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, verschiedene Repräsentationen von paarweisen Sequenzalignments und deren Kodierungen zu beschreiben und zu vergleichen, sowie basierend auf einer eigenen Implementierung einer speichereffizienten Repräsentation Unterschiede 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].

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 Kodierung eines CIGAR-Strings

Sei ein CIGAR-String $C = s_1c_1s_2c_2...s_nc_n$ mit $|C| = 2n$ eines Alignments mit $s_i \in \{\text{M,I,D}\}$ und $c_i \in \mathbb{N}, 0 < i \leq n$ gegeben.

Für die Kodierung eines CIGAR-Strings ist es sinnvoll, alle s_i und c_i seperat zu

kodieren, um so mit kleineren Alphabeten für die zu kodierenden Symbole arbeiten zu können.

$\mathcal{A}_s = (s_1, ..., s_n)$ und $\mathcal{A}_c = (c_1, ..., c_n)$ sind somit die Alphabete der zu kodierenden Symbole eines CIGAR-Strings mit den relativen Wahrscheinlichkeiten $p(x)$ jeden Symbols $x_s \in \mathcal{A}_s$ und $x_c \in \mathcal{A}_c$.

Im Folgenden vergleiche ich ausführlich für den CIGAR-String eines gegebenen Alignments die Verfahren der naiven binären Kodierung, der unären Kodierung und der Huffman-Kodierung.

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 `4M1I1M1I1M1I1M2D1M1D1M1I8M1D7M1D5M1I4M` repräsentiert werden kann.

Sei das Alphabet $\mathcal{A}_s = \{\text{M, I, D}\}$, welches alle Symbole aus dem CIGAR-String enthält, das Alphabet $\mathcal{A}_c = \{1, 2, 4, 5, 7, 8\}$, welches alle Zahlen aus dem CIGAR-String enthält, sowie die relativen Wahrscheinlichkeiten $p(x)$ jeden Symbols $x_s \in \mathcal{A}_s,\ x_c \in \mathcal{A}_c$ wie in Tabelle 2.1 dargestellt, gegeben.

Bei einer naiven binären Kodierung wird jedes Symbol $x_i \in \mathcal{A}_i$ 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 in Tabelle 2.2 beschrieben unär kodiert.

x_s	Häufigkeit	$p(x_s)$	x_c	Häufigkeit	$p(x_c)$
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

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.

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 [Matai u. a. 2014].

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 $(1, 2), (\text{M,D,I})$ und $(0, 2, 4), (1, 4, 2, 5, 7, 8)$ zu speichern. Diese können als soge-

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

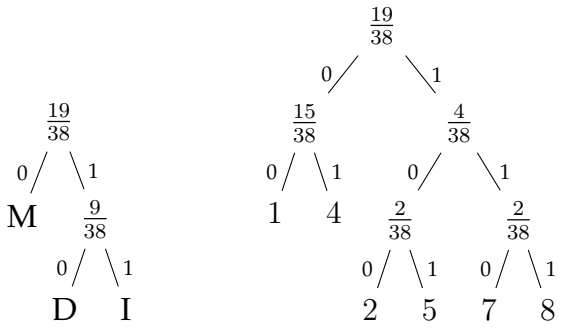

Abbildung 2.1: Huffman-Bäume der Kodierung des CIGAR-Strings

nannter 'Header' am Anfang der kodierten Datei unär kodiert werden. In diesem Fall würde der Header als $10110; 1110$,$011011110; 1111110$ kodiert werden, wobei jeweils der erste Teil die Häufigkeiten der Code-Längen und der zweite Teil die Gesamtanzahl der Symbole kodiert.

Die Größe dieser Kodierung ist demnach $28 + 42 + 9 + 16 = 95$ Bit.

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 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.

Im Folgenden Beispiel werde ich die Kodierung der Trace Point Differenzen ausführlich für die naive binäre, unäre und Huffman-Kodierung erläutern.

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.

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

Tabelle 2.4: Relative Wahrscheinlichkeiten der Differenzen-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 Differenzen-Kodierung

Für die Trace Point Darstellung ist somit eine Differenzen-Kodierung möglich.

Als neue Liste zu kodierender Werte ergibt sich nach 2.2.1 $L_{diff} = (5, 5, 5, 5, 4, 4, 6)$.

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

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

Sei das Alphabet $\mathcal{A} = \{4, 5, 6\}$ für L_{diff}, sowie die relativen Wahrscheinlichkeiten $p(x)$ jeden Symbols $x \in \mathcal{A}$ wie in Tabelle 2.4 dargestellt, gegeben.

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 und damit nur $\frac{24}{95} = 25.26\%$ der binären Kodierung für den CIGAR-String.

Die unäre Kodierung ergibt für dieses Beispiel die in Tabelle 2.5 beschriebene Kodierung. Die Größe dieser Kodierung ist demnach 12 Bit und benötigt damit nur $\frac{12}{67} = 17.91\%$ der unären Kodierung für den CIGAR-String.

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 Differenzen-Kodierung

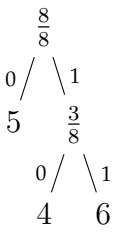

Abbildung 2.2: Huffman-Baum der Differenzen-Kodierung

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 2.1.1 beschrieben, nicht komplett gespeichert werden. Aufgrund der Beschaffenheit der Codewörter des kanonischen Huffman-Algorithmus ist hier lediglich die Speicherung der Listen $(1, 2), (5, 4, 6)$, welche als $10110; 1110$ im Header kodiert werden, nötig.

