tupel der Liste wird geprüft, ob es bereits Bestandteil der entsprechenden *ReadConnections*-Liste ist. Ist dies der Fall, so wird der Count um eins hochgesetzt. Andernfalls wird das Tupel neu eingefügt, wobei eine aufsteigende Reihenfolge beibehalten wird, damit eine schnellere Suche möglich ist.

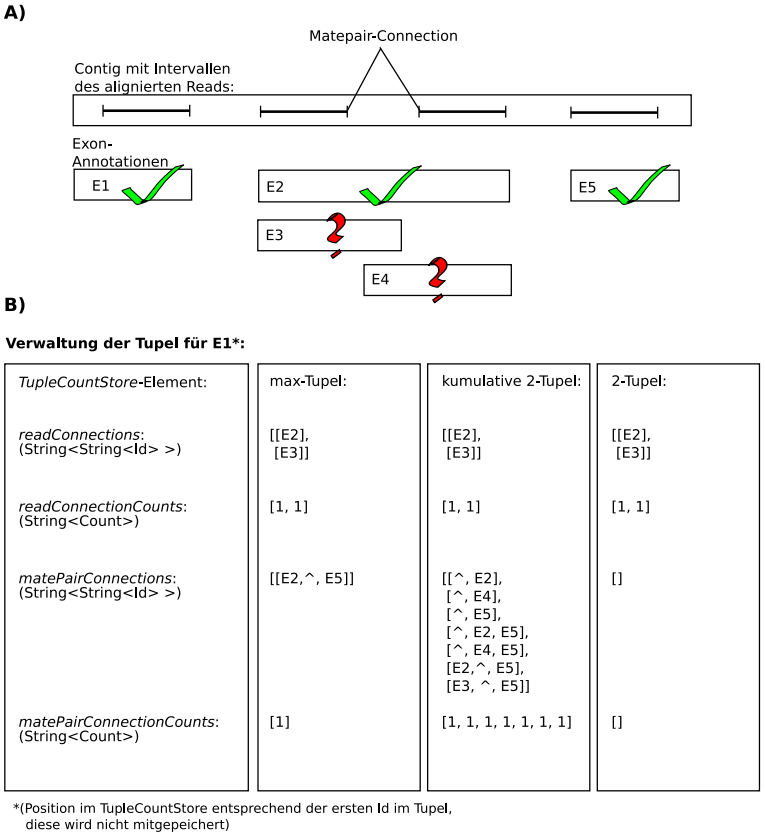

Abbildung 9: Bildung und Verwaltung von Matepair-Tupeln. A) Gegebene Annotationen: Annotation E3 und E4 schließen sich gegenseitig für die Tupelbildung aus. B) Struktur des *TupleCountStores* mit den zugehörigen Einträgen für max-Tupel, n-Tupel und kumulative n-Tupel. Die erste Id (E1) ist implizit durch die Position gegeben und wird nicht gespeichert. Für n ist hier 2 gewählt. Da Exon E3 und E4 nicht miteinander kombiniert werden können, ist in diesem Fall keine Tupelbildung durch Matepairs möglich. Für Exon E2 würde das zweite Vorkommen der Id nicht gespeichert werden, so dass das Tupel nicht mehr die korrekte Länge hätte. Für kumulative 2-Tupel entsteht der Eintrag [E2,ˆ, E5], indem das zweite Auftreten der Id von E2 nicht gespeichert wird. Der Eintrag [ˆ, E2, E5] kommt dadurch zustande, dass Exon E1 von dem ersten Read mit den Exons E2 und E5 vom zweiten Read kombiniert wurde.

Gleichzeitig werden ggf. alle entsprechenden Tupel des zweiten Matepairreads gebildet (siehe Kap. 3.1.2). Bei einer geforderten Länge von n werden dabei auch diese Tupel mit der Länge n gebildet, so dass Tupel, die durch Matepairs entstehen, insgesamt eine Länge von 2n haben. Um zu verhindern, dass diese Berechnungen für die Matepairs doppelt auftreten und um sicher zu stellen, dass die Reihenfolge bzgl. der Contigpositionen korrekt ist, wird vorher geprüft, ob dieser Partner hinter dem aktuellen Read im Contig auftritt. Anschließend werden alle Kombinationen von Tupeln des ersten Reads mit Tupeln (siehe Kap. 3.1.2) des zweiten Reads gebildet und entsprechend in die Liste *MatePairConnections* eingefügt oder der Count um eins hoch gesetzt. Dies verläuft äquivalent zu der Verwaltung der durch einzelne Reads entstehenden Tupeln, nur dass dabei der Übergang zwischen den beiden Reads im String markiert wird. Dadurch bleibt auch später ersichtlich, wie sich diese Tupel zusammengesetzt haben. Zusätzlich wird noch geprüft, ob die Matepair-Verknüpfungen Sinn machen (siehe Kap. 3.1.2). So werden keine Matepairtupel gebildet, welche

