Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften
Department Informatik

Bachelorarbeit

Speichereffiziente Methoden zur Repräsentation von paarweisen Sequenz-Alignments

Thorben Wiese

3wiese@informatik.uni-hamburg.de
Studiengang B.Sc. Informatik
Matr.-Nr. 6537204
Fachsemester 6

Erstgutachter Universität Hamburg:	Prof. Dr. Stefan Kurtz
Zweitgutachter Universität Hamburg:	Dr. Giorgio Gonnella

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Ein Sequenzalignment wird in der Bioinformatik dazu verwendet, zwei oder mehrere Sequenzen von zum Beispiel DNA-Strängen oder Proteinsequenzen miteinander zu vergleichen und die Verwandtschaft zu bestimmen. Ein Alignment ist das Ergebnis eines solchen Vergleichs. Bei einem globalen Alignment wird jeweils die gesamte Sequenz betrachtet, bei einem lokalen Alignment lediglich Teilabschnitte der beiden Sequenzen. Um die verschiedenen Sequenzen vergleichen zu können, berechnet man einen Score oder die Kosten, um den Aufwand, den man betreiben muss, um die gegebenene Sequenz in die Zielsequenz umzuwandeln, beschreiben zu können. Hierbei wird jeweils das Optimum, also entweder der maximale Score oder die minimalen Kosten gesucht. Die verschiedenen Schritte, um die Symbole der Strings zu verändern, sind bei Gleichheit ein 'match', bei der Substitution ein 'mismatch', bei der Löschung eine 'deletion' und bei der Einfügung eine 'insertion', welche je nach Verfahren unterschiedlich gewichtet werden können. Hierbei haben ähnliche Sequenzen einen hohen Score und geringe Kosten und unterschiedliche Sequenzen analog einen kleinen Score und hohe Kosten.

Ziel dieser Bachelorarbeit ist es, eine speichereffiziente Repräsentation von paarweisen Sequenzalignments zu implementieren und die Funktionsweise, sowie Vergleiche zu anderen Verfahren zu diskutieren.

Die Edit-Operationen

Die in diesem Kapitel eingeführten Begriffe werden in [Kurtz, S. 5-7, 14-16] definiert.

Sei $\mathcal{A}$ eine endliche Menge von Buchstaben, die man Alphabet nennt. Für DNA-Sequenzen verwendet man üblicherweise die Menge der Basen, also $\mathcal{A} = \{a, c, g, t\}$. $\mathcal{A}^i$ sei die Menge der Sequenzen der Länge i aus $\mathcal{A}$ und ε sei die leere Sequenz. Formal ausgedrückt ist eine Edit-Operation ein Tupel

$$(\alpha, \beta) \in (\mathcal{A}^1 \cup \{\varepsilon\}) \times (\mathcal{A}^1 \cup \{\varepsilon\}) \backslash \{(\varepsilon, \varepsilon)\}.$$

Eine äquivalente Schreibweise von (α, β) ist $\alpha \rightarrow \beta$. Es gibt drei verschiedene Edit-Operationen

$a \rightarrow \varepsilon$ ist eine Deletion für alle $a \in \mathcal{A}$

$\varepsilon \rightarrow b$ ist eine Insertion für alle $b \in \mathcal{A}$

$a \rightarrow b$ ist eine Substitution für alle $a, b \in \mathcal{A}$

Dabei ist zu beachten, dass $\varepsilon \rightarrow \varepsilon$ keine Edit-Operation darstellt.

Ein Alignment von zwei Sequenzen u und v lässt sich nun als eine Sequenz $(\alpha_1 \rightarrow \beta_1, ..., \alpha_h \rightarrow \beta_h)$ von Edit-Operationen definieren, sodass $u = \alpha_1 ... \alpha_h$ und $v = \beta_1 ... \beta_h$ gilt.

Die Edit-Distanz

Sei eine Kostenfunktion δ mit $\delta(a \rightarrow b) \geq 0$ für alle Substitutionen $a \rightarrow b$ und $\delta(\alpha \rightarrow \beta) > 0$ für alle Einfügungen und Löschungen $\alpha \rightarrow \beta$ gegeben. Die Kosten für ein Alignment $A = (\alpha_1 \rightarrow \beta_1, ..., \alpha_h \rightarrow \beta_h)$ ist die Summe der Kosten aller Edit-Operationen des Alignments.

$$\delta(A) = \sum_{i=1}^{h} \delta(\alpha_i \rightarrow \beta_i)$$

Ein Beispiel einer Kostenfunktion ist die Einheitskostenfunktion

$$\delta(\alpha \rightarrow \beta) = \begin{cases} 0, & \text{wenn } \alpha, \beta \in \mathcal{A} \text{ und } \alpha = \beta \\ 1, & \text{sonst.} \end{cases}$$

Die Edit-Distanz von zwei Sequenzen ist wie folgt definiert:

$$edist_\delta(u, v) = \min\{\delta(A) \mid A \text{ ist Alignment von } u \text{ und } v\}$$

Ein Alignment A ist optimal, wenn $\delta(A) = edist_\delta(u, v)$ gilt.

Wenn δ die Einheitskostenfunktion ist, so ist $edist_\delta(u, v)$ die Levenshtein Distanz [Kurtz, S. 19-21].

Ein Alignment kann für eine Edit-Distanz e mit der Einheitskostenfunktion in

$O(e)$ Zeit berechnet werden [Kurtz, S. 41-42].

2 Methoden

2.1 CIGAR-Strings

Ein Dateiformat, welches zur Speicherung von Alignments verwendet wird, ist das SAM-Format oder die binär komprimierte Version BAM. Dieses codiert ein Alignment in einem sogenannten CIGAR-String, der aus einzelnen Zeichen besteht, die jeweils eine Edit-Operation bezeichnen, also M für eine Substitution, I für eine Insertion und D für eine Deletion. Gleiche aufeinanderfolgende Operationen werden als Kombination von Quantität und Symbol geschrieben.

