Bachelorarbeit

Svenja Mehringer

September 10, 2015

Contents

1 Zusammenfassung

abstract...

2 Einleitung

2.1 Hintergrund

Genomische DNS ist heutzutage Gegenstand vieler Forschungsarbeiten. Ein Genom umfasst das gesamte Erbgut eines Organismus' in Form einer einzelnen, fortlaufenden Sequenz der vier Basen Adenin, Cytosin, Guanin und Thymin. Bei den meisten Organismen wird die Sequenz in ihre jeweiligen Chromosomen unterteilt.

Die naivste und häufigste Form der Speicherung einer solchen genomischen DNS-Sequenz, ist das textbasierte FASTA Format. Es speichert die Sequenz (A,C,G,T entsprechend der Basen), sowie einen vorangestellten Namen oder Kommentar, als Buchstaben im ASCII-Code(/Format?) ab. Ein ASCII-kodierter Buchstabe verbraucht ein Byte. Vernachlässigt man den geringen Einfluss des Sequenznamen, ist die Größe der gespeicherten Datei damit proportional zur Länge der Sequenz. Am Beispiel eines Humangenoms mit einer Gesamtlänge von ca. 3 Milliarden Basenpaaren (haploid) ergibt sich eine Dateigröße von 3GB (Gigabyte). Untersucht man in einer Studie 1,000 Genome von Patienten, führen dadurch allein die reinen Sequenzdaten ohne Qualitätsinformation zu einem Speicheraufkommen von 3TB (Terrabyte). Obwohl bis jetzt erst wenige Studien eine solche Vielzahl an Sequenzierungen realisieren konnten, werden in Zukunft immer mehr Projekte wie das *1000 Genome Preject*[3] erwartet. Der Grund dafür ist, dass es mit Hilfe von Next-Generation-Sequencing (NGS) in den letzten Jahren für viele Firmen und Institutionen immer erschwinglicher geworden ist, DNS-Proben zu sequenzieren (Grafik 1).

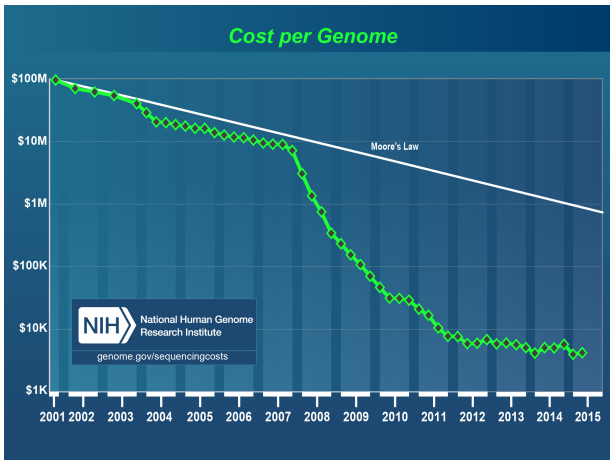

Figure 1: Trend der Genomesequenzierungskosten (NHGRI). Neben den Kosten zeigt die Grafik das sogenannte Mooresche Gesetz. Es beschreibt den langzeit Trend in der Computer Hardware Industrie. Die plötzlich einsetzende Entfernung der sinkenden Kosten vom Mooreschen Gestz im Jahr 2007 wird mit dem Übergang von Sanger-bastierten Sequenzierung zu 'second generation' oder 'next-generation'-Technologien erklärt [16].

Zwar sinken ebenfalls die Kosten für die Speicherung, allerdings ist der Fortschritt in der Sequenzierungstechnologie dem der Hardware Industrie weit voraus [16]. Das Resultat sind immer größere Mengen an produzierten Daten. Sie stellen nicht nur für die langfristige und sichere Speicherung eine Herausforderung dar, sondern auch für deren Prozessierung und Analyse. Je größer die Daten desto höher sind die Anforderungen an die Ausstattung des benutzten Computers. Oft kann dadurch nur mit sehr hochwertigen, kostspieligen *'high-end'* Computern gearbeitet werden. Wer keinen besitzt, muss oft auf externen Servern arbeiten, was wiederum zu einem zeitaufwändigen Datenransfer führt. Um diesen Problemen entgegenzuwirken, ist ein vielversprechender Lösungsansatz die Kompression der Daten. Da Sequenzdaten generell textbasiert gespeichert werden, finden die Grundlagen der Kompression von Texten Anwendung.

Ziel der Kompression ist es, eine Datei in einer Art und weise zu Verändern, bzw. zu Kodieren, sodass die dabei entstehende Version weniger Speicherplatz (Byte) als vorher benötigt. Ist die Kompression verlustfrei, kann die Originaldatei ohne Informationsverlust wieder vollständig dekodiert werden. Falls kleine, statisch nicht signifikante Unterschiede toleriert werden können, werden in manchen Fällen verlustreiche Methoden der Kompression aufgrund der Effizienz bevorzugt. Da bei DNS-Sequenzen schon kleinste Unterschiede von großer Bedeutung sein können (z.B. Single Nucleid Polymorphism (SNP)), wird eine verlustreiche Kompression vermieden und soll in dieser Arbeit nicht weiter thematisiert werden. Grundsätzlich unterscheidet man zwischen bit-manipulierender, statistischer und Wörterbuch-basierter Kompression.

Die Idee der Bit-Manipulation ist es zwei oder mehr Zeichen als mit Hilfe von Bit-Kodierung in einem einzigen Byte zu speichern. Durch verschiedene Kombinationen und limitierte Alphabete können so Kompressionsraten von 1:4 erreicht werden [19]. Statistische Methoden entwickeln Modelle, welche das nächste Zeichen voraussagen oder die abgeleiteten Häufigkeiten für eine sinnvolle Kodierung verwenden. Wörterbuch-basierte Methoden hingegen benutzen Substitution, um wiederholt auftretende Teilsequenzen durch einen Zeiger auf einen entsprechenden Wörterbucheintrag zu ersetzen.

Einer der am häufigsten als Basis benutzte Wörterbuch-basierten Algorithmen ist der Lempel-Ziv-Welch-Algorithmus (LZW) [25]. Er komprimiert verlustfrei jedes Format, da das dynamisches Wörterbuch erst bei Start des Verfahrens aufgebaut wird. Eine weitere, sehr wichtige Methode, ist die Huffman-Kodierung [11]. Sie weist einer festen Anzahl von Symbolen eine Bit-Codierung (Codewort) mit variabler Länge zu. Dabei bekommen häufig auftretende Symbole die kürzesten Codewörter, um so die Eingabe mit optimaler (kleinster) Länge zu kodieren. Oft werden beide Algorithmen kombiniert ([12]).

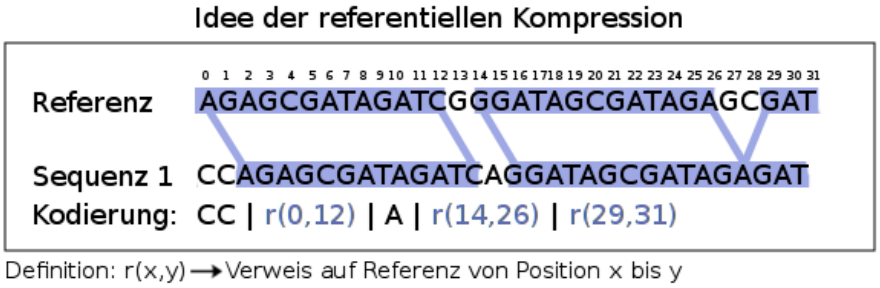

Figure 2: Idee der referentiellen Kompression.

Im Zuge der immer größeren Nachfrage nach speicherplatzeffizienten Formaten für DNS, wurde noch eine weitere Art der Kompression entwickelt.