Die Größe der kanonischen Huffman-Kodierung ist demnach $11+10 = 21$ Bit und benötigt damit nur $\frac{21}{70} = 30\%$ der Huffman-Kodierung für den CIGAR-String.

2.3 Entropie der Methoden

Die Entropie H beschreibt den durchschnittlichen Informationsgehalt einer Nachricht X für alle Symbole $x \in X$ mit den relativen Wahrscheinlichkeiten $p(x)$ jeden Symbols in der Einheit $\frac{\text{Bit}}{\text{Symbol}}$.

$$H(X) = -\sum_{i=1}^{n} p_i \cdot \log_2 p_i$$

Sie ist maximal, wenn alle Symbole mit der gleichen Wahrscheinlichkeit $\frac{1}{n}$ auftreten [Mézard u. Montanari 2009].

Für den in 2.1.1 genannten CIGAR-String ergibt sich eine Entropie von

$$\begin{aligned} H(Cigar) &= -(\tfrac{10}{38} \cdot \log_2 \tfrac{10}{38} + \tfrac{5}{38} \cdot \log_2 \tfrac{5}{38} + \tfrac{4}{38} \cdot \log_2 \tfrac{4}{38} + \tfrac{13}{38} \cdot \log_2 \tfrac{13}{38} + \tfrac{2}{38} \cdot \\ &\quad \log_2 \tfrac{2}{38} + \tfrac{1}{38} \cdot \log_2 \tfrac{1}{38} + \tfrac{1}{38} \cdot \log_2 \tfrac{1}{38} + \tfrac{1}{38} \cdot \log_2 \tfrac{1}{38} + \tfrac{1}{38} \cdot \log_2 \tfrac{1}{38}) \\ &\approx 2.54 \, \tfrac{\text{Bit}}{\text{Symbol}} \end{aligned}$$

Die in 2.2.1 genannten Trace Point Differenzen des selben Alignments ergeben analog

$$\begin{aligned} H(Diff) &= -(\tfrac{5}{8} \cdot \log_2 \tfrac{5}{8} + \tfrac{2}{8} \cdot \log_2 \tfrac{2}{8} + \tfrac{1}{8} \cdot \log_2 \tfrac{1}{8}) \\ &\approx 1.30 \, \tfrac{\text{Bit}}{\text{Symbol}} \end{aligned}$$

3 Resultate

3.1 Testläufe CIGAR Kodierung

Die ersten drei der 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 und auf 10 Bit gerundet. Die letzten drei der Grafiken wurden mit jeweils 100 zufällig generierten Sequenzpaaren mit je etwa 10.000 Basen, einer Fehlerrate von 15% und einem Δ-Wert von 100 berechnet und auf 10 Bit gerundet.

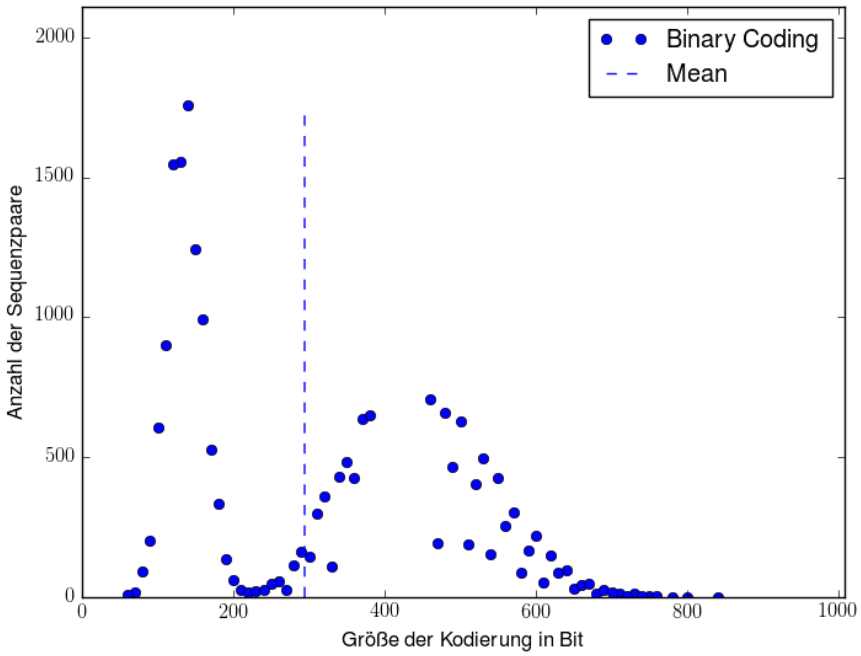

Abbildung 3.1: Größe der naiven binären Kodierung für einen CIGAR-String eines paarweisen Sequenz-Alignments in Bit