Beispiel 1. Sei $u =$ `actgaact`, $v =$ `actagaat` und das Alignment $A = (a \to a, c \to c, t \to t, ...)$ gegeben.

```
a c t - g a a c t
| | |   | | |   |
a c t a g a a - t
```

Ein Alignment wird üblicherweise in drei Zeilen geschrieben, wobei in der ersten Zeile die Sequenz u und in der dritten Zeile die Sequenz v geschrieben wird. In der mittleren Zeile symbolisiert das Zeichen '|' eine Substitution, wobei üblicherweise nur ein Match markiert wird. Außerdem wird ein ε aus der Edit-Operation in diesem Fall durch das Zeichen '-' dargestellt.

Dieses Alignment wird duch den CIGAR-String `3M1I3M1D1M` repräsentiert [The SAM/BAM Format Specification Group 2015].

2.1.1 Speicherverbrauch

2.1.2 Kodierung eines CIGAR-Strings

Beispiel 2. Sei das Alignment A

```
0    5    0    5    0    5    0    5    0
gagc-a-t-gttgcc-tggtcctttgctaggtactgta-gaga
||  |  |  |  |   | |  |||  ||||  |||  |||  |||||  ||||
gaccaagtag--g-cgtggacctt-gctcggt-ctgtaagaga
0    5    0    5    0    5    0    5    0
```

gegeben, welches durch den CIGAR-String `4M1I1M1I1M1I1M2D1M1D1M1I8M 1D7M1D5M1I4M` repräsentiert werden kann.

Im Folgenden vergleiche ich für diese Art der Alignment-Repräsentation die Verfahren der naiven binären Kodierung, der unären Kodierung und der Huffman-Kodierung.

Sei das Alphabet $\mathcal{A}_1 = \{\text{M, I, D}\}$, welches alle Symbole aus dem CIGAR-String enthält, das Alphabet $\mathcal{A}_2 = \{1, 2, 4, 5, 7, 8\}$, welches alle Zahlen aus dem CIGAR-String enthält, sowie die relativen Wahrscheinlichkeiten $p(c_1)$ jeden Symbols $c_1 \in \mathcal{A}_1$ und $p(c_2)$ jeden Symbols $c_2 \in \mathcal{A}_2$ gegeben.

c_1	Häufigkeit	$p(c_1)$	c_2	Häufigkeit	$p(c_2)$
M	10	$\frac{10}{38}$	1	13	$\frac{13}{38}$
I	5	$\frac{5}{38}$	4	2	$\frac{2}{38}$
D	4	$\frac{4}{38}$	2	1	$\frac{1}{38}$
			5	1	$\frac{1}{38}$
			7	1	$\frac{1}{38}$
			8	1	$\frac{1}{38}$

Tabelle 2.1: Relative Wahrscheinlichkeiten CIGAR-String

Bei einer naiven binären Kodierung wird jedes Symbol $c_i \in \mathcal{A}_i$ mit $\lceil \log_2 n \rceil, n = |\mathcal{A}|$ Bit kodiert, also $\lceil \log_2 3 \rceil + \lceil \log_2 6 \rceil = 2+3 = 5$ Bit pro Symbol. Insgesamt ergibt das somit $19 \cdot 2 + 19 \cdot 3 = 95$ Bit.

Die unäre Kodierung kodiert jedes Symbol nach der Häufigkeit des Auftretens im Alphabet mit $i - 1$ $'1'$-Bits, gefolgt von einem $'0'$-Bit, wobei i die Position des Symbols in einer nach der Häufigkeit absteigend sortierten Liste ist. Das am Häufigsten auftretende Symbol des Alphabets wird also mit $'0'$, das zweithäufigste mit $'10'$, das dritthäufigste mit $'110'$ usw. kodiert [Moffat u. Turpin 2002, S. 29-30].

Der oben genannte CIGAR-String wird demnach wie folgt unär kodiert:

Symbol	Kodierung	Anzahl Bits	Symbol	Kodierung	Anzahl Bits
M	0	$10 \cdot 1 = 10$	1	0	$13 \cdot 1 = 13$
D	10	$5 \cdot 2 = 10$	4	10	$2 \cdot 2 = 4$
I	110	$4 \cdot 3 = 12$	2	110	$1 \cdot 3 = 3$
			5	1110	$1 \cdot 4 = 4$
			7	11110	$1 \cdot 5 = 5$
			8	111110	$1 \cdot 6 = 6$
Gesamtanzahl:		32			35

Tabelle 2.2: Unäre Kodierung des CIGAR-Strings

Insgesamt benötigt die unäre Kodierung also $32 + 35 = 67$ Bit.

Der durchschnittliche Bitverbrauch für ein Symbol beträgt

$$\frac{67}{38} \approx 1.76 \, \frac{\text{Bit}}{\text{Symbol}}$$

Bei einer *minimalen* binären Kodierung, wie sie auch im Huffman-Alogrithmus verwendet wird, werden die Längen der Codewörter anhand der relativen Wahrscheinlichkeit des Symbols im Alphabet angepasst. Somit lässt sich eine Kodierung ermöglichen, welche im Durchschnitt weniger Bit pro Symbol beansprucht [Moffat u. Turpin 2002, S. 53-57]. Eine sparsamere Variante des regulären Huffman-Algorithmus ist der Kanonische Huffman-Algorithmus, welcher im Gegensatz zu der ursprünglichen Variante eine eindeutige Menge von Codewörtern liefert und keinen vollständigen Huffman-Baum, sondern lediglich die Anzahl der Codewörter für jede vorhandene Codewortlänge, sowie die sortierten Symbole benötigt, um die Informationen zu dekodieren.

Der kanonische Huffman-Algorithmus würde bei dem oben genannten Beispiel des CIGAR-Strings nach [Moffat u. Turpin 2002, S. 54] die Symbole wie in Tabelle 2.3 beschrieben kodieren. Die Huffman-Bäume beider Alphabete sind in Abbildung 2.1 dargestellt. Für die Dekodierung sind somit zusätzlich die Listen $(1, 2)$, (M,D,I) und $(0, 2, 4), (1, 4, 2, 5, 7, 8)$ zu speichern.