Die *referentielle Kompression* geht auf die spezifischen Eigenschaften von von DNS ein. Ähnlich der Wörterbuch-basierten Variante, werden bei der referentiellen Kompression ebenfalls Zeiger verwendet. Sie verweisen allerdings auf Teile einer Referenzsequenz anstatt eines Wörterbucheintrags. Man macht sich hierbei die Tatsache zu Nutze, dass, besonders innerhalb einer Spezies, die Ähnlichkeit zwischen Sequenzen sehr hoch ist. Beispielweise unterscheiden sich statistisch gesehen im durchschnitt zwei zufällige Humangenome um nur ca. 0,1% [21]. Folglich kann man 99,9% der Sequenz als Zeiger auf eine Referenz kodieren und sollte somit eine sehr hohe Kompression erreichen können. Der Nachteil dieses Verfahrens ist die starke Abhängigkeit von der verwendeten Referenz. Vergleicht man ein Humangenom mit z.B. einem Mausgenom würde man nur eine geringe Kompression erreichen können. [22] schätzt die Länge von gleichen Teilsequenzen zwischen Maus und Mensch auf nur ca. 20-25 Basen, wodurch eine Kodierung ähnlich der Grafik 2, mehr Speicherplatz als vorher verbrauchen würde. Folglich ist die referentielle Kompression weitestgehend für sogenannte Resequenzierungs-Projekte (*resequencing*) geeignet, welche sich auf die Sequenzierung von Genomen innerhalb einer Spezies konzentrieren.

Es existieren bereits einige Algorithmen und Programme mit verschiedenen Ansätzen für die referentiellen Kompression von DNS. Methoden die auf die Kompression von Rohdaten in Form von *reads* abzielen ([6]) werden vernachlässigt, da sie trotz ähnlicher Problemstellung nur sehr gering auf das Ziel dieser Arbeit übertragbar sind. Der Fokus liegt auf Algorithmen für die Kompression von ganzen Genomen bzw. Genomsammlungen. Erste erfolgreiche Resultate konnten 2008 von [2] verzeichnet werden: Das James Watson Genom wurde von 3.1 GB auf lediglich 4.1 MB komprimiert. Allerdings wurde hierbei das Wissen von genauen Unterschieden, namentlich SNP's und kurze Indel's, zum

Table 1 Summary of the most important compressors of sequencing data

Software name	Implementation availability src code / binaries / libs	Website	Lossless / lossy	Ambig. codes	Var. length reads	Speed of compr./ decompr.	Ratio	Random access	Methods	Remarks
Compressors of genome collections										
gzip	C++ / many / many	www.gzip.org	yes / no	yes	N/A	low / very high	very low	no	LZ, Huf	
bzip2	C / many / many	www.bzip.org	yes / no	yes	N/A	low / high	very low	no	BWT, Huf	
7z	C, C++ / many / many	www.7-zip.org	yes / no	yes	N/A	low / very high	high	no	LZ, AC	chr-ordered
ABRC [38]	C++ / Lin, Win / C++	www2.informatik.hu-berlin.de/~wandelt/blockcompression/	yes / no	yes	N/A	high / very high	very high	yes	LZ, Huf	
GDC [39]	C++ / Lin, Win / C++	sun.aei.polsl.pl/gdc	yes / no	yes	N/A	high / very high	very high	yes	LZ, Huf	
GReEn [40]	C / – / –	ftp://ftp.ieeta.pt/~ap/codecs/	yes / no	yes	N/A	high / high	high	no	M. models, AC	
GRS [41]	C / Lin / –	gmdd.shgmo.org/Computational-Biology/GRS/	yes / no	yes	N/A	moderate / low	high	no	LCS, Huf	
RLZ [42]	C++ / – / –	www.genomics.csse.unimelb.edu.au/product-rlz.php	yes / no	yes	N/A	moderate / very high	high	no	LZ, Gol	

Abbreviations used in the table: src—source codes, libs—libraries, Lin—Linux, Win—Windows, Pyt—Python, exe—binary executables, AC—arithmetic coding (a statistical coding method [12]), CM—context mixing for arithmetic coding [12], diff—differential coding (paradigm: store only changes between sequences), Gol—Golomb (a statistical coding method [12]), Huf—Huffman, LCS—longest common subsequence (a measure of similarity of sequences [43]), LZ—an algorithm from Ziv–Lempel family, M. models—Markov models [12], PPM—prediction by partial matching (an efficient general-purpose compressor [12]). "Ambig. codes" means the ability to compress DNA symbols other than {*A, C, G, T, N*}. "chr-ordered" for 7z and genome collections means that the input (human) genomes were split into chromosomes and ordered according to them before the actual compression. In this way several chromosomes fit the 7z LZ-buffer which is beneficial for the compression.

Figure 3: Zusammenfassung der wichtigsten Kompressoren von Sequenzdaten

Referenzgenom vorausgesetzt. Das heißt, die Eingabe für den Algorithmus ist das durch *Variant Calling* vorprozessierte VCF-Format (*Varaint Calling Format*). Ebenfalls auf VCF-Basis erzielte [4] die bisher höchste höchste Kompressionsrate von ca 1:15000, auf den Daten des *1000 Genome Projects*.

Das Problem des Variant Calling ist jedoch äußerst schwierig, denn es erfordert ein perfektes globales Alignment zum entsprechenden Referenzgenom. Häufig kann diese nötige Voraussetzung nicht erfüllt werden. Aus diesem Grund muss zu den Algorithmen, welche auf reinen Sequenzdaten (FASTA-Format oder ähnliches) arbeiten können, unterschieden werden. Eine Übersicht dieser Algorithmen bis 2013 wird in dem Bericht [8] aufgestellt (Grafik 3 – gekürzte Originaltabelle).

Die Programme gzip, bzip2 und 7z (Referenzen notwendig?), sind allgmeine, textbasierte Kompressoren, deren Kompression, unter Ausnahme von 7z, sehr gering eingeschätzt wird. Tatsächlich erzielt das sehr verbreitete Programm gzip lediglich eine Rate von ca. 1:4 (???nachgucken referenz). Dagegen erzielen die speziell für genomische DNS entworfenen Programme deutlich bessere Ergebnisse. Die zwei Algorithmen GRS [24] und GreEn [17] verzeichnen Kompressionsraten für einzelne Humangenome von ca. 1:200. Während GRS als Basis das Prinzip der *Longest Common Subsequence* mit anschließender Huffman-Codierung benutzt, verwendet GreEn einen statischen Ansatz mit Hilfe Markov-Modellen. Ein Ansatz mittels der Lempel-Ziv Methode LZ77 wird von [14] vorgeschlagen (RLZ) und erreicht ähnliche Kompressionraten von 1:250. Ein weiterer Kompressionsalgorithmus (ABRC) wurde von Wandelt und Leser angeboten, der einen komprimierten Suffixbaum für die Suche von Gemeinsamkeiten blockweise aufbaut [23]. Durch anschließende Lempel-

Ziv Transformation und Huffman-Kodierung erreicht das Programm ABRC Kompressionsraten von 1:400 auf Daten des *1000 Genome Projects*. Eine noch höhere Kopressionsrate erreichte der Algorithmus GDC (Rate ca. 1:600) bzw. GDC Ultra (Rate ca. 1:2000) [7], gefolgt von GDC 2 (Rate ca. 1:9000) [5]. Ähnlich dem Programm RLZ basiert der GDC/GDC-Ultra/GDC 2 Algorithmus auf LZ77, allerdings mit einem wichtigen Unterschied zu anderen Methoden. Alle vorherigen Methoden komprimieren auch bei einer Sammlung von Genomen jedes einzelne unabhängig. Im Gegensatz dazu, werden bei GDC mehrere Sequenzen als Referenz genutzt. Die zusätzliche Zusammenführung von Informationen zwischen den zu komprimierenden Sequenzen, stellt sich als sehr vielversprechende Idee da.