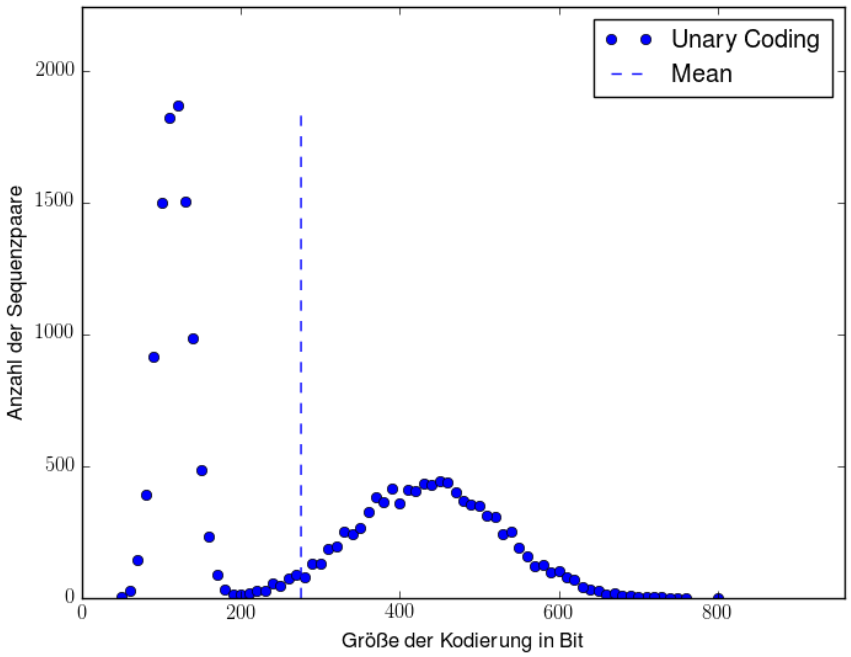

Abbildung 3.2: Größe der unären Kodierung für einen CIGAR-String eines paarweisen Sequenz-Alignments in Bit

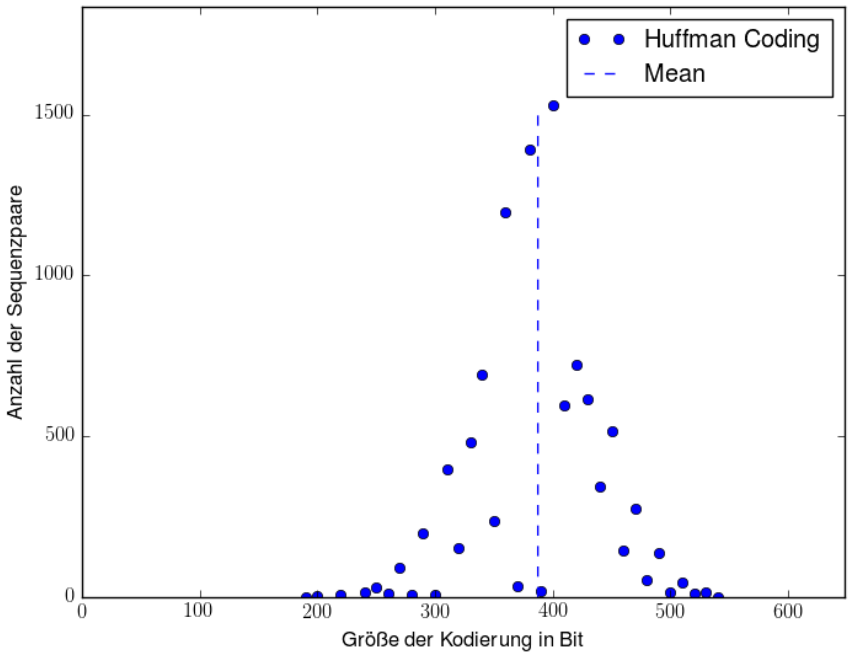

Abbildung 3.3: Größe der Huffman-Kodierung für einen CIGAR-String eines paarweisen Sequenz-Alignments in Bit

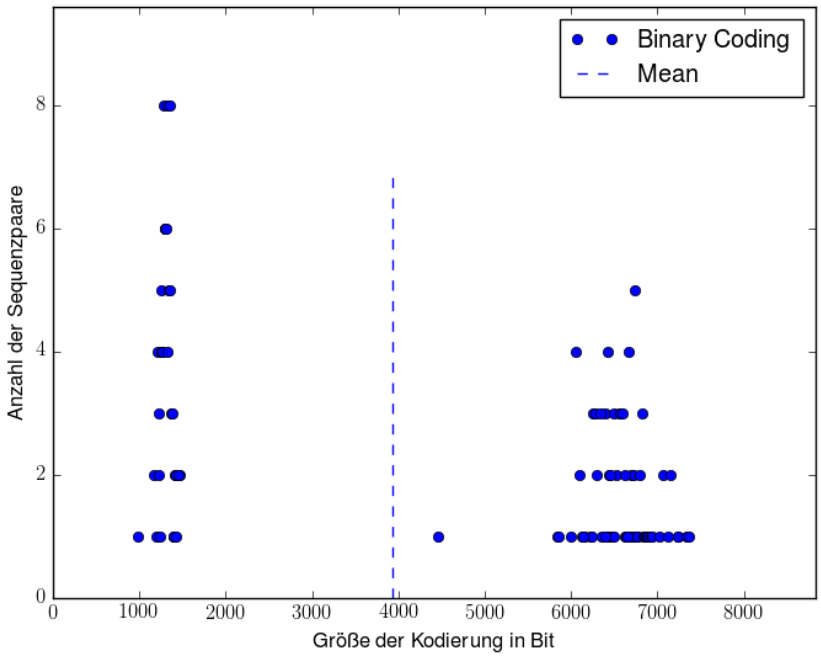

Abbildung 3.4: Größe der binären Kodierung für einen CIGAR-String eines paarweisen Sequenz-Alignments in Bit