Der gesamte Bitverbrauch dieser Kodierung ist demnach $28 + 42 = 70$ Bit und der durchschnittliche Bitverbrauch für ein Symbol beträgt

$$\frac{70}{38} \approx 1.84 \, \frac{\text{Bit}}{\text{Symbol}}.$$

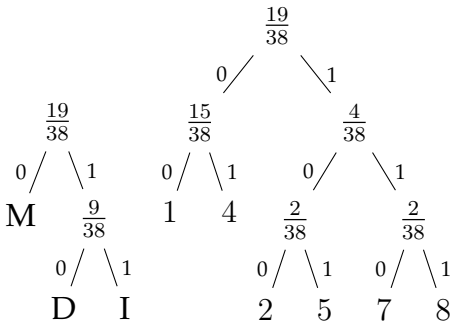

Abbildung 2.1: Huffman-Bäume der Kodierung des CIGAR-Strings

Symbol	Kodierung	Anzahl Bits	Symbol	Kodierung	Anzahl Bits
M	0	$10 \cdot 1 = 10$	1	00	$13 \cdot 2 = 26$
D	10	$5 \cdot 2 = 10$	4	01	$2 \cdot 2 = 4$
I	11	$4 \cdot 2 = 8$	2	100	$1 \cdot 3 = 3$
			5	101	$1 \cdot 3 = 3$
			7	110	$1 \cdot 3 = 3$
			8	111	$1 \cdot 3 = 3$
Gesamtanzahl:		28			42

Tabelle 2.3: Huffmann-Kodierung des CIGAR-Strings

2.2 Trace Point Konzept

Ein neuer Ansatz der speichereffizienten Repräsentation von Alignments wurde von Gene Myers in [Myers 2015] beschrieben und basiert auf dem Konzept der Trace Points.

Sei A ein Alignment von $u[i...j]$ und $v[k...\ell]$ mit $i < j$ und $k < l$ und sei $\Delta \in \mathbb{N}$. Sei $p = \lceil \frac{i}{\Delta} \rceil$. Man unterteilt $u[i...j]$ in $\tau = \lceil \frac{j}{\Delta} \rceil - \lfloor \frac{i}{\Delta} \rfloor$ Substrings $u_0, u_1, ..., u_{\tau-1}$ mit

$$u_q = \begin{cases} u[i...p \cdot \Delta] & \text{falls } q = 0 \\ u[(p+q-1) \cdot \Delta + 1...(p+q) \cdot \Delta] & \text{falls } 0 < q < \tau - 1 \\ u[(p+\tau-2) \cdot \Delta...j] & \text{falls } q = \tau - 1 \end{cases}$$

Für alle q mit $0 \leq q < \tau - 1$ sei t_q der letzte Index des Substrings von v, der in A mit u_q aligniert. t_q nennt man Trace Point. Für $q = 0$ aligniert u_0 mit $v_0 = v[k...t_0]$. Für alle q mit $0 < q < \tau - 1$ aligniert u_q mit $v_q = v[t_{q-1} + 1...t_q]$.

Seien i, j, k, ℓ, Δ und die Trace-Points eines Alignments von u und v gegeben. Dann kann ein Alignment A' von u und v mit $\delta(A') \leq \delta(A)$ konstruiert werden. Danach bestimmt man aus den Trace-Points die Substring-Paare u_q und v_q, berechnet hierfür ein optimales Alignment und konkateniert die Alignments von den aufeinanderfolgenden Substring-Paaren zu A'.

Beispiel 3.

```
Sequenz 1: gagcatgttgcctggtcctttgctaggtactgtagaga
Sequenz 2: gaccaagtaggcgtggaccttgctcggtctgtaagaga
Delta: 15

Gesamtalignment:

0    5    0    5    0    5    0    5    0
gagc-a-t-gttgcc-tggtcctttgctaggtactgta-gaga
||  |  |  |  |   |  | |||  ||||  |||  |||  |||||  ||||
gaccaagtag--g-cgtggacctt-gctcggt-ctgtaagaga
0    5    0    5    0    5    0    5    0

seq1[0...14] aligniert mit seq2[0...15]
gagc-a-t-gttgcc-tgg
||  |  |  |  |   |  | |||
gaccaagtag--g-cgtgg

seq1[15...29] aligniert mit seq2[16...28]
tcctttgctaggtac
||||  |||  |||  |
acctt-gctcggt-c

seq1[30...37] aligniert mit seq2[29...37]
tgta-gaga
||||  ||||
tgtaagaga

Trace Points: [15, 28]
```

2.2.1 Speicherverbrauch

2.2.2 Differenzen-Kodierung

Gegeben sei eine Liste $L = (a_1, a_2, ..., a_n)$ mit $a_i < a_{i+1}, 0 < i \leq n$.

Anstatt jeden Wert $a \in L$ als solchen abzuspeichern, kann alternativ die Differenz eines Wertes a_i zu dem nachfolgenden Wert a_{i+1} abgespeichert werden. Lediglich der erste (oder letzte) Wert aus L wird benötigt, um später sukzessive die ursprüngliche Liste rekonstruieren zu können.

$$L_{diff} = (a_1, (a_2 - a_1), (a_3 - a_2), ..., (a_n - a_{n-1}))$$

Bei gleichmäßig ansteigenden Werten ist die Abweichung der Differenzen zweier aufeinanderfolgender Werte in der Liste untereinander gering und die Menge der zu kodierenden Symbole verringert sich.

2.2.3 Kodierung der Trace Point Differenzen

Beispiel 4. Sei $\Delta = 5$ und das Alignment A

```
0    5    0    5    0    5    0    5    0
gagc-a-t-gttgcc-tggtcctttgctaggtactgta-gaga
||  |  |  |  |   |  |  |||  ||||  |||  |||  |||||  ||||
gaccaagtag--g-cgtggacctt-gctcggt-ctgtaagaga
0    5    0    5    0    5    0    5    0
```

wie in Abschnitt 2.1.1 mit den dazugehörigen TracePoints $5, 10, 15, 20, 24, 28$ und 34 gegeben. Es ergibt sich somit das Alphabet $\mathcal{A} = \{5, 10, 15, 20, 24, 28, 34\}$ mit ausschließlich positiven und aufsteigenden Werten.