Über die Anstrengungen zur Reduzierung des Speicherplatzes und des Datentransfers hinaus, muss auch die zeitaufwändige Analyse solcher Datenmengen beachtet werden. Alle bisher angesprochenen Algorithmen, dienen lediglich der Kompression von Daten hinsichtlich ihrer Speicherung. Will man die Daten analysieren, müssen sie wieder dekodiert werden. Die Sequenzen bzw. Genome würden zudem einzeln und nacheinander untersucht werden. Man verwirft dabei die vorher zur Kompression benutzte Information der hohen Ähnlichkeit zwischen Sequenzen, was zur überflüssigen Analyse von redundanten Teilsequenzen führt. Es liegt folglich nahe, die referentielle Kompression auch auf die Analyse zu erweitern. Diese Problemstellung wird von [15] als *compressive genomics* benannt.

Die erste Möglichkeit, verschiedene sequentielle Algorithmen auf mehreren Sequenzen gleichzeitig laufen zu lassen, wird von [18] vorgestellt. Durch eine unterliegende Datenstruktur, als JST (*journaled string tree*) bezeichnet, werden Sequenzen als eine Menge von Varianten, basierend auf einem gemeinsamen Koordinatensystem repräsentiert. Eine einzelne Sequenz kann wiederum aus ihren zugehörigen Varianten, als referentiell komprimierter String (*journaled string*), generiert werden. Durch die referentielle Kompression von Sequenzen zu Varianten, werden gemeinsame Bereiche nur einmal analysiert und die Laufzeit wird um den Faktor 115 Beschleunigt [18].

2.2 Motivation

Die Arbeit beschäftigt sich mit der Implementierung eines C++ Programms innerhalb der SeqAn Bibliothek [9], welches der verlustfreien referentiellen Kompression von DNS-Sequenzen dienen soll. Sie steht in engem Zusammenhang mit dem SeqAn Projekt des JST [18] und ist auf dessen Kompatibilität abgestimmt.

Derzeit können zur Verwendung des JST nur Dateien im VCF-Format benutzt werden. Das Hauptinteresse liegt daher nicht nur in der bestmöglichen Speicherplatzreduzierung

einer Datei, sondern in der möglichst effizienten Transformation und Speicherung der Eingabedaten (FASTA-Format o.ä.) in einem für den JST weiterverwendbares Format.

Der vorgestellte Algorithmus soll lediglich die Unterschiede zur Referenz kodieren und dabei die Idee der Zusammenführung von Informationen zwischen den zu komprimierenden Sequenzen von [7, 5] aufgreifen. Da viele Autoren auf die Abhängigkeit der Kompression von der verwendeten Referenzsequenz hindeuten [8, 22, 13], ist eine Besonderheit gegenüber anderen Arbeiten, dass die verwendete Referenzsequenz variabel sein soll. Sie wird bei jeder Eingabe neu angelegt und kann fortlaufend verändert werden, mit dem Ziel, das Speicherformat weiter zu optimieren.

3 Methoden

3.1 Definitionen und allgemeine Notation

Ein **String** $s = s_0....s_{n-1}$, $s_i \in \sum$, ist eine Sequenz der Länge $|s| = n$, auf Basis des Alphabets $\sum$. Ein Teilstring von s, von Position i bis j, wird als $s[i,j] = s_i....s_{j-1}$ angegeben. Falls $i \geq j$, wird die Rückgabe als leerer String definiert. Der Spezialfall $s[i,\ i+1]$ wird zu $s[i]$ abgekürzt.

Ein **Journaled String** verhält ich wie ein normaler *String* (Notation s.o.), mit dem Unterschied, dass er aus zwei Datenstrukturen besteht. Die Erste, der *'Holder'*, speichert die Referenzsequenz. Die Zweite speichert Veränderungen mittels eines *journal tree* und eines *insertion buffers* [referenz rene].

Ein **Journal Entry** ist ein Eintrag im *journal tree* eines *journaled string* der eine Veränderung der Referenzsequenz beschreibt. Für mehr Details siehe [referenz rene].

Ein **Seed** ist ein Tupel $seed = (b_r, b_s, e_r, e_s)$ und beschreibt einen gemeinsamen Teilstring von zwei Strings r und s. Es gilt: $r[b_r, e_r] = s[b_s, e_s]$. Die Länge $|seed|$ wird definiert als $b_r - e_r = b_s - e_s$.

Ein **Repeat** ist ein sich wiederholender Teilabschnitt innerhalb einer Sequenz. Formal wird er hier folgend definiert: *Seed* $a = (b_{1r}, b_{1s}, e_{1r}, e_{1s})$ und *seed* $b = (b_{2r}, b_{2s}, e_{2r}, e_{2s})$, welche beim untersuchen eines q-grams von s gegen die Referenz r gefunden werden, beschreiben dann einen Repeat, wenn $|b_{1r} - b_{2r}| < q$.

Ein **Δ-Event** $e = (pos, type, seqs, ins, del)$ repräsentiert eine Veränderung einer Referenzsequenz r and der Position pos. Es kann einen von vier Typen annehmen ($type$): SNP, Deletion(DEL), Insertion(INS) und strukturelle Variation(SV). Mit Hilfe von $seqs$ wird gespeichert, in welchen Sequenzen e vorkommt. ins bezeichnet einen String der an Stelle pos eingefügt wird und del die Länge des Teilstrings $r[pos, pos + del]$ der gelöscht wird.

Eine **dependent region** ist eine Gruppe von Δ-Events, welche jeweils direkt oder indirekt voneinander abhängig sind. Definition von Abhängigkeit: $e_1 = (pos_1, type_1, seq_1, ins_1, del_1)$, $e_2 = (pos_2, type_2, seq_2, ins_2, del_2)$, o.B.d.A gilt $pos_1 \leq pos_2$; dann ist e_1 ist direkt abhängig von e_2 ($e_1 \sim e_2$) wenn: $(pos_1 + del) > pos_2$. e_1 ist indirekt abhängig von e_2, wenn ein Event (oder eine Kette mehrerer Events) e_i existieren, sodass gilt $(e_1 \sim e_i \wedge e_i \sim e_2)$.

3.2 Konzept

Der vorgestellte Algorithmus kann grundlegend in in zwei Abschnitte unterteilt werden: Dem "Variant Calling" und der Verbesserung der Kompression durch Evenprozessierung. Die Eingabe erfordert Sequenzen im FASTA-Format, welche komprimiert in die Enddatenstruktur, die DeltaMap, überführt werden. Zur Visualisierung ist ein Beispiel in Grafik 3.2 dargelegt.

Beim *"Variant Calling"* (Sektion 3.3) werden Sequenzen gegen eine Referenz aligniert, um die daraus resultierenden Unterschiede zur Referenz als Δ-Events abzuspeichern. Diese Methode entspricht vom Prinzip her dem *Variant Calling* zur Erstellung eines VCF-Formats (GATK [20]). Der Unterschied ist, dass kein perfektes globales Alignment erstellt wird und die Δ-Events somit nicht mit den VCF-Einträgen vergleichbar wären. Zunächst werden hierzu die Sequenzen eingelesen und die erste dient im Folgenden als Referenz. Durch eine Index-basierte Suche werden *seeds* zwischen den einzelnen Sequenzen und der Referenz gefunden und erweitert. Die Menge an gefundenen *seeds* wird anschließend zu einer globalen Kette reduziert. Die globale Kette kann als Repräsentation eines Alignments gesehen werden. Die Bereiche zwischen den gewählten *seeds* der globalen Kette stellen die Unterschiede, oder auch Varianten, dar und werden als Δ-Events gespeichert.