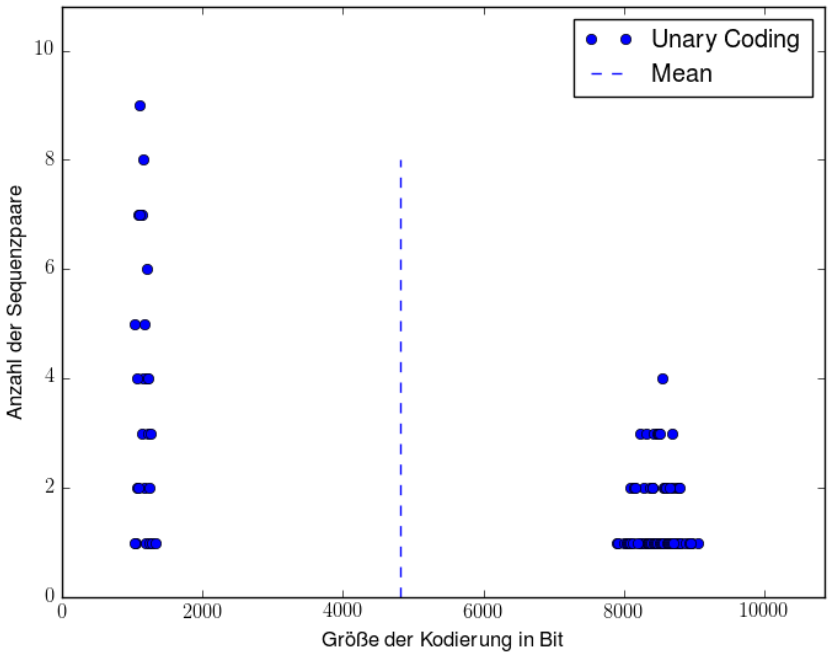

Abbildung 3.5: Größe der unären Kodierung für einen CIGAR-String eines paarweisen Sequenz-Alignments in Bit

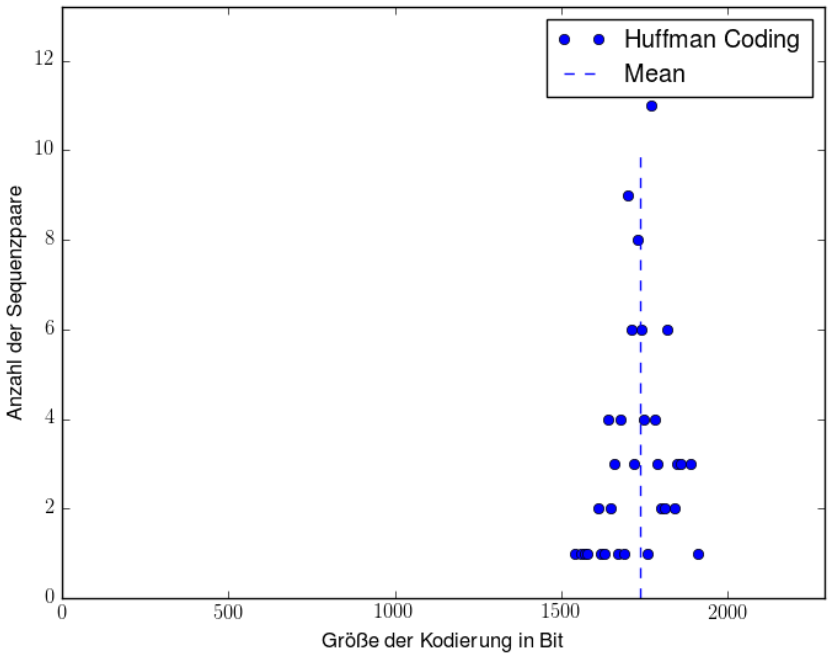

Abbildung 3.6: Größe der Huffman-Kodierung für einen CIGAR-String eines paarweisen Sequenz-Alignments in Bit

Die naive binäre Kodierung der CIGAR-Strings benötigt wie in Abbildung 3.1 dargestellt für 10000 Sequenzpaare mit je 1000 Basen im Mittel 294.27 Bit, wobei Werte zwischen 50 und 900 Bit erreicht werden. Diese bilden zwei Maxima bei 1800 Sequenzpaaren mit jeweils etwa 120 Bit als globales Maximum und 700 Sequenzpaaren mit jeweils etwa 450 Bit als lokales Maximum. Für 100 Sequenzpaare mit je 10000 Basen werden, wie in Abbildung 3.4 zu sehen ist, Werte zwischen 900 und 7500 Bit mit einem Mittelwert von 3942.06 Bit erreicht, wobei hier eine ähnliche Verteilung wie in Abbildung 3.1 auftritt. Die Maxima liegen hier bei 8 Sequenzpaaren mit 1100 Bit und 5 Sequenzpaare mit 6700 Bit.

Wie in Abbildung 3.2 zu erkennen ist, ähnelt die Anzahl der Bits für die unäre Kodierung der CIGAR-Strings für 10000 Sequenzpaare mit je 1000 Basen der der naiven binären Kodierung. Hier werden ebenfalls zwei Maxima erreicht, wobei hier das globale Maximum bei 1900 Sequenzpaaren mit etwa 150 Bit und das lokale Maximum bei 500 Sequenzpaaren mit etwa 450 Bit liegt. Für die unäre Kodierung wird im Mittel 275.70 Bit für die Kodierung eines CIGAR-Strings und damit nur etwa 6.3% weniger als bei der naiven binären Kodierung benötigt. Für 100 Sequenzpaare mit je 10000 Basen werden, wie in Abbildung 3.5 dargestellt, ebenfalls zwei Maxima erreicht, einmal bei 9 Sequenzpaaren zu 1700 Bit und 4 Sequenzpaare mit 8500 Bit. Der Mittelwert liegt hier bei 4794.50 Bit.