Für die Trace Point Darstellung ist somit eine Differenzen-Kodierung möglich. Als neue Liste zu kodierender Werte ergibt sich nach 2.2.2 $L_{diff} = (5, 5, 5, 5, 4, 4, 6)$.

Um aus den Trace Points ein neues Alignment rekonstruieren zu können, benötigt man zusätzlich mindestens den Δ-Wert, damit die Grenzen der Substrings

beider Sequenzen berechnet werden können. Hierfür muss also der Δ-Wert zu L_{diff} hinzugefügt werden.

Für das oben genannten Beispiel ergibt sich somit

$$L_{diff} = (\Delta, a_1, (a_2 - a_1), (a_3 - a_2), ..., (a_n - a_{n-1})) = (5, 5, 5, 5, 5, 4, 4, 6).$$

Sei das Alphabet $\mathcal{A} = \{4, 5, 6\}$ für L_{diff}, sowie die relativen Wahrscheinlichkeiten $p(c)$ jeden Symbols $c \in \mathcal{A}$ gegeben.

c	Häufigkeit	$p(c)$
5	5	$\frac{5}{8}$
4	2	$\frac{2}{8}$
6	1	$\frac{1}{8}$

Tabelle 2.4: Relative Wahrscheinlichkeiten der Delta-Kodierung

Bei der naiven binären Kodierung ergibt sich analog zu 2.1.1 ein Bedarf von $\lceil \log_2 8 \rceil = 3$ Bit pro Symbol, also $8 \cdot 3 = 24$ Bit insgesamt.

Die unäre Kodierung ergibt für dieses Beispiel die folgende Kodierung:

Symbol	Kodierung	Anzahl Bits
5	0	$5 \cdot 1 = 5$
4	10	$2 \cdot 2 = 4$
6	110	$1 \cdot 3 = 3$
Gesamtanzahl:		12

Tabelle 2.5: Unäre Kodierung der Delta-Kodierung

Insgesamt benötigt die unäre Kodierung also 12 Bit mit einem durchschnittlichen Verbrauch pro Symbol von

$$\frac{12}{8} \approx 1.5 \frac{\text{Bit}}{\text{Symbol}}$$

Die Ausführung des Huffman-Algorithmus kodiert nach [Moffat u. Turpin 2002, S. 54] die Symbole wie in Tabelle 2.6 aufgelistet. Der dazugehörige Huffmann-Baum aus 2.2 verdeutlicht die Kodierung der einzelnen Symbole, muss aber für den kanonischen Huffman-Algorithmus, wie in 2.1.2 beschrieben, nicht komplett gespeichert werden. Aufgrund der Beschaffenheit der Codewörter des ka-

Symbol	Kodierung	Anzahl Bits
5	0	$5 \cdot 1 = 5$
4	10	$2 \cdot 2 = 4$
6	11	$1 \cdot 2 = 2$
Gesamtanzahl:		11

Tabelle 2.6: Huffman-Kodierung der Delta-Kodierung

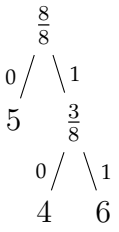

Abbildung 2.2: Huffman-Baum der Delta-Kodierung

nonischen Huffman-Algorithmus ist hier lediglich die Speicherung der Listen $(1, 2), (5, 4, 6)$ nötig.

Der gesamte Bitverbrauch der Huffman-Kodierung ist demnach 11 Bit mit einem durchschnittlichen Bedarf pro Symbol von