im Übergang überlappende Annotationsbereiche enthalten (Abb. 9, z.B. E3 und E4 werden nicht kombiniert). In diesem Fall muss es andere Annotationen geben, da beide Reads bekanntlich von demselben Transkript abstammen. Ebenso wird für den Fall, dass beide Matepairs im selben Annotationsbereich auftauchen, sicher gestellt, dass die zugehörige Id nicht zweimal gespeichert wird (Abb. 9, E2). Erst damit können alle Vorkommen dieser Kombinationen unter ein und demselben Eintrag gespeichert und gezählt werden (z.B. E1-E2^E2-E5 und E1-E2^E5). Bei max-Tupeln wird das zweite Auftreten dieser Annotation-Id nicht gespeichert, so dass auf den Separator "^" direkt die Id des zweiten Exons vom zweiten Read folgt. Für n-Tupel findet keine Verknüpfung der Exons von den einzelnen Reads statt, da die Länge nach der Löschung nicht mehr genau n wäre (z.B. E1-E2^E5 für n = 2). Bei Tupeln bis zu einer Länge n findet ebenfalls keine Verknüpfung statt, weil ansonsten Tupel doppelt auftreten würden (z.B. E1-E2^E5 für n = 1 und n = 2 bei dem zweiten Read).

4.11 Normalisierung

Nachdem die Berechnung der jeweiligen Counts abgeschlossen ist, werden die Ergebnisse zusätzlich normalisiert, um eine allgemein gültige Aussage über die verschiedenen Expressionslevel zu ermöglichen (siehe Kap. 3.1.4). Dafür werden die Counts der gemappten Reads, die Gesamtzahl der verwendeten und gemappten Reads sowie die Längen der Annotationsbereiche benötigt. Die Expressionslevel der einzelnen Exons und der gebildeten Tupel können direkt ermittelt werden. Anhand der gespeicherten Ids werden aus dem *AnnotationStore* die Längen (Abstand zwischen Beginn- und Endposition) und aus dem entsprechenden *Anno-* bzw. *TupleCountStore* die jeweiligen Counts gewonnen. Für die Genexpressionslevel muss zusätzlich eine Map angelegt werden, um die Längen der einzelnen Exonbereiche zwischenzuspeichern. Damit kann zu guter Letzt auf alle Geneinträge zugegriffen und anhand der summierten Längen die normalisierten Expressionslevel bestimmt werden.

4.12 Ausgabe

Die Ausgabe von INSEGT wird entsprechend der internen Verwaltung in drei Teile unterteilt. Als Format wurde dabei passend zur Eingabe GFF gewählt, allerdings gelten hier andere Spezifikationen.

Für die Annotation-Counts ist die Ausgabe gleich der GFF Eingabe der Annotationen, nur dass in Spalte 6 zusätzlich der entsprechende Count und in Spalte 9 der normalisierte Wert für das Expressionslevel angegeben wird (Abb. 10, A)).
Die Ausgabe für die einzelnen Reads besteht aus dem Readnamen, des zugehörigen Sequenznamen, der Orientierung, der gematchten Annotation-Ids und der Parent-Ids (Abb. 10, B)). Hintereinander gematchte Annotationen werden mit einem ":" gekennzeichnet, während überlappende Annotationen durch ein ";" separiert werden. Gibt es mehrere gematchte Parent-Ids, so werden diese und ihre Kinder-Ids hinten angehängt. Sofern eine `UNKNOWN_REGION` unter den Annotationnamen vorkommt, wurde dieser Read für die entsprechende Parent-Id und die Kinder-Ids nicht gezählt. Bei der Tupelausgabe wird zuerst die Sequenz genannt, dann die Parent-Id, die Orientierung mittels " + " oder " - " und dann das entsprechende Tupel, auf welches der Count und das Expressionslevel folgt (Abb. 10, C)). Dabei werden die Verbin-

dungen durch einzelne Reads mit einem ":" und Verbindungen über Matepairs mit " ^ " markiert.

A)

```
<Sequenzname> < > < > <Start> <End> <Count> <Strang> < > \
    <ID=Annotationname;ParentID=Parentname;Expressionslevel (RPKM)>

 chr1          .   .   30      .     60      +        .   ID=P1;30.0;
 chr1          .   .   75      300   20      +        .   ID=E1;ParentID=P1;25.0;
```

B)

```
<Readname> <Sequenzname> <Strang> <Annotationnamen> <Parentname1> \
   [<Annotationnamen>   <Parentname2> ...]

 read_1     chr1          +        E1:E2-1;E2-2:E3   P1            E10:UNKNOWN_REGION  P2
```

C)

```
<Sequenzname> <Parentname> <Strang> <Tupel von Annotationnamen> <Count> <Expressionslevel (RPKM)>

 chr1          P1           +        E1:E2-1:E3^E7               50      30.0
 chr1          P1           +        E1:E2-2:E3^E7               10      11.32
```

Abbildung 10: Ausgaben in GFF-Files. A) Ausgabe der Counts und der Expressionslevel für die einzelnen Annotationsbereiche (hier unbedeutende Felder sind durch ein < > bzw. "." markiert). B) Ausgabe der gematchten Ids und deren Parent-Ids für jeden Read. C) Ausgabe der gebildeten Tupel, deren Counts und Expressionslevel.