Die Huffman-Kodierung kodiert wie in Abbildung 3.3 zu erkennen ist die CIGAR-Strings für 10000 Sequenzpaare mit je 1000 Basen zwischen 200 und 550 Bit. Die Bitanzahl ist hier nahezu normalverteilt mit einem Maximum von 1500 Sequenzpaaren für etwa 390 Bit, was in etwa dem Durchschnitt von 387.38 Bit entspricht. Für 100 Sequenzpaare mit je 10000 Basen werden, wie in Abbildung 3.6 zu sehen ist, zwischen 1500 und 1900 Bit benötigt, wobei das Maximum bei 11 Sequenzpaaren mit 1750 Bit liegt und damit dem Mittelwert von 1734.18 Bit entspricht.

3.2 Testläufe Differenzen Kodierung

Die ersten beiden der 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 und auf 3 Bit gerundet. Die letzten beiden Grafiken wurden mit jeweils 100 zufällig generierte Sequenzpaaren mit je etwa 10.000

Basen, einer Fehlerrate von 15% und einem Δ-Wert von 100 berechnet und auf 3 Bit gerundet.

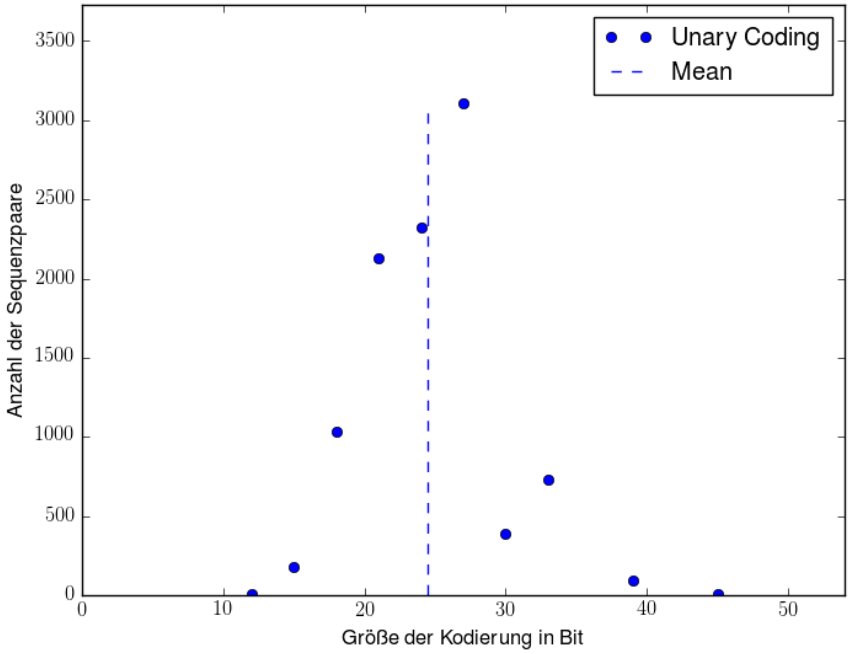

Abbildung 3.7: Größe der unären Kodierung für die Differenzen der Trace Points eines paarweisen Sequenz-Alignments in Bit

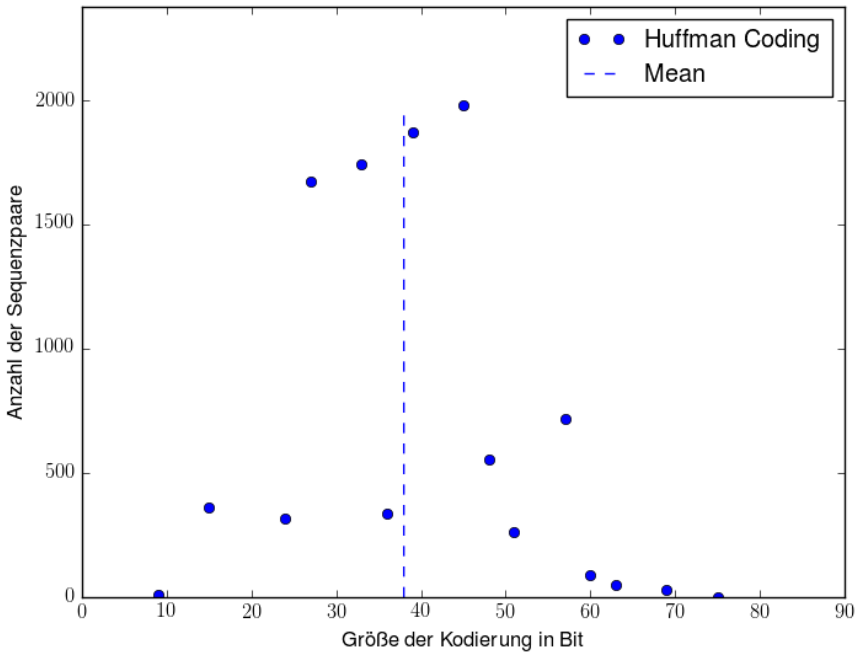

Abbildung 3.8: Größe der Huffman-Kodierung für die Differenzen der Trace Points eines paarweisen Sequenz-Alignments in Bit