Die *Eventproessierung* (Sektion 3.4) soll die Anzahl der generierten Δ-Events, sowie den Speicheraufwand der Enddatenstruktur (DeltMap) verringern. Hierzu werden als Erstes gleiche Events auf verschiedenen Sequenzen als ein Event zusammengeführt. Die Kodierund der Sequenzen in Events entspricht nun eher einem Multilen-Alignment. Nachfolend werden voneinander abhängige Events zur weiteren Evaluierung in Gruppen eingeteilt (*dependent regions*). In Jeder Gruppe wird den jeweiligen Events dann ein Score zugewiesen,

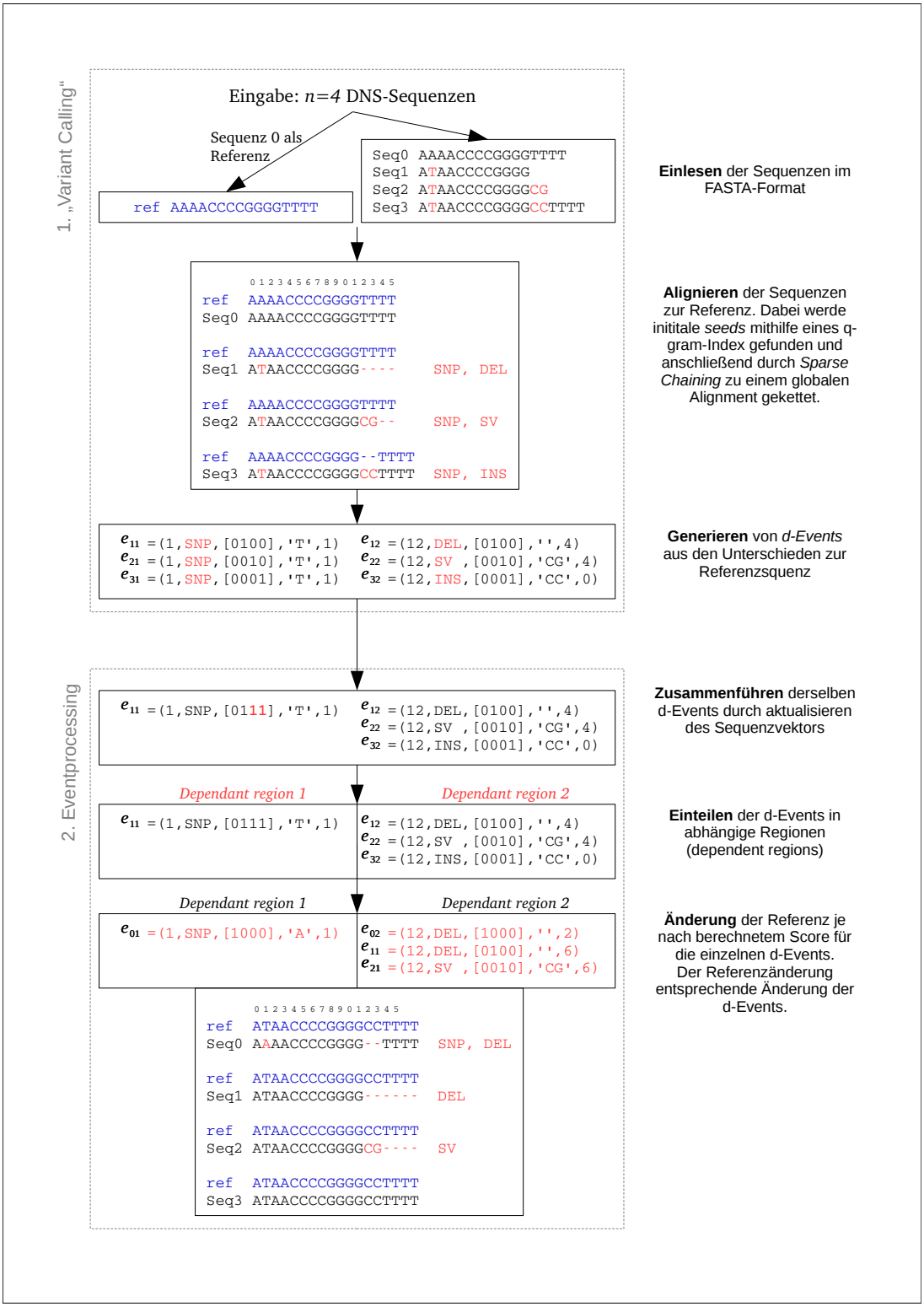

Figure 4: Konzept des Kompressionsalgorithmus' anhand eines Beispiels.

welcher die mögliche Speicherverringerung beschreibt. Je nach Score werden einige Events in die Referenzsequenz eingebaut und die Sequenzkodierung wird entsprechend aktualisiert.

3.3 Implementierung - "Variant Calling"

3.3.1 Indexsuche

Die Suche nach Gemeinsamkeiten zwischen einer Sequenz und ihrer Referenz ist eines der Schlüsselprobleme bei der referentiellen Kompression. In dem hier vorgestellten Algorithmus wird dazu zunächst ein q-Gram-Index über der Referenz aufgebaut. Die Wahl des Parameters q wird dem Benutzer überlassen. Allerdings wird empfohlen, das q bei genomischer DNS über 15 liegen sollte, damit zufällige q-mere vermieden werden [22]. Da bei großen Werten für q, auf einem Alphabet $\sum$, die Anzahl an möglichen q-meren exponentiell steigt ($|\sum|^q$), wird ab einem Wert von

$$q > log_{|\sum|}(RAM\ Speicher\ in\ Byte) \tag{1}$$

das *OpenAdressing()* (SeqAn [9]) verwendet. Beim *OprenAdressing()* werden nur solche q-mere gespeichert, die auch tatsächlich in der indizierten Sequenz zu finden sind.

Wird jede der m Eingabesequenzen, mit einer jeweiligen Durchschnittslänge n, sequentiell nach gemeinsamen q-meren durchsucht, ergibt sich trotz Zugriff auf den q-Gram-Index in $O(1)$ eine Laufzeit von $O(n * m)$. Aus diesem Grund wird die Suche um einige Faktoren erweitert, die auf die spezifischen Eigenschaften von DNS eingehen.

3.3.2 Verbesserung der Suche

Die Indexsuche wird um folgende vier Punkte erweitert:

(1) *Ignorieren von Repeats.* Genomische DNS enthält meist sehr viele repititive Abschnitte. Umfasst ein q-gram einen Repeat würde er die Suche enorm verlangsamen, denn pro q-gram in der durchsuchten Sequenz würden viele tausende in der Referenz gefunden werden. Solche q-gramme werden als Treffer ignoriert, indem jeder gefundene seed mittels seines Vorgängers auf einen Repeat getestet wird (siehe formale Definition in Sektion 3.1). Ähnlich verhält es sich auch, falls das DNS-Alphabet um den Buchstaben 'N' erweitert ist und ein q-gram ausschließlich aus 'N' besteht. Sie werden ebenfalls ignoriert.

(2) *Iterativer Schritt*(Grafik 3.3.2). Die Schnelligkeit kann auf kostet der Sensitivität erhöht werden, wenn der Benutzer einen relativ hohen Wert für q wählt. Um dennoch eine ausreichende Genauigkeit zu gewährleisten, können im Programmaufruf ein oder mehrere

iterative Schritte mit niedrigeren Werten für q spezifiziert werden. Der Algorithmus durchläuft dann eine initiale Suche mit dem ersten q-Wert, speichert die resultierende globale Kette und starten dann in den Bereichen zwischen den ausgewählten *seeds* der Kette einen erneute Suche mit dem nächsten Wert für q. Diese Methode ist der des LAGAN Algorithmus für globale Alignments nachempfunden [1].