4.13 Parametereinstellungen

Um die Berechnung der Ergebnisse von dem Programm INSEGT optimal an die verwendeten Daten anzupassen, können bestimmte Parameter mitübergeben werden. So kann die Anzahl der erlaubten Gaps in den Alignments zwischen Reads und Genom festgelegt werden, für die keine Bewertung als Intron stattfindet (siehe Kap. 4.4). Das spielt z.B. dann eine Rolle, wenn unterschiedliche Genome miteinander verglichen bzw. aligniert werden, so dass deutlich mehr Gaps auftreten. Der Standardwert ist 5. Auch der Wert, um den die Intervalle vor der Suche im Intervallbaum gekürzt werden, kann bestimmt werden (siehe Kap. 4.6). Dieser ist ebenfalls standardmäßig auf 5 eingestellt. Er ist hauptsächlich dafür da, um Artefakte im Alignment auszugleichen und kann damit an dessen Genauigkeit angepasst werden. Gleichzeitig legt dieser bei Exon überspannenden Reads fest, auf wieviele Basen genau der Bereich der Annotation mit dem Bereich des Alignment-Intervalls übereinstimmen muss (siehe Kap. 4.7). Bei der Tupelausgabe kann zusätzlich ein Grenzwert für den Count angegeben werden, ab dem die Tupel ausgeben werden. Damit ist es möglich eventuell falsch positive Resultate auszuselektieren (wenn bei einem großen Readsatz ein Tupel z.B. von nur einem Read abgedeckt wird) oder sich nur die am häufigsten exprimierten Tupel anzeigen zu lassen. Alternativ kann dafür auch ein Grenzwert für die normalisierten Expressionslevel in RPKM angegeben werden (siehe Kap. 4.11). Sind beide Werte angegeben, so findet die Ausgabe nur dann statt, wenn beide Grenzwerte gleichzeitig überschritten werden. Werden diese Parameter nicht an INSEGT übergeben, so wird jedes gebildete Tupel ausgegeben.

Für die Tupelbildung kann angegeben werden, ob max-Tupel, n-Tupel oder alle kumulative n-Tupel erstellt werden sollen (siehe Kap. 4.10). Auf diese Weise wird kein unnötiger Speicherplatz verbraucht, wenn nicht alle Tupel benötigt werden. Als Standardwert sind kumulative 2-Tupel gesetzt.

Zusätzlich muss für den Fall, dass die Orientierungen der Reads nicht bekannt sind, dies INSEGT mitgeteilt werden (siehe Kap. 4.5). Ansonsten werden die Orientierungen aus den eingegebenen Alignments übernommen.

5 Ergebnisse

Die SeqAn SAM Einlesefunktion hat eine quadratische Laufzeit und benötigt bei einer Datensatzgröße von 50000 Reads schon ca. 76 min. Damit war es nicht möglich, Experimente mit größeren Datensätzen durchzuführen.

Aus diesem Grund wurde diese Funktion auch bei allen Laufzeitanalysen von INSEGT ausgeschlossen. Desweiteren wurde das Einlesen der Annotationen nicht berücksichtigt, weil für alle Datensätze das gleiche GFF-File verwendet wurde und die Laufzeit mit ca. 52. sec. konstant ist. Dadurch können die Unterschiede des eigentlichen Programmes INSEGT bei der Bearbeitung von verschiedenen Datensätzen bzw. bei unterschiedlicher Parametereingabe für die Tupelbildung genauer analysiert werden. Weil bei den Ausgaben von INSEGT relativ starke Schwankungen in den Laufzeiten für die einzelnen Datensätze in verschiedenen Durchläufen aufgetreten sind, wurde nur die Laufzeit der Berechnungen selbst ermittelt.

Der verwendete Single-End-Read Datensatz (siehe Kap. 3.5) vom Gehirn hat ursprünglich 34493914 Reads enthalten, von denen 10940940 gemappt sind. Der Paired-End-Read Datensatz bestand ursprünglich aus 3543016 Reads, von denen 7172122 gemappt sind. Für die Experimente wurden aus diesen Datensätzen jeweils kleinere Datensätze mit den ersten 10000, 20000, 30000, 40000 und 50000 Reads erstellt. Auf diesen wurde INSEGT mit Standardparametern laufen gelassen, so dass kumulative Tupel bis zu einer Größe von $n = 2$ gebildet wurden.