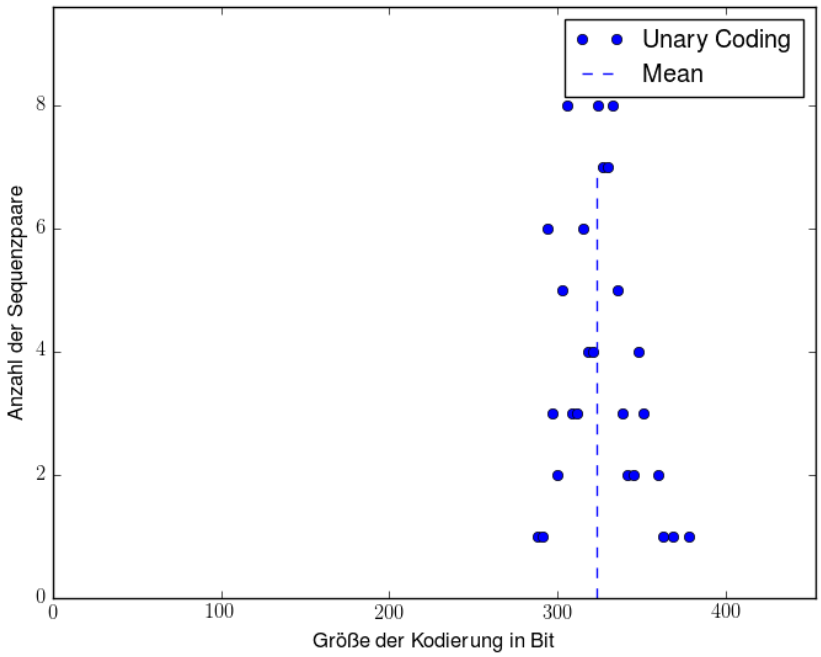

Abbildung 3.9: Größe der unären Kodierung für die Differenzen der Trace Points eines paarweisen Sequenz-Alignments in Bit

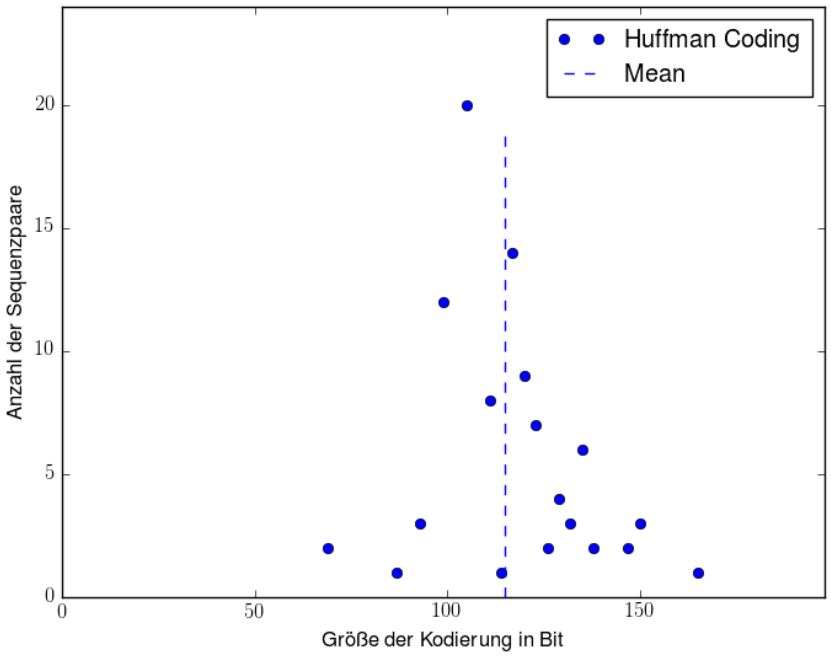

Abbildung 3.10: Größe der Huffman-Kodierung für die Differenzen der Trace Points eines paarweisen Sequenz-Alignments in Bit

Die naive binäre Kodierung benötigt für 10000 Sequenzpaare mit 1000 Basen und einem Δ-Wert von 100 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. Für 100 Sequenzpaare mit 10000 und einem Δ-Wert von 100 werden in jedem Durchlauf und im Mittel 700 Bit verbraucht, da $\lceil \log_2 100 \rceil \cdot 100 = 700$ Bit für die binäre Kodierung benötigt werden.

Für die unäre Kodierung der Differenzen der Trace Points wird für 10000 Sequenzpaare mit 1000 Basen, wie in Abbildung 3.7 verdeutlicht, zwischen 12 und 45 Bit benötigt. Die Bitanzahl ist in etwa normalverteilt, wobei das Maximum bei 3050 Sequenzpaaren mit 26 Bit liegt. Im Mittel wird für diese Kodierung 24.49 Bit und damit nur etwa 61.22% der binären Kodierung benötigt. Für 100 Sequenzpaare mit je 10000 Basen werden, wie in Abbildung 3.9 dargestellt, zwischen 290 und 390 Bit mit einem Maximum bei 8 Sequenzpaaren und 320 Bit benötigt, wobei der Mittelwert bei 323.2 Bit liegt.

Die Huffman-Kodierung kodiert für 10000 Sequenzpaare mit 1000 Basen, wie in Abbildung 3.8 zu sehen ist, die Differenzen der Trace Points mit 3 bis 30 Bit, wobei hier das Maximum bei 2600 Sequenzpaaren mit 15 Bit liegt. Sie verbraucht durchschnittlich nur 13.90 Bit und damit nur etwa 56.75% des Speicherbedarfs der unären und 34.74% der naiven binären Kodierung. Für 100 Sequenzpaare mit 10000 Basen wird, wie in Abbildung 3.10 dargestellt, zwischen 70 und 170 Bit benötigt, wobei das Maximum bei 20 Sequenzpaaren mit 110 Bit und der Mittelwert bei 114.04 Bit liegt.

3.3 Testläufe Entropie der Repräsentationen

Die folgenden Grafiken zeigen unabhängig vom Kodierungsverfahren die Entropie der CIGAR-Strings und Trace Point Differenzen von 10.000 zufällig generierten Sequenzpaaren mit je etwa 1.000 Basen, einer Fehlerrate von 15% und einem Δ-Wert von 100.