$$\frac{11}{8} \approx 1.38 \, \frac{\text{Bit}}{\text{Symbol}}$$

3 Resultate

3.1 CIGAR Entropie unabhängig vom Kodierungsverfahren

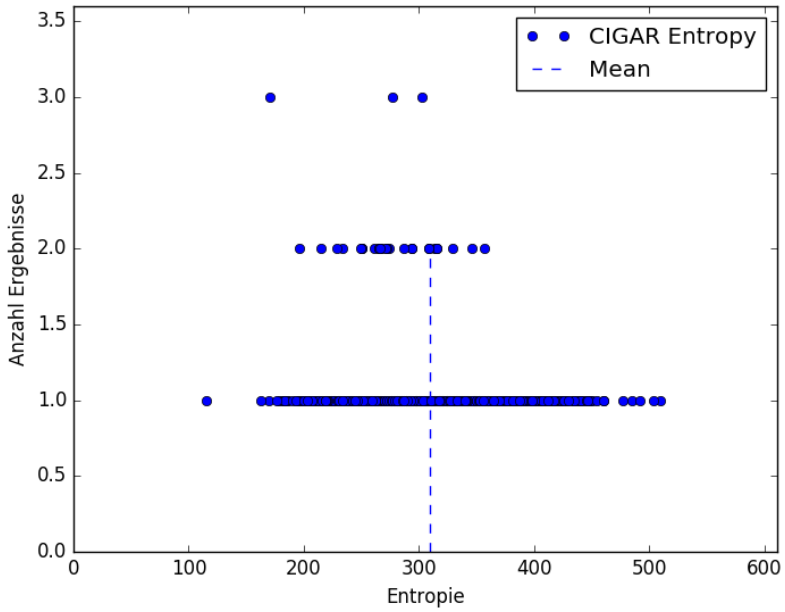

Abbildung 3.1: Entropie des CIGAR-Strings

3.2 Differenzen Entropie unabhängig vom Kodierungsverfahren

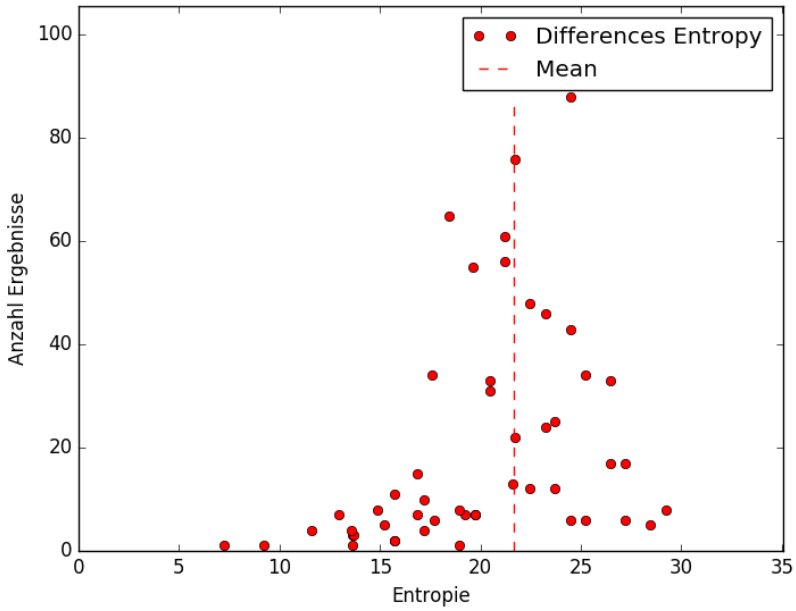

Abbildung 3.2: Entropie des CIGAR-Strings

3.3 Testläufe CIGAR Kodierung

Die folgenden Grafiken wurden mit jeweils 10.000 zufällig generierte Sequenzpaaren mit je etwa 1.000 Basen, einer Fehlerrate von 15% und einem Δ-Wert von 100 berechnet.