Die Abbildung 11 stellt die Ausgabe von INSEGT für die Annotationen von einem Gen mit seinen Exons dar (Single-End-Read Datensatz mit 50000 Reads). Der letzte Eintrag gibt jeweils die normalisierten Expressionslevel in RPKM (siehe Kap. 3.1.4) an. Da nur 50000 Reads verwendet wurden, ist keine qualitative Aussage über die Expression möglich. Aber es wird deutlich, dass der RPKM-Wert von der Anzahl der Counts und von der Länge des Genombereiches abhängt. Deswegen liegt das Expressionslevel für das erste Exon (Abb. 11, 2. Eintrag), das eine Länge von 308 bp und einen Count von 2 hat, bei 129 RPKM. Das Expressionslevel des vierten Exons (Abb. 11, 5. Eintrag) liegt mit demselben Count hingegen bei 360 RPKM, weil es mit 121 bp eine geringere Länge hat.

```
<Sequenzname> < >              < >     <Start>         <End>         <Count> <Strang> < > \
    <ID=Annotationname;ParentID=Parentname;Expressionslevel (RPKM)>

 chr13         Annotation_Count  region  .               .             15      +        .
     ID=ENSG00000136156;22.694185;

 chr13         Annotation_Count  region  47705307        47705614      2       +        .
     ID=ENSG00000136156-1;ParentID=ENSG00000136156;129.870132;

 chr13         Annotation_Count  region  47725945        47726073      0       +        .
     ID=ENSG00000136156-2;ParentID=ENSG00000136156;0.000000;

 chr13         Annotation_Count  region  47728314        47728520      3       +        .
     ID=ENSG00000136156-3;ParentID=ENSG00000136156;289.855072;

 chr13         Annotation_Count  region  47730263        47730373      2       +        .
     ID=ENSG00000136156-4;ParentID=ENSG00000136156;360.360352;

 chr13         Annotation_Count  region  47730934        47731084      1       +        .
     ID=ENSG00000136156-5;ParentID=ENSG00000136156;132.450333;

 chr13         Annotation_Count  region  47733276        47734233      7       +        .
     ID=ENSG00000136156-6;ParentID=ENSG00000136156;56.472500;
```

Abbildung 11: INSEGT Ausgabe für die Annotationen von einem Gen und von seinen Exons (Single-End-Read Datensatz mit 50000 Reads).

Die Datensätze dienen mit ihren unterschiedlichen Größen dazu, die Laufzeit und den Speicherverbrauch von INSEGT abhängig von der Größe der Alignmenteingabe zu testen.

Dabei wurde für diesen Teil der Mittelwert von den Laufzeiten aus 10 Durchläufen gebildet, so dass der Einfluss von INSEGT unabhängigen Faktoren auf die Ergebnisse minimiert wurde. Die Abbildung 12 (links) zeigt deutlich, dass die Laufzeit wie erwartet mit der Größe der Datensätze linear ansteigt. Der Single-End-Read Datensatz der Größe 10000 benötigt 6,38 sec.. Diese Zeit steigt linear, so dass bei einer Größe von 50000 Reads 8,72 sec. benötigt werden. Für Paired-End-Read Datensätze liegt die Laufzeit minimal über der Laufzeit von Single-End-Read Datensätzen und steigt etwas schneller an (Abb. 12, links). So ist diese für 10000 Reads nur 0,03 sec. und bei 50000 Reads 0,06 sec größer. Da die einzelnen Reads keine Splicejunctions überspannen und somit auch keine Tupel aus über Read verknüpften Exons entstehen, liegt der einzige und entscheidene Unterschied bei diesen Datensätzen darin, dass für die Paired-End-Reads Matepairtupel der Größe 2 gebildet werden. Weil mit steigender Readanzahl zusätzlich die Anzahl der Matepairtupel steigt, steigt die Laufzeit für Paired-End-Reads auch schneller an.

Der Speicherverbrauch steigt ebenfalls linear mit der Größe der Datensätze (Abb. 12, rechts). Für Single-End-Reads liegt der Speicherverbrauch für 10000 Reads bei 326 MB und steigt für 50000 Reads auf 378 MB. Der Grund dafür liegt zum einen darin, dass die eingelesenen Alignments selbst im *AlignedReadStore* gespeichert werden müssen. Zum anderen wächst damit proportional die Größe des *AlignIntervalsStores* und des *ReadAnnoStores* (siehe Kap. 4.2). Der Paired-End-Read Datensatz benötigt allgemein bei gleicher Größe mehr Speicher (Abb. 12, rechts). Er verbraucht mit einer Größe von 10000 Reads 328 MB, d.h. 2 MB mehr als der Single-End-Read Datensatz. Mit einer Größe von 50000 Reads werden 389 MB benötigt, also bereits 11 MB mehr im Vergleich zum Single-End-Datensatz. Der Grund dafür liegt wieder in den Matepairtupeln.