(3) *Initiale Suche mit dynamischen Offset.* Diese Erweiterung basiert auf der Erwartung, dass bei sehr ähnlichen Sequenzen lange gleiche Abschnitte zwischen einer Sequenz und ihrer Referenz existieren. Anstatt eine sequentielle Suche (Offset=1 oder q) durchzuführen, wird in jedem Schritt der Suche ein neuer Offset bestimmt. Hierfür wird an einer Position p in der Sequenz jeder gefundene $seed_i(p)$ (jedes initiale q-gram) zunächst erweitert. Die Erweiterung, bzw. Verlängerung des *seeds*, erfolgt solange in beide Richtungen, bis ein *mismatch* (Unterschied) auftritt. Anschließend wird jedem Seed ein Score

$$Score(seed) = |seed_i(p)| + diagonal(seed_i(p)) \tag{2}$$

zugewiesen, wobei $diagonal(seed_i(p))$ die Abweichung zur Hauptdiagonalen angibt. Die Länge des *seeds* mit dem besten Score wird als neuer Offset gesetzt und die Suche auf der Sequenz wird an Position $p + offset$ fortgeführt. Etwaige Verluste an Sensitivität können durch iterative Schritte wieder ausgeglichen werden.

(4) *Parallelisierung der initialen Suche.* Um die Suche abermals zu beschleunigen wird die initiale Suche parallelisiert. In Anlehnung an [19], wird die zu untersuchende Sequenz in Blöcke aufgeteilt. Jeder Block wird einzeln nach Gemeinsamkeiten untersucht aber im Gegensatz zu [19], werden die Blöcke vor der weiteren Analyse (und Kodierung) wieder zusammengeführt. Somit entsteht kein Informationsverlust und die Ergebnisse entsprechen denen ohne Parallelisierung.

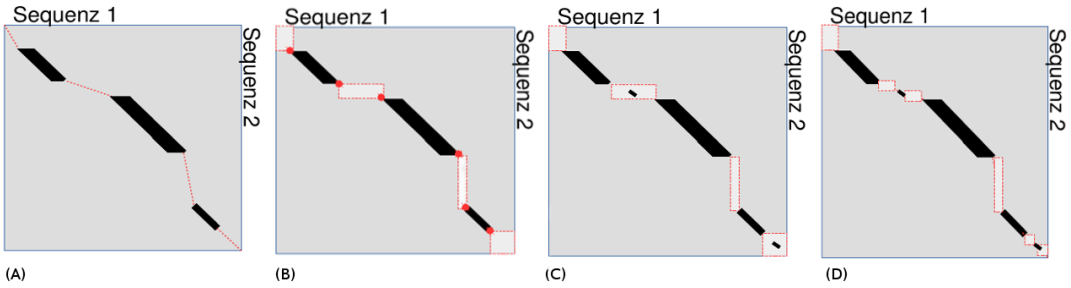

Figure 5: Iterative Schritte in der Indexsuche. In (A) sind in der initialen Suche drei große *seeds* gefunden und für die globale Kette selektiert worden. Wie in (B) dargestellt, ergeben sich dadurch Bereiche zwischen den *seeds*; Jeweils von der rechten unteren Ecke des vorhergehenden *seeds* zu linken oberen Ecke des darauffolgenden. In (C) sind dadurch kleinere *seeds* gefunden worden, die zu neuen Bereichen wie in (D) führen.

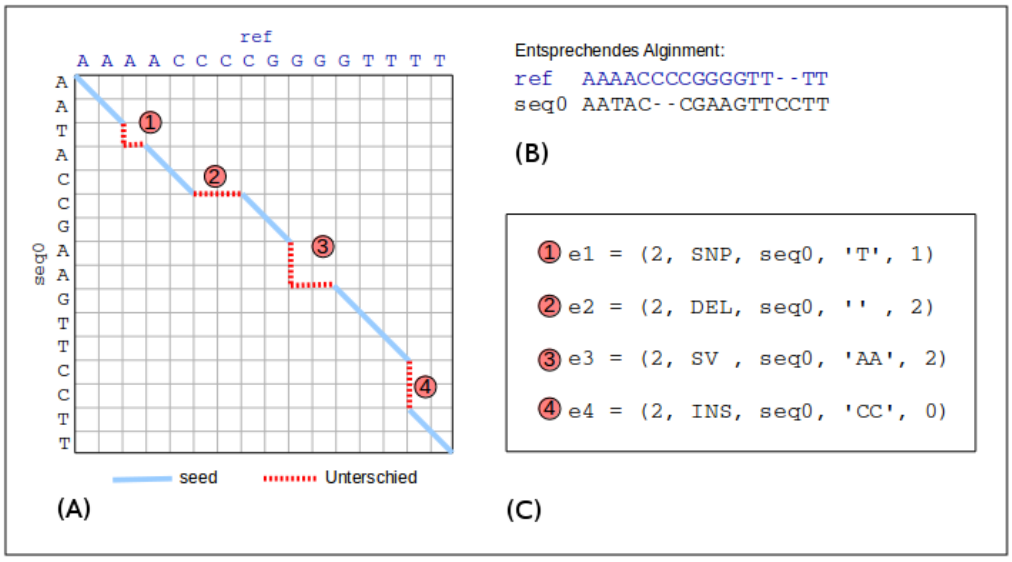

Figure 6: *Generierung von Δ-Events.* Abbildung (A) visualisiert das Ergebnis der Indexsuche. Die Bereiche zwischen den *seeds* bilden die Vorlage für die Δ-Events. (B) zeigt die entsprechende Alignment-Repräsentation und (C) die generierten Δ-Events.

3.3.3 Reduktion zu einer globalen Kette

Für die Reduktion einer Menge von *seeds* wird der *Sparse Chaining* Algorithmus von [10] verwendet. Er konstruiert in $O(r * log(r))$ eine optimale globale Kette von *seeds*, denen ein bestimmtes Gewicht (Score) zugewiesen wurde. Der Algorithmus ist bereits in der SeqAn-Bibliothek [9] enthalten und benutzt dort als Gewichtung die Länge des *seeds*. Die Gewichtung ist geeignet, da längere Gemeinsamkeiten später eine geringere Anzahl an Δ-Events bedeuten und somit zu bevorzugen sind.

Um die Laufzeit des Programmes zu verbessern, ist dem Benutzer möglich für den initialen Suchlauf (und nur für diesen) eine Mindestlänge der *seeds* zu spezifizieren. Liegt ein gefundener *seed* bei der Suche unterhalb dieser Länge wird er wieder verworfen. Mit dieser Einschränkung lässt sich die Anzahl an *seeds* kontrollieren, um den *Chaining*-Algorithmus zu entlasten. Auch hier wird ein drohender Verlust an Sensitivität, durch ausscheiden einiger mitunter wichtigen *seeds*, mit Hilfe des iterativen Schritts ausgeglichen.

3.3.4 Generierung von Δ-Events

Steht die globale Kette fest, repräsentiert sie ein globales Alignment zwischen der untersuchten Sequenz und der Referenz. Jeder *seed* beschreibt dabei eine gemeinsame Teilsequenz, wohingegen die Bereiche dazwischen Unterschiede bzw. Varianten darstellen. Jeder dieser Zwischenbereiche wird zu einem einzelnen Δ-Event, indem die zwei angrenzenden

seeds mittels Manhattan-Distanz verbunden werden. Die Position eines Δ-Events wird dabei als absolute Position in der Referenz gespeichert (Abbildung 6).