Die Entropie für die CIGAR-String Repräsentation liegt für die gegebenen Sequenzpaare, wie in Abbildung 3.11 zu erkennen ist, zwischen 110 und 580 Bit, wobei der Durchschnittswert bei 323.98 Bit liegt. Die Werte sind normalverteilt, wobei das Maximum bei 700 Sequenzpaaren, welche mit etwa 320 Bit kodiert wurden, liegt.

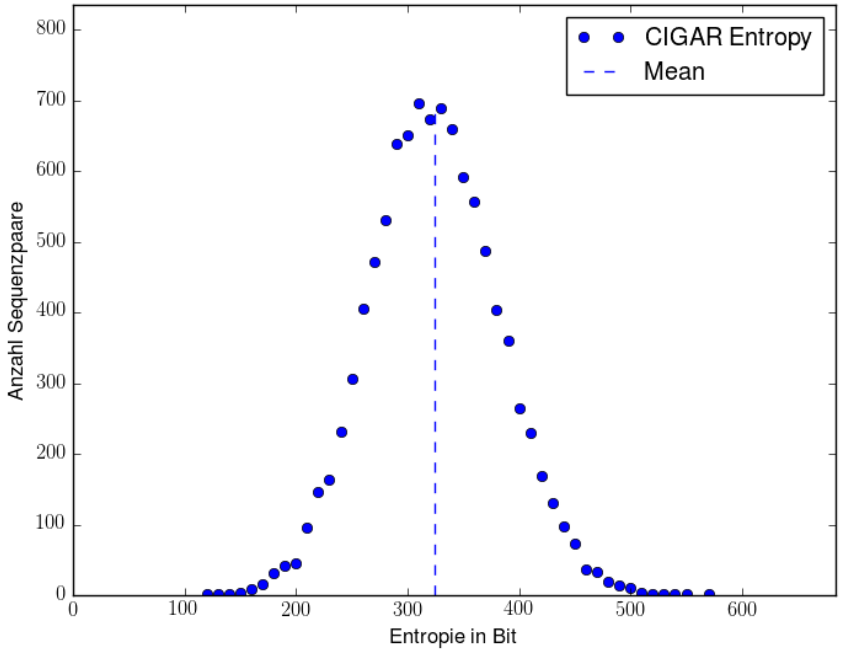

Abbildung 3.11: Entropie des CIGAR-Strings

Die Differenzen der Trace Points weisen hingegen, wie in Abbildung 3.12 verdeutlicht, eine Entropie im Bereich von 6 bis 30 Bit auf, wobei der Durchschnittswert von 21.53 Bit nur etwa 7% des Durchschnittwertes der CIGAR-String Repräsentation ausmacht. Das Maximum der nahezu normalverteilten Entopie-Werte liegt hier bei etwa 21 Bit, mit welchem 4000 Sequenzpaare kodiert wurden.