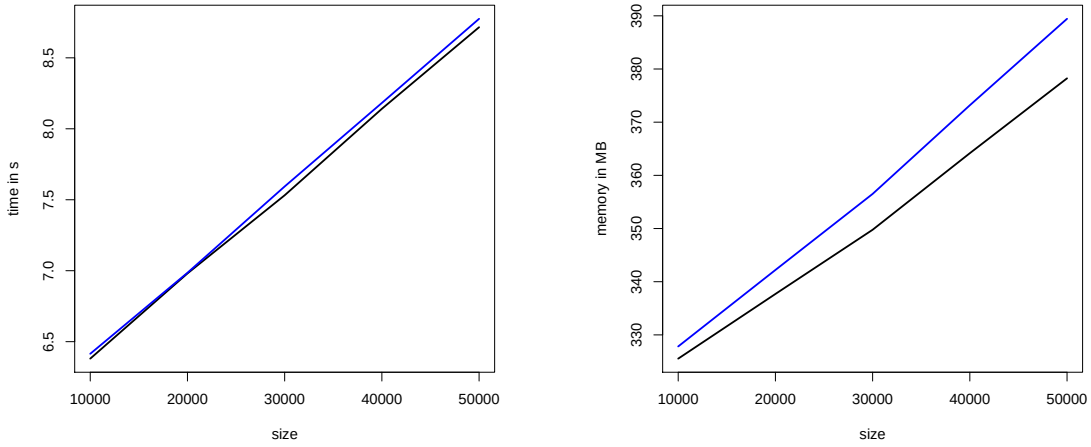

Abbildung 12: Linke Abb.: Laufzeiten für Gehirn-Datensätze mit unterschiedlichen Größen für Single-End-Reads (schwarze Linie) und für Paired-End-Reads (blaue Linie). Rechte Abb.: Zugehöriger Speicherverbrauch.

Der Datensatz mit den Transkripten hat eine Größe von 591, wobei alle Transkripte gemappt sind. Die Größe wurde für alle Experimente beibehalten. INSEGT wurde für die Bildung von n-Tupeln laufen gelassen, wobei nacheinander eine Tupelgröße von $n = 2, 3, \ldots, 10$ gewählt wurde. Ansonsten wurden Standardparameter verwendet (siehe Kap. 4.13). Zusätzlich dazu wurde einmal die Bildung von max-Tupeln gefordert.

Bei den Experimenten mit den Transkripten sind in der Ausgabe für die Reads die aufeinander folgenden gemappten Annotationen gut zu erkennen (Abb. 13, B)). Da es sich bei diesem Datensatz aber nicht um richtige Reads sondern um heruntergeladene Transkripte aus der aktuelleren Datenbank ENSEMBL Version 53 handelt, liefern diese Ergebnisse eine Aussage über die Unterschiede im Vergleich zum GFF-File von ENSEMBL Version 46. Unbekannte Bereiche sind durch `UNKNOWN_REGION` gekennzeichnet. Das heißt in diesem Fall, dass es für das jeweile Transkriptintervall keine passende Annotation in der ENSEMBL Version 46 gibt. In diesem Experiment konnte so z.B. für das Gen *RABLP1* gezeigt werden, dass es für das letzte Exon der eingelesenen Transkripte ENST00000019317 und ENST00000383432 kein übereinstimmendes Exon in ENSEMBL 53 gibt. Die Abbildung 13 A), welche die Transkripte von den verschiedenen ENSEMBL Versionen darstellt, bestätigt, dass das letzte Exon in Version 53 deutlich länger ist, als das letzte Exon in Version 46. Dementsprechend mappt INSEGT das letzte Transkriptintervall nicht in das kürzere Exon aus den gegebenen Annotationen. Zusätzlich konnte für das 9. Exon der Transkripte aus Version 46 gezeigt werden, dass seine Annotation nicht mit denen aus Version 53 übereinstimmt. Das liegt daran, dass es dieses Exon in ENSEMBL 53 gar nicht gibt (Abb. 13, A)). Außerdem wurde deutlich, dass die beiden Transkripte sich zudem in ihrem ersten Exon unterscheiden. Das erste Exon von dem ersten Transkript ENST00000019317 stimmt mit der Annotation aus ENSEMBL 46 überein. Das erste Exon von dem zweiten Transkript ENST00000383432 hingegen mappt in keine gegebene Annotation aus ENSEMBL 46, was durch die `UNKNOWN_REGION` in der Ausgabe zu sehen ist (Abb. 13, B)).