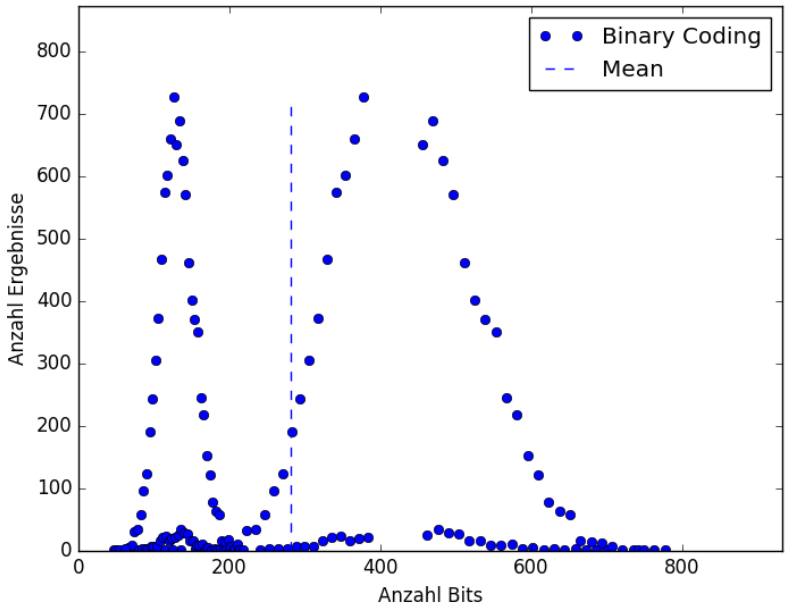

Abbildung 3.3: Bitverbrauch der binären Kodierung des CIGAR-Strings

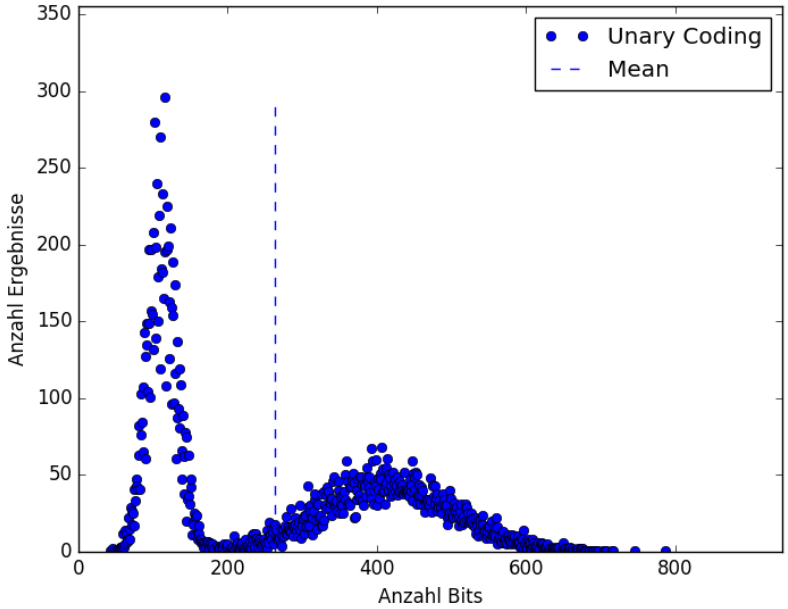

Abbildung 3.4: Bitverbrauch der unären Kodierung des CIGAR-Strings

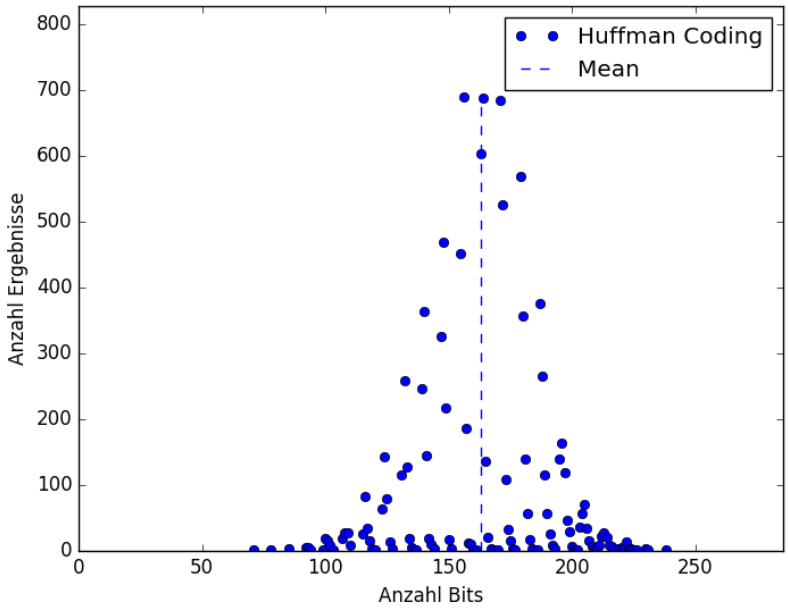

Abbildung 3.5: Bitverbrauch der Huffman-Kodierung des CIGAR-Strings

Die Abbildungen 3.3 und 3.4 verdeutlichen, dass die binäre und unäre Kodierung deutliche Schwankungen im Bitverbrauch aufweisen, welche sich durch die zufällig generierten Sequenzpaare erklären lassen, deren Alignments unter Umständen sehr viele oder ausschließlich Matches bzw. sehr viele InDels aufweisen können. Die Abbildung 3.5 hingegen weist eine Normalverteilung des Bitverbrauchs für die Huffman-Kodierung auf. Im Mittel benötigt die naive binäre Kodierung 284.60 Bit, die unäre Kodierung 264.84 Bit und die Huffman-Kodierung 162.93 Bit.

In den dargestellten Testläufen verbraucht die Huffman-Kodierung somit durchschnittlich mit Abstand am wenigsten Speicher, obwohl in dem unter 2.1.2 beschriebenen Beispiel die unäre Kodierung den CIGAR-String etwas effizienter kodiert. Der Grund hierfür ist die deutlich höhere Fehlerrate in 2.1.2, welche die Anzahl der zu kodierenden Symbole deutlich erhöht.

Das CIGAR-Format benötigt folglich wenig Speicher für Alignments mit einer kleinen Edit-Distanz und deutlich mehr Speicher für Alignments mit einer großen Edit-Distanz, da in diesem Fall eine höhere Anzahl unterschiedlicher Symbole kodiert werden muss.

3.4 Testläufe Differenzen Kodierung

Die folgenden Grafiken wurden mit jeweils 10.000 zufällig generierte Sequenzpaaren mit je etwa 1.000 Basen, einer Fehlerrate von 15% und einem Δ-Wert von 100 berechnet.

Die naive binäre Kodierung benötigt, wie in Abbildung 3.6 dargestellt, in jedem Durchlauf und somit auch im Mittel konstant 40 Bit, da für jeden Durchlauf 9 Trace Points und der Δ-Wert gespeichert werden für $\lceil \log_2 10 \rceil \cdot 10 = 40$ Bit.

Die Abbildungen 3.7 verdeutlicht, dass die unäre Kodierung Schwankungen im Bitverbrauch aufweist, welche sich wie bei den CIGAR-Strings durch die zufällig generierten Sequenzpaare erklären lassen, deren Alignments unter Umständen sehr viele oder ausschließlich Matches bzw. sehr viele InDels aufweisen können. Sie benötigt im Mittel 24.70 Bit und damit nur etwa 62% der binären Kodierung.

Für die Huffman-Kodierung wird, wie in Abbildung 3.8 zu sehen ist, deutlich

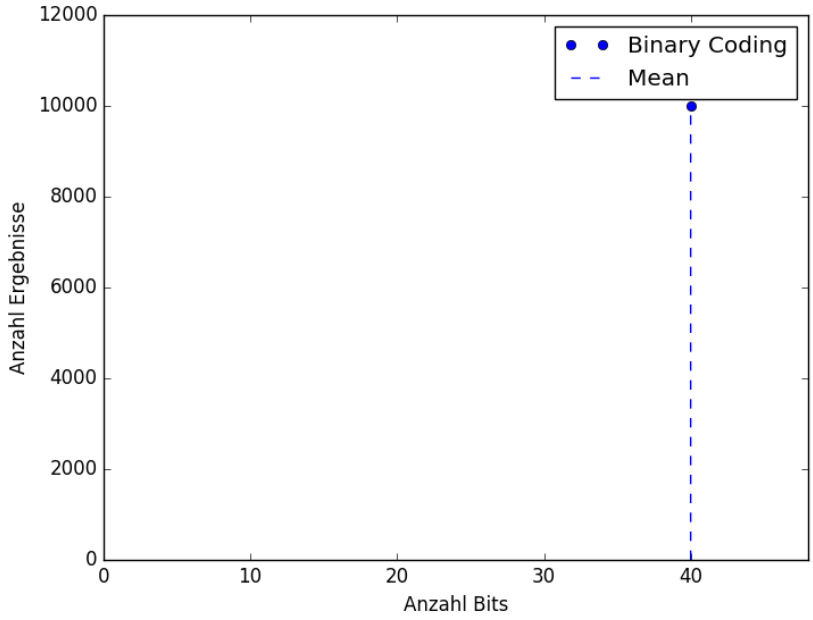

Abbildung 3.6: Bitverbrauch der naiven binären Kodierung der Delta-Kodierung

weniger Speicher für die Kodierung benötigt als für die naive binäre oder unäre Kodierung. Sie verbraucht durchschnittlich nur 14.00 Bit und damit nur etwa 57% des Speicherbedarfs der unären Kodierung.

Es ist somit zu erkennen, dass die Kodierung der Differenzen der Trace Points mit den oben genannten Parametern der Testläufe in allen Kodierungen weniger Speicher benötigt, als die Kodierung eines CIGAR-Strings, wobei die Huffman-Kodierung mit Abstand am effizientesten ist. Hierbei ist jedoch zu beachten, dass der Speicherverbrauch der Trace Point Kodierung von der Wahl des Δ-Wertes abhängt, da bei einem kleinen Δ mehr Trace Points und somit mehr Symbole gespeichert werden müssen, als bei einem großen Δ-Wert.