Formale Definition. *Seed* $a = (b_{1r}, b_{1s}, e_{1r}, e_{1s})$ und *seed* $b = (b_{2r}, b_{2s}, e_{2r}, e_{2s})$ liegen hintereinander in der globalen Kette, welche durch Analyse der i-ten Sequenz s gegen die Referenz r erstellt wurde. O.B.d.A liegt a über und links von b. Dann werden aus dem Bereich zwischen a und b folgende Attribute für das Event e definiert:

$$pos_e = e_{1r} \tag{3}$$

$$ins_e = s[e_{1s}, b_{2s}] \tag{4}$$

$$del_e = b_{2r} - e_{1r} \tag{5}$$

$$type_e = \begin{cases} SNP & falls\ |ins| = 1\ und\ del = 1, \\ DEL & falls\ |ins| = 0, \\ INS & falls\ del = 0, \\ SV & sonst \end{cases} \tag{6}$$

$seqs_e$ ist ein Bitvektor bestehend aus Nullen, bei dem Position i auf 1 gesetzt wird. Aus diesen Attributen wird anschließend das Δ-Event e generiert:

$$e = (pos_e, type_e, seqs_e, ins_e, del_e) \tag{7}$$

3.4 Implementierung - Verbesserung der Kompression

Die Darstellung der Eingabesequenzen als Δ-Events kann bereits als referentielle Kompression gesehen und als solche verwendet werden. Nichtsdestotrotz soll die Kompression weiterhin verbessert werden. Hierzu werden zwei Ansätze verfolgt:

Vorerst sollen gleiche Δ-Events auf unterschiedlichen Sequenzen zusammengeführt und gruppiert werden. Dies verringert den Speicherverbrauch. Zum Einen im Arbeitsspeicher für die folgende Prozessierung und zum anderen den Speicherplatz, falls die Events gespeichert werden sollten.

Ziel des zweiten Ansatzes ist die Kompression auf Ebene der *Journaled Strings*. Nach einer Gewichtung (Scoring) der zusammengefassten Events werden manche prozessiert. Mit Prozessierung ist in diesem Fall der eventuelle Einbau eines Events in die Referenz gemeint. Ziel dessen ist es, die Referenzsequenz so zu verändern, dass eine Verringerung der Anzahl an *Journal Entries* erreicht wird.

3.4.1 Zusammenführung von Events

Alle gesammelten Δ-Events (aller Sequenzen) werden zu Beginn sortiert. Eine Sortierung erfolgt nach Position, Typ und den Werten für *del* und *ins* in eben gegebener Reihenfolge. Nun wird die aufsteigend sortierte Liste durchlaufen.

Gleiche Events sollten durch die Sortierung hintereinander liegen und nun können durch Aktualisierung des *seqs*-Vektors zusammengeführt werden. Der *seqs*-Vektors ist ein Bitvektor mit einer Länge, die gleich der Anzahl an zu komprimierenden Sequenzen ist. Der i-te Eintrag in *seqs* eines Events e gibt an ob e in Sequqnz i vorkommt. Der Bitvektor gewährleistet eine effiziente Speicherung und eine einfache Handhabung mittels logischer Operationen.

3.4.2 Präprozessierung

Während der Zusammenführung werden die Events gleichzeitig in Gruppen eingeteilt. Die Gruppen beschreiben voneinander direkt oder indirekt abhängige Events (siehe Sektion 3.1 für die Definition von Abhängigkeit) innerhalb einer Region der Referenzsequenz (*dependant region*). Die Zuteilung der Events in eine Gruppe erfolgt direkt beim Durchlaufen der Liste: Gegeben sei Δ-Event e_j in einer aufsteigend sortierte List von Events. Ist e_j abhängig von e_{j-1}, wird e_j derselben Gruppe wie e_{j-1} hinzugefügt. Andernfalls ist die Gruppe von e_{j-1} vollständig und es wird eine neue Gruppe mit e_j als erstem Element initialisiert.

Die Einteilung in Gruppen ist für die nachfolgende Prozessierung von großer Bedeutung. Falls ein Event e durch seinen Einbau in die Referenz deren Sequenzabfolge verändert, sind die von e abhängigen Events davon betroffen und müssen entsprechend aktualisiert werden. Ansonsten kann keine fehlerfreie Kodierung der Sequenzen garantiert werden.

3.4.3 Prozessierung

Die Prozessierung von Events geschieht gruppenweise. Dazu wird zunächst für alle Events einer Gruppe G ein Score berechnet (siehe Sektion 3.4.4). Wird ein Event e durch seinen Score für geeignet befunden, wird es in die Referenz eingebaut.

Der Einbau in die Referenz bedeutet, das die Änderungen, die e beschreibt, in die Referenzsequenz r übernommen werden. Es wird die Teilsequenz $r[pos_e,\ pos_e + del_e]$ aus r gelöscht und die Teilsequenz ins_e an Position pos_e eingefügt. Aufgrund der Änderung der Referenz sind nun die übrigen Kodierungen der Sequenzen falsch und müssen aktualisiert werden. Bei einer Aktualisierung wird zwischen Sequenzen, welche direkt abhängige

Events zu e besitzen und denen die unabhängig sind, unterschieden.

Für die unabhängigen oder nur indirekt abhängigen Sequenzen muss lediglich ein neues Event $\tilde{e}$ erstellt werden. $\tilde{e}$ kann als gegenteiliges Event zu e gesehen werden und wird definiert als:

$$\tilde{e} = (pos_e,\ \neg type_e)\ \neg seqs_e,\ r[pos_e,\ pos_e + del_e],\ |ins_e|) \tag{8}$$

wobei

$$\neg type_e = \begin{cases} DEL & falls\ type_e = INS, \\ INS & falls\ type_e = DEL, \\ type_e & sonst \end{cases} \tag{9}$$

Für Sequenzen mit von e abhängigen Events muss jedes Event einzeln aktualisiert werden. Es wird dabei vorausgesetzt, das eine Sequenz an einer Position nur ein Event gleichzeitig haben kann. Das heißt, jede Sequenz hat nur ein von e abhängiges Event und folglich können alle abhängigen Events unabhängig voneinander bearbeitet werden. Gegeben das in die Referenz einzubauende Event e und ein von e abhängiges Event k_e, dann wird k_e wie folgt aktualisiert:
Falls $pos_e \leq pos_{k_e}$

$$k_e = (pos_{k_e},\ \ *type,\ seqs_{k_e}, ...) \tag{10}$$

Falls $pos_e \geq pos_{k_e}$

$$k_e = (pos_{k_e},\ \ *type,\ seqs_{k_e}, ...) \tag{11}$$

* wobei jeweils *type* anschließend wie Gleichung (6) ermittelt wird.

Nach der erfolgreichen Prozessierung wird der Vorgang für die aktualisierten Events der Gruppe wiederholt. Falls kein Δ-Event mehr für vorteilhaft befunden wird, wird mit der nächsten Gruppe fortgefahren.

3.4.4 Scoring - Konzept der Referenzänderung

Das Scoring (die Gewichtung) von Events kontrolliert die Prozessierung und wird für jede Gruppe unabhängig untersucht. Der Score soll ausdrücken, ob der Einbau des jeweiligen Events in die Referenz hinsichtlich einer Verbesserung der Kompression von Vorteil wäre.