Für die Laufzeitanalyse wurde wieder der Mittelwert aus 10 Durchläufen ge-

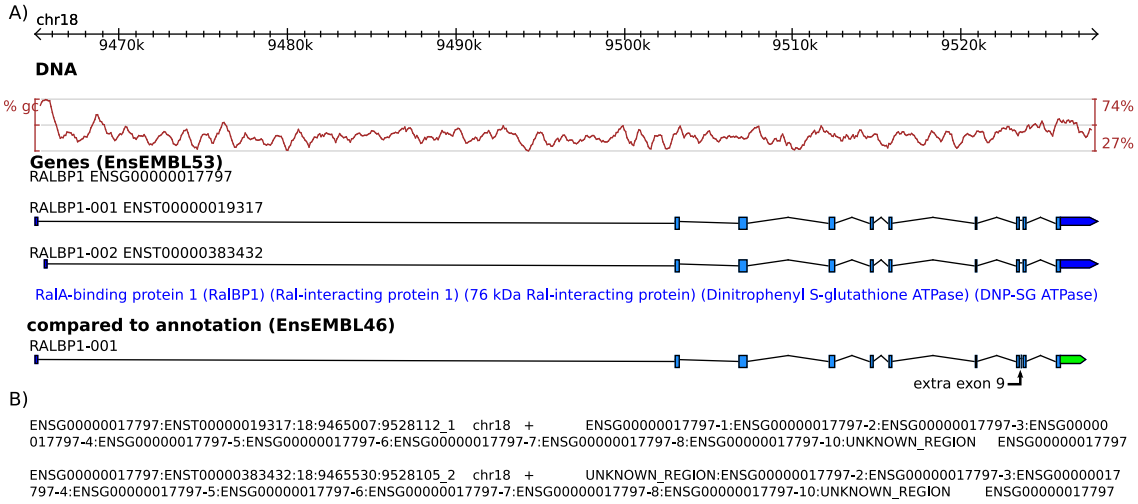

Abbildung 13: A) Die Abbildung zeigt die beiden Transkripte ENST00000019317 und ENST00000383432, die zum *RABLP1* Gen gehören (obere Zeile ENSEMBL Version 53). Die Annotation, mit der verglichen wurde, basiert auf den Transkripten die in ENSEMBL 46 für das *RABLP1* Gen gespeichert sind. Dabei ist in dieser älteren ENSEMBL Version 46 noch ein zusätzliches Exon 9 annotiert und außerdem ist das letzte Exon (hier in grün, nicht skalengetreu) 516 bp kürzer. B) INSEGT-Ausgabe für die Reads (hier Tranksripte): In der 4. Spalte stehen die Namen der gemappten Exons. Die Zahl nach dem Bindestrich gibt hier jeweils die Nummer des Exons im Gen an (ENSEMBL 46). Die `UNKNOWN_REGION`s zeigen die Transkriptintervalle, die nicht mit Annotationen der ENSEMBL Version 46 übereinstimmen. Exon 8 und Exon 10 aus Version 46 stimmen mit denTranskriptintervallen überein. Da keine `UNKNOWN_REGION` zwischen diesen in der Ausgabe enthalten ist, wird deutlich, dass Exon 9 in der aktuelleren Version 53 nicht mehr enthalten ist. Außerdem zeigt bei dem zweiten Transkript die erste `UNKNOWN_REGION`, dass es keine passende Annotation für das erste Transkriptintervall gibt.

bildet. Weil für jedes Gen und somit für jedes Tupel höchstens zwei Transkripte vorliegen, muss für die meisten gebildeten Tupel (mind. für jedes zweite Tupel) ein neuer Eintrag im *TupleCountStore* getätigt werden. Dadurch wird die Laufzeit und der Speicherverbrauch hauptsächlich davon beeinflusst, wie viele Tupel insgesamt berechnet werden müssen.

Die Laufzeiten für die unterschiedlichen Tupel liegen zwischen 6,015 und 6,035 sec., d.h. sie verändern sich kaum (Abb. 14, links). Da die Abweichungen in einem Rahmen von nur 0,02 sec. liegen, wirken sich externe Einflüsse zu stark aus, um die geringen Unterschiede abhängig von der Tupelgröße zu messen. Für größere Datensätze wird erwartet, dass die Laufzeiten für max-Tupel relativ gering sind, weil für jeden Read nur einmal über die gesamte Readspanne Tupel gebildet werden müssen. Für n-Tupel hingegen müssen für den Fall, dass der Read insgesamt mehr als n Exons abdeckt, alle möglichen Tupel für alle n Exon großen Ausschnitte ermittelt werden (siehe Kap. 4.10). Die Anzahl dieser Ausschnitte sinkt für größer werdende n-Tupel, so dass dementsprechend auch die Laufzeiten sinken werden. Hinzukommt, dass es für große n-Tupel nicht mehr so viele Reads gibt, die mindestens n Exons abdecken und für die eine Tupelbildung stattfinden muss.