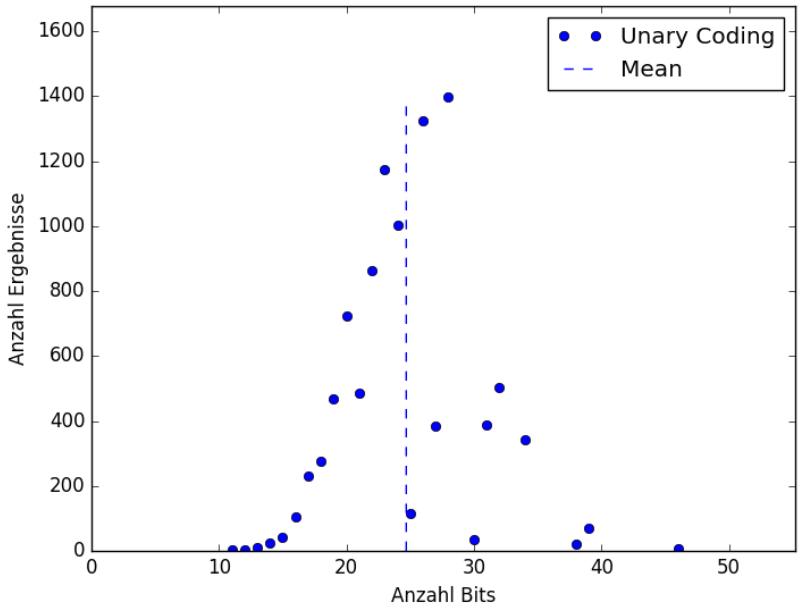

Abbildung 3.7: Bitverbrauch der unären Kodierung der Delta-Kodierung

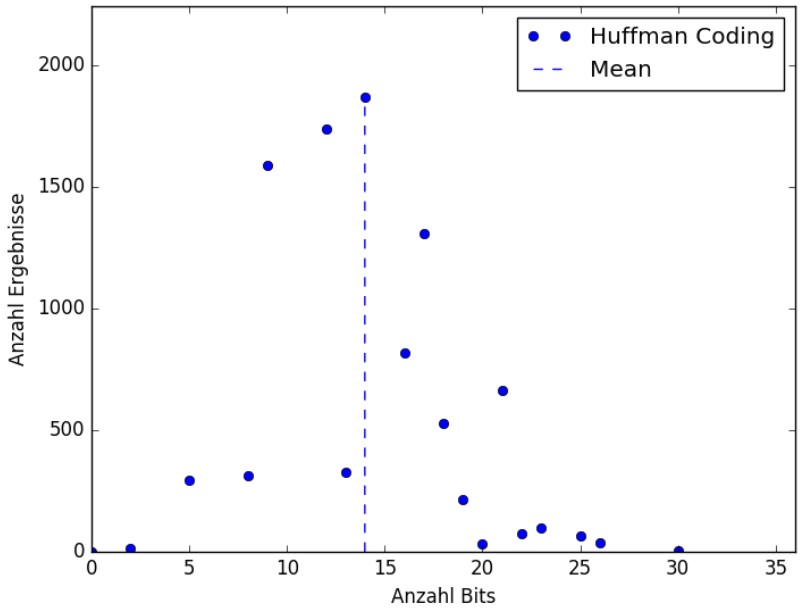

Abbildung 3.8: Bitverbrauch der Huffman-Kodierung der Delta-Kodierung

4 Diskussion

4.1 Bewertung CIGAR-Kodierung

4.2 Bewertung Differenzen-Kodierung der Trace Points

5 Programm

5.1 Aufbau

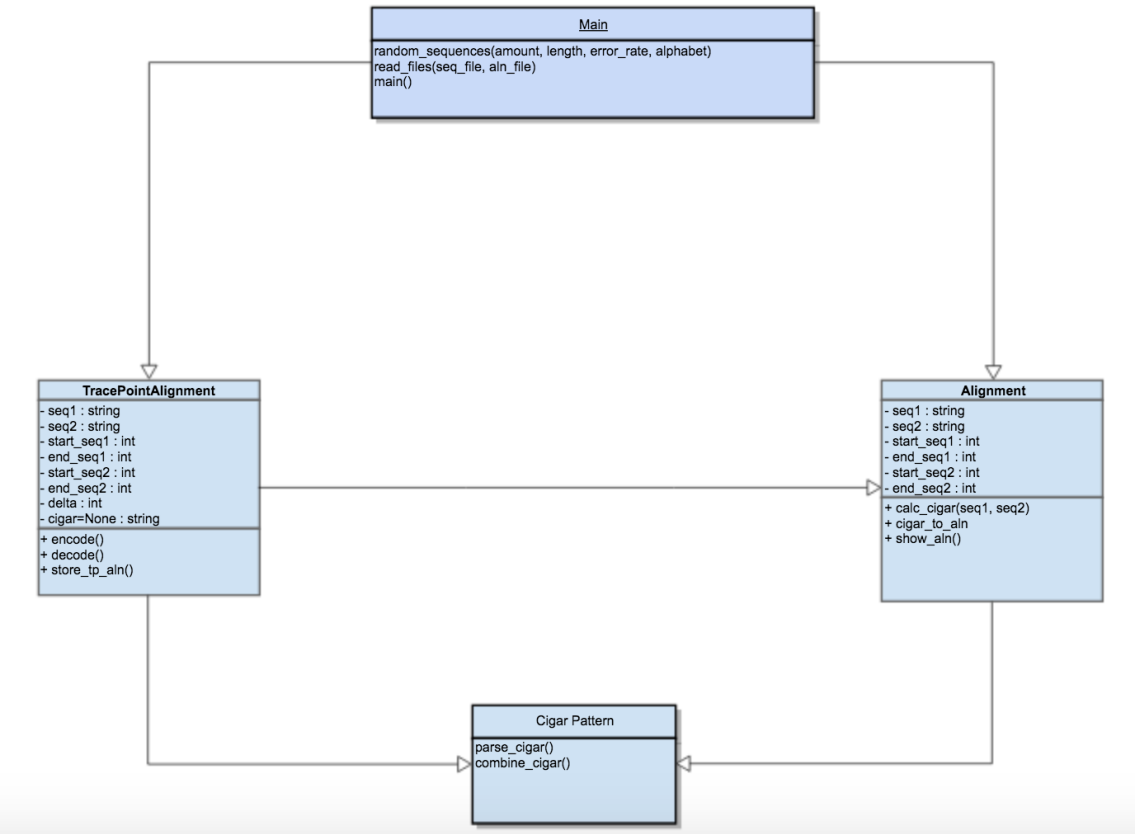

Abbildung 5.1: UML-Diagramm

5.2 Funktionalität

Algorithm 1 Computation of Trace Points from a given CIGAR-String

Input: $seq1, seq2, start\_seq1, end\_seq1, start\_seq2, \Delta, cigar$ mit
$|seq1|, |seq2|, |cigar| > 0$;
$start\_seq1, start\_seq2 \geq 0$;
$start\_seq1 < end\_seq1$ und
$\Delta > 0$