Eine Verbesserung der Kompression bezieht sich in diesem Fall auf die (Arbeits-) Speicherverringung in der Enddatenstruktur. Die Enddatenstruktur ist eine Δ-Map, die in einen JST (*Journaled String Tree*) umgewandelt werden kann (???). Bei der Analyse

mit Hilfe eines JST werden innerhalb eines Kontextes Journaled Strings aufgebaut. Je weniger Journal Entries ein Journaled String hat, desto schneller die Analyse. Ziel ist es also die Anzahl an Journal Entries zu minimieren. (Absatz ist doof...)

Unter dieser Voraussetzung wird der Score eines Δ-Events wie folgt definiert: Gegeben sei Δ-Event e. Dann ist $score(e)$ definiert als die Anzahl an Journal Entries, welche durch den Einbau von e in die Referenz hinzukommen oder wegfallen würden. Es kommt folglich nur ein negativer Score für das Prozessieren in Betracht, da er eine gewünschte Verringerung der Anzahl an Journal Entries verspricht.

Ein Beispiel: Die *dependant region 1* in der Abbildung 3.2 enthält nur ein Event. Das Event e_{11} ist ein SNP der auf den Sequenzen 1,2 und 3 zu finden ist aber nicht auf Sequenz 0. Das heißt, die drei erstgenannten Squenzen hätten jeweils zwei extra Journal Entries, insgesamt sechs, als Kodierung, während Sequenz 0 keinen Eintrag benötigt. Würde man den SNP in die Referenz einbauen, würden 6 Einträge wegfallen und zwei (für Sequenz 0) hinzukommen.

$$score(e_{11}) = -(2 + 2 + 2) + 2 = -4 \tag{12}$$

Da sich kein weiteres Event in der Gruppe befindet ist dieser Score der beste und zu dem negativ: Das Event wird prozessiert.

3.4.5 Endddatenstruktur

Wie angesprochen Delta map bla bla

4 Ergebnisse

Die Auswertung wurde auf einem Server der Freien Universität Berlin durchgeführt. Es wurde ein Linux 3.13.0 System mit .... two Intel®Core™i5-3317U CPU's at 1.70GHz (a total of 2 physical and 2 virtual cores) and 5.8GB of RAM.

Das Programm wurde auf Realdaten getestet. Der erste Datensatz besteht aus bis zu 200 Versionen des menschlichen Chromosoms 21 erhalten vom *1000 Genome Project* [3]. Alle Varianten der 1000 Individuen sind einer VCF-Datei gespeichert und mussten für die Analyse zurück in einzelne FASTA-Dateien überführt werden. Hierfür wurde vcf-subset aus dem Paket vcf-tools benutzt, um den VCF-Datensatz auf einzelne Individuen zu reduzieren. Anschließend wurde mittels GATK -Genome..bla.bla aus dem VCF-Format und der im Projekt angegebenen Referenz die entsprechende FASTA-Datei generiert. Der zweite Datensatz besteht aus 20 E.Coli Genomen aus/von [Referenz].

Der Kompressionsalgorithmus besitzt folgende Parametereinstellungen:

Kürzel	Parameter	Beschreibung
-h	*--help*	Displays this help message.
-v	*--version*	Display version information.
-i	*--input_file*	Path to the input file. Valid filetype: FASTA.
-q	*--$q_g$ram*	Size of the q-gram(-index) when finding seeds. Define multiple times for iterative search
-t	*--threads*	The number of threads that can be used.
-mss	*--maximum_seed_size*	The threshold in the first seeding step for the maximum seed size.
-ev	*--evaluation*	If set, this will trigger an evaluation of the performed algorithm.Evaluation: number of Journal Entries before and after compression, verification whether sequences can be restored correctly,estimated bytes of storage needed in memory space.

Table 1: *Parameter des SeqAn Tools für referentielle Kompression.*

4.1 Evaluation der Indexsuche

Um zunächst die Qualität der Indexsuche zu evaluieren, wird die Seed-Anzahl mit der bekannten Anzahl an VCF-Einträgen verglichen. Das durch die Indexsuche produzierte Alignment ist, wie bereits erwähnt, kein optimales globales Alignment aber es sollte sich

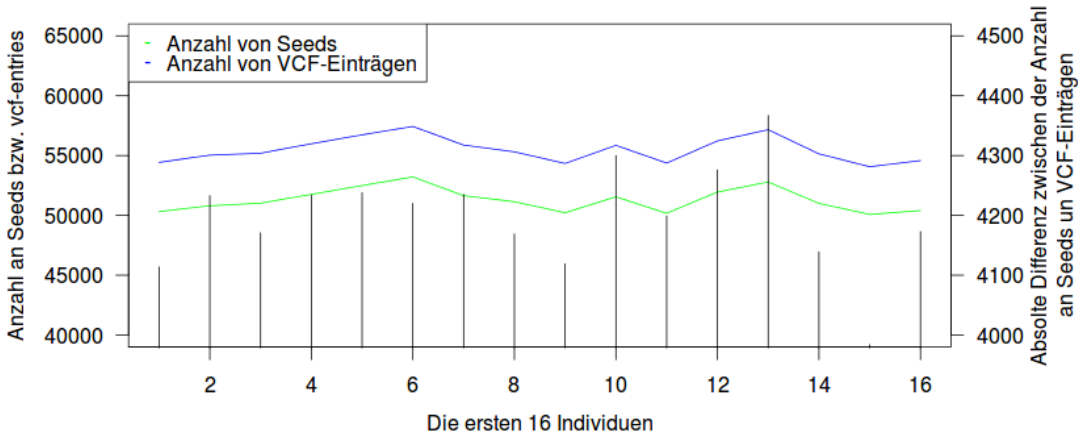

Figure 7: *Vergleich der Anzahl von Seeds gegen die zugrunde liegenden VCF-Einträge.* Der blaue Graph zeigt die Anzahl an Varianten der Referenzdatei des 1000 Genome Project. Der grüne Graph zeigt die entsprechende Anzahl an gefundenen Seeds als Ergebnis der Indexsuche. Die Indexsuche wurde mit dem Parameter $q = 20$ und im Iterativen Schritt mit $q = 15$ auf 16 Kernen durchgeführt, mit einer initialen Mindestgröße für Seeds von 100 . Die hinzugefügten Balken zeigen die absolute Differenz zwischen beiden Graphen.

diesem idealerweise annähern.

Abbildung 7 zeigt die Untersuchung am Chromosomen 21 der ersten sechzehn Individuen des 1000 Genome Projekt. Es ist eine starke Korrelation der beiden Graphen erkennbar. Die Absoluten Differenzen zwischen den Anzahlen variieren leicht zwischen den Individuen. Für die spätere Auswertung wichtig zu erwähnen ist, dass die Ergebnisse in Abbildung 7 mit dem Parametern $q = 20$ und im Iterativen Schritt $q = 15$ durchgeführt wurden. Das bedeutet Das Varianten die weniger als 15 Basen von einander entfernt liegen nicht separat erkannt werden und folglich als eine einzelne große Variante gespeichert werden. Im Durchschnitt der sechzehn untersuchten Individuen, haben 3792 Varianten einen geringeren Abstand als 15 Basen zu Ihrem Vorgänger.

4.2 Leistung beim menschlichen Chromosoms 21

Der Kompressionsalgorithmus wurde auf einer verschiedenen Anzahl an menschlichen Chromosomen 21 getestet. Für alle 4 Datensätze (Anzahl 10,50,100 und 200 Sequenzen) wurden folgende Parametereinstellungen gewählt:

```
1  $ compressFASTA -i input.fa -q 20 -q 15 -mss 100 -t 16 (-ev)
```

Auf den Parameter -*ev* wurde zur Laufzeit- und Arbeitsspeicheranalyse verzichtet.