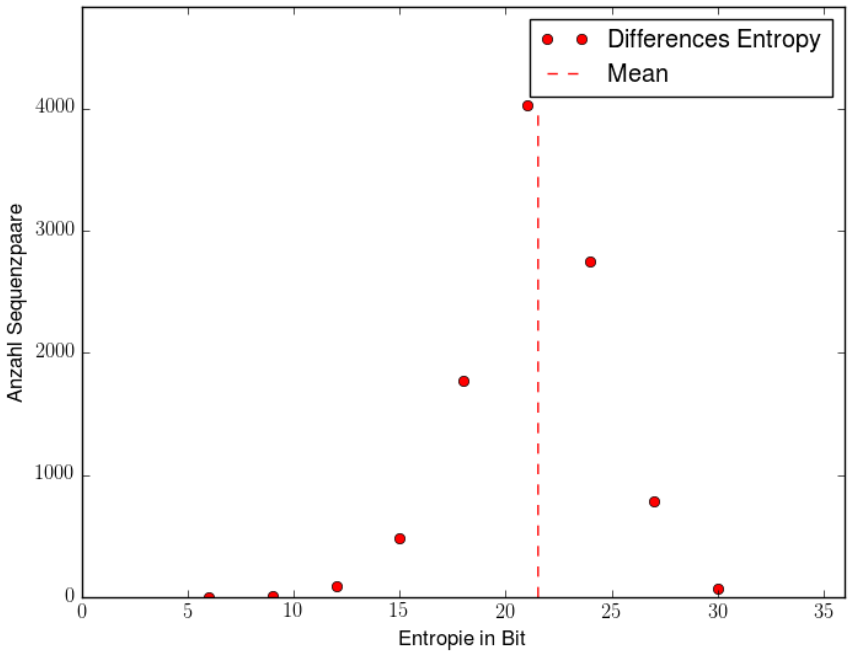

Abbildung 3.12: Entropie der Trace Point Differenzen

4 Diskussion

TODO

4.1 Bewertung CIGAR-Kodierung

4.2 Bewertung Kodierung der Differenzen der Trace Points

4.3 Bewertung Entropie beider Methoden

5 Programm

5.1 Aufbau

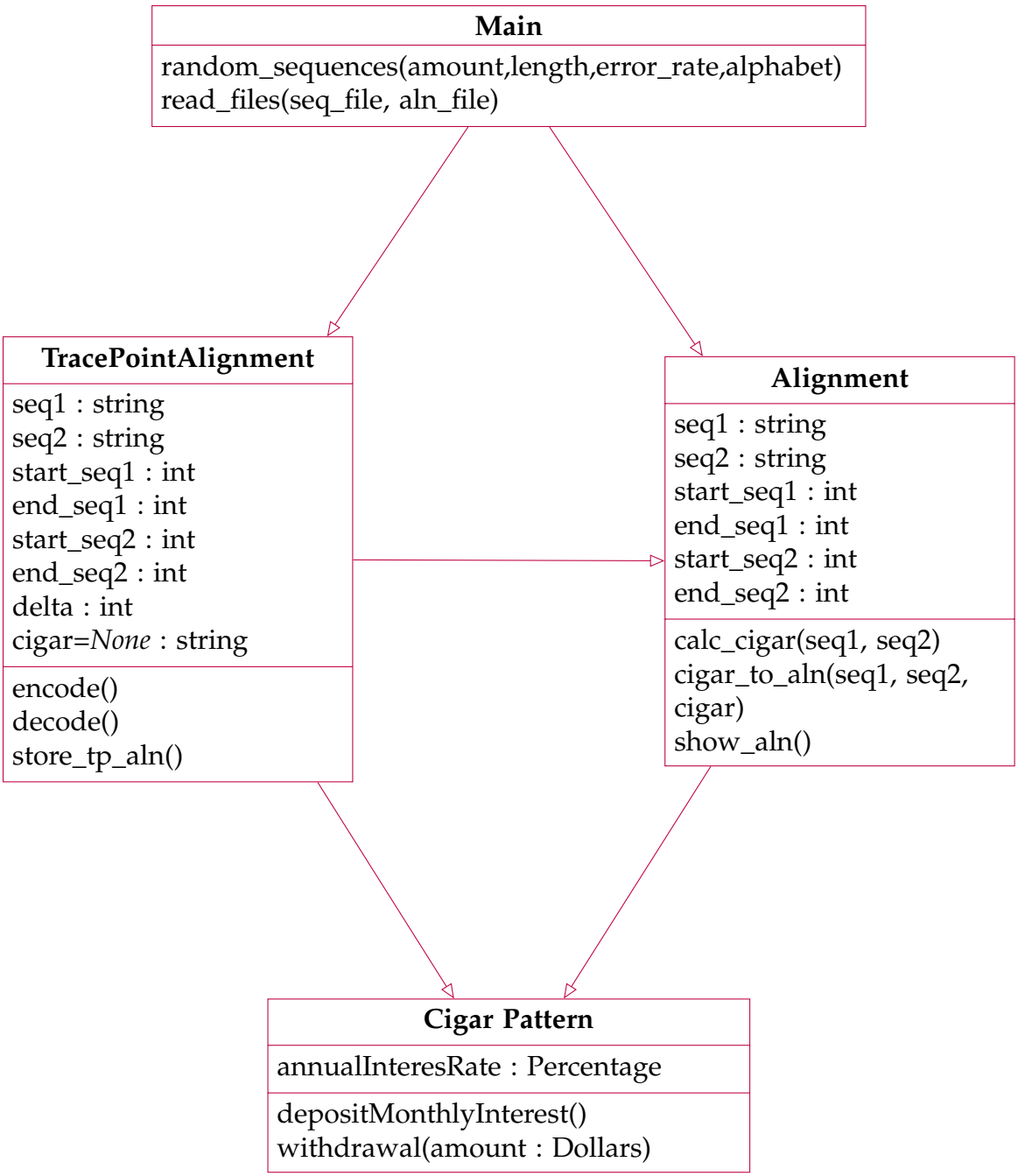

Abbildung 5.1: UML-Diagramm

5.2 Funktionalität

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 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*($seq1, seq2, start\_seq1, end\_seq1, start\_seq2, \Delta, cigar$)
  $p \leftarrow MAX(1, \lceil start\_seq1/\Delta \rceil)$
  $\tau \leftarrow \lceil end\_seq1/\Delta \rceil - \lfloor start\_seq2/\Delta \rfloor$
  $uTP \leftarrow$ Array for interval termini in the first sequence
  for $i \leftarrow 0$ **upto** $|\tau|$ **do**
    $uTP[i] \leftarrow (p + i) \cdot (\Delta - 1)$
  end for
  $uChars, vChars, count \leftarrow 0$
  $TP \leftarrow$ 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 $uChars$
      else if cig_symbol = 'D' **then**
        increment $vChars$
      else
        increment $uChars, vChars$
      end if
      if $uChars = uTP[count]$ **then**
        TP.append($vChars$)
      end if
      if $count \neq |uTP| - 1$ **then**
        return TP
      else
        increment $count$
      end if
    end for
  end for
end function

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: cig.append(**cigar**($seq1[0...\Delta], seq2[0...TP[i]+1]$))
6: **else if** i = |TP| - 1 **then**
7: cig.append(**cigar**($seq1[i \cdot \Delta ... |seq1|], seq2[TP[i-1]+1 ... |seq2|]$))
8: **else**
9: cig.append(**cigar**($seq1[i \cdot \Delta ... (i+1) \cdot \Delta], seq2[TP[i-1]+1] ... TP[i]+1$))
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 is in cigar **then**
27: cig.append($tmp + cig\_count, cig\_symbol$)
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

[Matai u. a. 2014] MATAI, Janarbek ; KIM, Joo-Young ; KASTNER, Ryan: *Energy Efficient Canonical Huffman Encoding*. https://www.zurich.ibm.com/asap2014/presentations/day2/ses6_ASAP2014_Final-kastner.pdf. Version: 2014. – Presentation at IBM Research - Zurich

[Mézard u. Montanari 2009] MÉZARD, Marc ; MONTANARI, Andrea: *Information, Physics and Computation*. Oxford University Press, 2009

[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