Output: Array TP of Trace Points

function *encode*($seq1, seq2, start\_seq1, end\_seq1, start\_seq2, \Delta, cigar$)
  $itv\_size \leftarrow MAX(1, \lceil start\_seq1/\Delta \rceil)$
  $itv\_count \leftarrow MIN(\lceil |seq1|/\Delta \rceil, \lceil |seq2|/\Delta \rceil)$
  for $i \leftarrow 0$ **upto** $|itv\_count|$ **do**

$$itv[i] \leftarrow \begin{cases} start\_seq1, itv\_size \cdot \Delta - 1 & \text{if } i = 0 \\ (itv\_size + i - 1) \cdot \Delta, (itv\_size + i) \cdot \Delta - 1 & \text{if } 0 < i < |itv\_count| \\ (itv\_size + i - 1) \cdot \Delta, end\_seq1 - 1 & \text{else.} \end{cases}$$

  end for
  $count1, count2, count3 \leftarrow 0$
  $TP \leftarrow$ Array for Trace Points
  for each (cig_count, cig_symbol) in cigar **do**
    for $i \leftarrow 0$ **upto** cig_count **do**
      if cig_symbol = 'I' **then**
        increment $count1$
      else if cig_symbol = 'D' **then**
        increment $count2$
      else
        increment $count1, count2$
      end if
      if count1 = intervals[count3][1] + 1 **and** count1 $\neq |seq1|$ **then**
        append (count2 - 1 + start_seq2) to TP
      end if
      if count $\neq |itv| - 1$ **then**
        increment $count3$
      end if
    end for
  end for
  return TP
end function

5.2.1 Informationsverlust bei der encode()-Funktion

Die encode-Funktion extrahiert aus dem gegebenen CIGAR-String die Trace Points, welche dann zusammen mit dem Δ-Wert und den Start- und Endpositionen der Sequenzabschnitte gespeichert werden. Hierbei geht die Information, wie die jeweiligen Intervalle zwischen den Trace Points zu den komplementären Intervallen in der Ursprungssequenz aligniert werden, verloren. Für die Rückgewinnung dieser Information muss in der decode()-Funktion zunächst ein neues Alignment des jeweiligen Intervall-Paares errechnet werden und alle Teilalignments zu einem Gesamtalignment konkateniert werden.

Algorithm 2 Computation of a CIGAR-String from a given Trace Point Array

Input: $seq1, seq2, \Delta, TP$ mit
$|seq1|, |seq2|, \Delta, |TP| > 0$

Output: CIGAR-String

1: **function** *decode*($seq1, seq2, \Delta, TP$)
2: $cig \leftarrow$ empty String
3: **for** $i \leftarrow 0$ **upto** $|TP|$ **do**
4: **if** i = 0 **then**
5: append **cigar**($seq1[0...\Delta], seq2[0...TP[i]+1]$) to cig
6: **else if** i = |TP| - 1 **then**
7: append **cigar**($seq1[i \cdot \Delta...|seq1|], seq2[TP[i-1]+1...|seq2|]$) to cig
8: **else**
9: append **cigar** ($seq1[i \cdot \Delta...(i+1) \cdot \Delta]$, $seq2[TP[i-1]+1]...TP[i]+1$) to cig
10: **end if**
11: **end for**
12: $cig \leftarrow$ **combine**(cig)
13: **return** cig
14: **end function**
15:
16: **function** *combine*($cigar$)
17: $cig \leftarrow$ empty String
18: $tmp \leftarrow 0$
19: **for** each (cig_count, cig_symbol) in cigar **do**
20: $tmp \leftarrow tmp + previous\_cig\_count$
21: **if** cig_symbol = previous_cig_symbol **then**
22: **if** not last element in cigar **then**
23: $tmp \leftarrow 0$
24: **end if**
25: **end if**
26: **if** last element in cigar **then**
27: append $(tmp + cig\_count, cig\_symbol)$ to cig
28: **end if**
29: **end for**
30: **return** cig
31: **end function**

6 Fazit

Je größer der vorher definierte positive Parameter Δ ist, desto weniger Trace Points werden gespeichert und umso länger dauert die Berechnung, um die Teil-Alignments zu rekonstruieren. Bei einem kleinen Δ werden analog mehr Trace Points gespeichert, aber die Rekonstruktionszeit der Teil-Alignments ist geringer.

Mithilfe von Δ lässt sich somit ein Trade-Off zwischen dem Speicherplatzverbrauch und dem Zeitbedarf für die Rekonstruktion der Teil-Alignments einstellen.

Literaturverzeichnis

[Kurtz] KURTZ, Stefan: *Foundations of Sequence Analysis*. – Lecture notes for a course in the Wintersemester 2015/2016

[Moffat u. Turpin 2002] MOFFAT, Alistair ; TURPIN, Andrew: *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002

[Myers 2015] MYERS, Eugene: *Recording Alignments with Trace Points*. https://dazzlerblog.wordpress.com/2015/11/05/trace-points/. Version: November 2015

[The SAM/BAM Format Specification Group 2015] THE SAM/BAM FORMAT SPECIFICATION GROUP: *Sequence Alignment/Map Format Specification*. https://samtools.github.io/hts-specs/SAMv1.pdf. Version: November 2015