#Chr 21	Rohdaten	Kompression auf Event-Ebene			Kompression auf JS-Ebene		
		Vorher	Nachher	**Ratio**	Vorher	Nachher	**Ratio**
10	573.72	7.24	2.29	**250.16**	12.93	7.28	**78.82**
50	2659.97	36.08	7.02	**378.94**	64.19	34.83	**76.38**
100	5215.63	71.68	12.77	**408.56**	127.49	69.03	**75.55**
200	10431.27	144.08	24.69	**422.50**	256.47	139.53	**74.76**

Table 2: *Übersicht des Ergebnisse für das menschliche Chromosom 21.* Alle Angaben, bis auf die erste Zeile mit der Anzahl an Sequenzen, sind in Megabyte. Die Bezeichnung *Vorher* bezieht sich hierbei auf das Ergebnis vor den in Sektion 3.4 beschriebenen Verbesserungen: Auf Event-Ebene vor der Zusammenführung und auf Journaled String (JS)-Ebene vor der Prozessierung. Das Verhältnis (Ratio) bezieht sich jeweils auf die *'Nachher'*-Werte zu den entsprechenden Rohdaten.

Die resultierenden Ergebnisse sind in Tabelle 2 aufgelistet. Es wird zwischen dem Speicheraufwand der eigentliche Speicherung einer Datei (Kompression auf Event-Ebene) und dem Arbeitsspeicheraufwand bei der Weiterverwendung mittels Journaled Strings (Kompression auf JS-Ebene) unterschieden.

Die generierten (vorher) und zusammengeführten (nachher) Δ-Events können als solche in einer Datei gespeichert werden. Bisher wurde noch kein entsprechendes Ausgabeformat erstellt, weswegen die angegeben Werte hypothetischen Berechnungen entsprechen. Dabei werden für die Attribute *pos*, *type* und *del* jeweils ein *unsigned integer* (4 Byte) und für den Bitvektor *seqs* eine üngefähre Größe von #Sequenzen/8 Byte (8 Bit pro Byte) berechnet. Der Wert für *ins* wird bei der Ausgabe des Programms für jede Sequenz angegeben (vorrausgesetzt *-ev* wurde spezifiziert). Es fällt auf, dass je mehr Sequenzen komprimiert werden, desto bessere Kompressionsverhältnisse werden erreicht. Die Verhältnisse sind mit denen anderer referentieller Algorithmen nur bedingt vergleichbar, da andere Datensätze verwendet wurden. [referenzen mit einem beispiel.].

Die Kompression auf Ebene der Journaled Strings erzielte geringere Verhältnisse. Es ist wichtig anzumerken, dass die Zusammenführung von Events keine Auswirkungen auf die Anzahl an Journal Entries hat, da jede Sequenz individuell und unabhängig aufgebaut werden muss. Daher entsprechen die Werte auch eher den 'Vorher'-Werten der Kompression auf Event-Ebene. Ein Δ-Event ist prinzipiell mit einem Journal Entry gleichzusetzen. Zur Berechnung des Speicheraufwands eines Journaled Strings ..... . Bei zunehmender Anzahl an Sequenzen singt das Kompressionsverhältnis in geringem Maße. Außerdem kann festgehalten werden, dass die Anzahl an Journal Entries mit Hilfe der Prozessierung um durchschnittlich die Hälfte reduziert wird, wodurch sich auch die der Speicherplatzverbrauch halbiert.

Laufzeit .... ?!

4.3 Leistung bei E.Coli Genomen

5 Disskusion und Fazit

References

[1] Brudno, M., Do, C. B., Cooper, G. M., Kim, M. F., Davydov, E., Program, N. I. S. C. C. S., Green, E. D., Sidow, A., and Batzoglou, S. LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA. *Genome Res 13*, 4 (2003), 721–731.

[2] Christley, S., Lu, Y., Li, C., and Xie, X. Human genomes as email attachments. *Bioinformatics 25*, 2 (Jan. 2009), 274–275.

[3] Consortium, T. . G. P. An integrated map of genetic variation from 1,092 human genomes. *Nature 491*, 7422 (Oct. 2012), 56–65.

[4] Deorowicz, S., Danek, A., and Grabowski, S. Genome compression: a novel approach for large collections. *Bioinformatics 29*, 20 (oct 2013), 2572–2578.

[5] Deorowicz, S., Danek, A., and Niemiec, M. Gdc 2: Compression of large collections of genomes. *Cornell University Library arXiv:1503.01624* (2015).

[6] Deorowicz, S., and Grabowski, S. Compression of genomic sequences in FASTQ format. *Bioinformatics 27*, 6 (Jan. 2011), 860–862.

[7] Deorowicz, S., and Grabowski, S. Robust relative compression of genomes with random access. *Bioinformatics 27*, 21 (Nov. 2011), 2979–2986.

[8] Deorowicz, S., and Grabowski, S. Data compression for sequencing data. *Algorithms for Molecular Biology 8*, 1 (Nov. 2013), 25+.

[9] Döring, A., Weese, D., Rausch, T., and Reinert, K. SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics 9* (2008), 11.

[10] Gusfield, D. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, Janurary 1997.

[11] Huffman, D. A method for the construction of minimum-redundancy codes. *Resonance 11*, 2 (Feb. 2006), 91–99.

[12] Jean. gzip. Programa de computador, 1992.

[13] Kuruppu, S., Puglisi, S., and Zobel, J. Reference sequence construction for relative compression of genomes. In *String Processing and Information Retrieval*, R. Grossi, F. Sebastiani, and F. Silvestri, Eds., vol. 7024 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 420–425.

[14] Kuruppu, S., Puglisi, S. J., and Zobel, J. Optimized relative lempel-ziv compression of genomes. In *Proceedings of the Thirty-Fourth Australasian Computer Science Conference - Volume 113* (Darlinghurst, Australia, Australia, 2011), ACSC '11, Australian Computer Society, Inc., pp. 91–98.

[15] Loh, P.-R., Baym, M., and Berger, B. Compressive genomics. *Nature Biotechnology 30*, 7 (July 2012), 627–630.

[16] NHGRI. Dna sequencing costs: Data from the nhgri genome sequencing program (gsp). Tech. rep., National Human Genome Research Institute, 2015.

[17] Pinho, A. J., Pratas, D., and Garcia, S. P. GReEn: a tool for efficient compression of genome resequencing data. *Nucleic Acids Research 40*, 4 (Feb. 2012), e27.

[18] Rahn, R., Weese, D., and Reinert, K. Journaled string tree - a scalable data structure for analyzing thousands of similar genomes on your laptop. *Bioinformatics* (2014).

[19] Rani, M. M. S. a new referential method for compressing genomes. *International Journal of Computational Bioinformatics* (2015).

[20] Van der Auwera, G. A., Carneiro, M. O., Hartl, C., Poplin, R., del Angel, G., Levy-Moonshine, A., Jordan, T., Shakir, K., Roazen, D., Thibault, J., Banks, E., Garimella, K. V., Altshuler, D., Gabriel, S., and DePristo, M. A. From FastQ Data to High-Confidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline. *Current protocols in bioinformatics / editoral board, Andreas D. Baxevanis ... [et al.] 11*, 1110 (Oct. 2002).

[21] Venter, J., et al. The sequence of the human genome. *Science 291*, 5507 (2001), 1304–1351.

[22] Wandelt, S., Bux, M., and Leser, U. Trends in genome compression. *Current Bioinformatics 9*, 3 (2014), 315–326.

[23] Wandelt, S., and Leser, U. Adaptive efficient compression of genomes. *Algorithms for Molecular Biology 7*, 1 (2012), 30+.

[24] Wang, C., and Zhang, D. A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Research 39*, 7 (Apr. 2011), e45.

[25] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY 23*, 3 (1977), 337–343.