Der Speicherverbrauch verhält sich dementsprechend (Abb. 14, rechts). Die Bil-

dung von max-Tupel hat mit 316,23 MB den geringsten Speicherverbrauch, weil entsprechend wenige Tupel entstehen. Bei Tupeln der Länge 2 wird mit 316,94 MB am meisten Speicher benötigt. Weil der Speicherbedarf hauptsächlich von der Anzahl der zu berechnenden Tupel abhängt, sinkt dieser fast linear bis zu einer Tupelgröße von 10, wo er 361,46 MB beträgt.

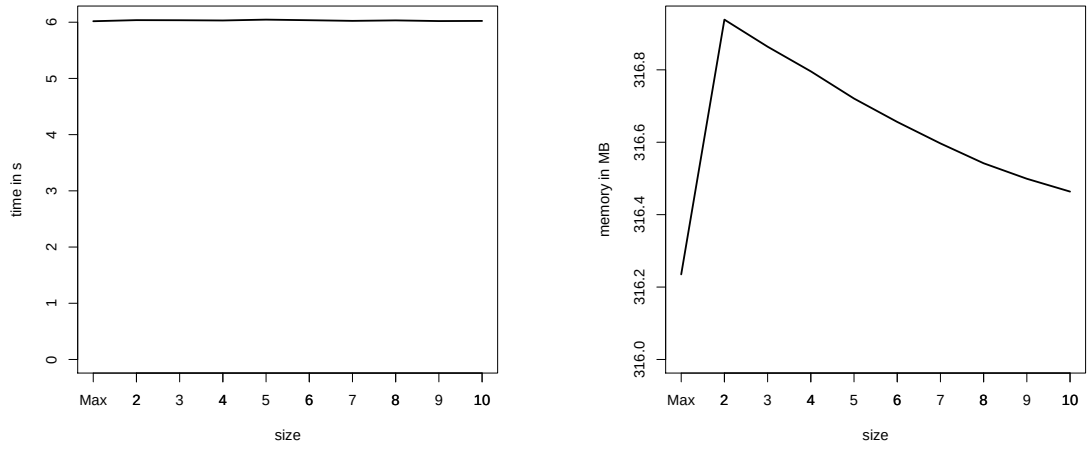

Abbildung 14: Linke Abb.: Laufzeiten für den Transkript-Datensatz für die Bildung von max-Tupeln und n-Tupeln ($n = 2, \ldots, 10$). Rechte Abb.: Zugehöriger Speicherverbrauch.

6 Diskussion

Ursprünglich war geplant, ein Experiment mit Gehirn- und Leber-Datensätzen der Größe von 10 Mio. Reads durchzuführen. Damit wären genauere Aussagen über die verschiedenen Expressionslevel möglich gewesen. Mittels des RPKM Wertes (siehe Kap. 4.11) hätte man anschließend die Expressionslevel in den unterschiedlichen Gewebetypen miteinander vergleichen können. Allerdings hat die SAM Einlesefunktion eine quadratische Laufzeit und ist somit zur Zeit der limitierende Faktor von INSEGT bzgl. der Anzahl der verwendeten Alignments. Aus diesem Grund konnten keine Datensätze in dieser Größe eingelesen werden. Eine Optimierung dieser Funktion ist bereits geplant, so dass in Zukunft auch größere Alignmentsätze mit INSEGT bearbeitet werden können. INSEGT wird Teil der RAPID Pipeline für RNA-Seq Datenanalyse werden (http://cmb.molgen.mpg.de/2ndGenerationSequencing/RAPID/).

Durch die Option, dass sowohl max-Tupel, n-Tupel als auch alle Tupel bis zu einer Länge n gebildet werden können, kann die Ausgabe von INSEGT optimal an die Anforderungen des Benutzers angepasst werden. Die Berechnung des vollen Umfangs ist vor allem dann sinnvoll, wenn Informationen darüber benötigt werden, welche unterschiedlichen Transkripte eines Gens vorkommen, die Reads aber nicht das ganze Transkript abdecken. Das Problem dabei ist, dass bei mehreren gegebenen n-Tupeln nicht eindeutig darauf zurück geschlossen werden kann, aus welchen Transkripten diese entsprungen sind. Bei uneingeschränkter Tupelgröße (für kumulative Tupel) ist dies zwar aufgrund der Einschränkung durch die Readgröße oft auch nicht möglich, jedoch können dabei weitaus mehr Informationen über größere Teilbereiche der

Transkripte gegeben werden. Besonders in Verbindung mit Matepair-Tupeln kann auf diese Weise die Informationsdichte maximiert werden. Der entscheidene Nachteil für dieses Verfahren ist, dass für die Verwaltung der verschiedenen Kombinationen deutlich mehr Speicherkapazität benötigt wird und die Laufzeit ansteigt. Allerdings kann dies gerade für die Zukunft bei steigender Readlänge von großer Bedeutung sein. Die Beschränkung auf n-Tupel ist vor allem dann sinnvoll, wenn spezifische Informationen über die Expression von einzelnen Exon-Verbindungen benötigt werden.

Durch bestimmte Erweiterungen von INSEGT wäre zukünftig auch die Anwendung für hier nicht beschriebene Daten möglich. Ein Beispiel dafür ist die Expressionsanalyse von Fusionsgenen, welche durch Translokationen entstehen. Diese spielen u.a. bei der Entstehung der akuten myeloischen Leukämie eine bedeutende Rolle. Momentan sind in INSEGT Alignments auf ein Chromosom beschränkt. Es wäre jedoch möglich, die interne Verwaltung der Alignments so zu erweitern, dass ein Read nacheinander auf Intervalle von verschiedenen Chromosomen matchen kann. Dafür müssten der *AlignedReadStore*, der *ReadAnnoStore* sowie der *TupleCountStore* um die Verwaltung für mehrere Contigs ergänzt werden. Für die Suche nach passenden Annotationen könnte für alle Chromosome ein gemeinsamer Intervallbaum erstellt werden. Damit wäre durch die Tupelbildung die Aufdeckung und Expressionslevelmessung von bisher unbekannten Fusionsgenen möglich. Die Tupelbildung müsste dafür lediglich so abgewandelt werden, dass auch Exons mit unterschiedlicher Contig-Id kombiniert werden dürfen. Gleichzeitig könnten bekannte Fusionsgene mittels GFF eingelesen und deren Expression analysiert werden.

Literaturverzeichnis

[1] Benjamin J Blencowe. Alternative splicing: new insights from global analyses. *Cell*, 126(1):37–47, Jul 2006.

[2] Fabio De Bona, Stephan Ossowski, Korbinian Schneeberger, and Gunnar Rätsch. Optimal spliced alignments of short sequence reads. *Bioinformatics*, 24(16):i174–i180, Aug 2008.

[3] Volker Brendel, Liqun Xing, and Wei Zhu. Gene structure prediction from consensus spliced alignment of multiple ESTs matching the same genomic locus. *Bioinformatics*, 20(7):1157–1169, May 2004.

[4] Anthony Cox. Unpublished software.

[5] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, 9:11, 2008.

[6] Anne-Katrin Emde. Segment-Based Progressive Alignment of Genomic Sequences. Master's thesis, Freie Universitaet Berlin, Germany, 2007.

[7] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, Aug 2009.

[8] Tom Maniatis and Bosiljka Tasic. Alternative pre-mRNA splicing and proteome expansion in metazoans. *Nature*, 418(6894):236–243, Jul 2002.

[9] Ali Mortazavi, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. Mapping and quantifying mammalian transcriptomes by RNA-Seq. *Nat Methods*, 5(7):621–628, Jul 2008.

[10] R. Mott. EST_GENOME: a program to align spliced DNA sequences to unspliced genomic DNA. *Comput Appl Biosci*, 13(4):477–478, Aug 1997.

[11] Ugrappa Nagalakshmi, Zhong Wang, Karl Waern, Chong Shou, Debasish Raha, Mark Gerstein, and Michael Snyder. The transcriptional landscape of the yeast genome defined by RNA sequencing. *Science*, 320(5881):1344–1349, Jun 2008.

[12] Guy St C Slater and Ewan Birney. Automated generation of heuristics for biological sequence comparison. *BMC Bioinformatics*, 6:31, 2005.

[13] Marc Sultan, Marcel H Schulz, Hugues Richard, Alon Magen, Andreas Klingenhoff, Matthias Scherf, Martin Seifert, Tatjana Borodina, Aleksey Soldatov, Dmitri Parkhomchuk, Dominic Schmidt, Sean O'Keeffe, Stefan Haas, Martin Vingron, Hans Lehrach, and Marie-Laure Yaspo. A global view of gene activity and alternative splicing by deep sequencing of the human transcriptome. *Science*, 321(5891):956–960, Aug 2008.

[14] Cole Trapnell, Lior Pachter, and Steven L Salzberg. TopHat: discovering splice junctions with RNA-Seq. *Bioinformatics*, 25(9):1105–1111, May 2009.

[15] Eric T Wang, Rickard Sandberg, Shujun Luo, Irina Khrebtukova, Lu Zhang, Christine Mayr, Stephen F Kingsmore, Gary P Schroth, and Christopher B Burge. Alternative isoform regulation in human tissue transcriptomes. *Nature*, 456(7221):470–476, Nov 2008.

[16] David Weese, Anne-Katrin Emde, Tobias Rausch, Andreas Döring, and Knut Reinert. RazerS–fast read mapping with sensitivity control. *Genome Res*, 19(9):1646–1654, Sep 2009.