



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

1	Einleitung	1
2	CIGAR-Strings	3
2.0.1	Komplexität	3
2.0.2	Speicherverbrauch	3
2.0.3	Grafiken	6
3	TracePoint Konzept	7
3.0.1	Komplexität	9
3.0.2	Speicherverbrauch	9
3.0.3	Bewertung	11
3.0.4	Grafiken	11
4	Optimierung	13
4.0.1	Delta-Kodierung	13
4.0.2	Grafiken Vergleich	13
5	Programm	15
5.0.1	Aufbau	15
5.0.2	Funktionalität	15
5.0.3	Grafiken Geschwindigkeit, Speicher	15
6	Fazit	19
	Literaturverzeichnis	21
	Eidesstattliche Erklärung	23

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 gegebene 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.

Die Edit-Operationen

Die in diesem Dokument 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

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

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 $(\alpha_1 \rightarrow \beta_1, \dots, \alpha_h \rightarrow \beta_h)$ von Edit-Operationen definieren, sodass $u = \alpha_1 \dots \alpha_h$ und $v = \beta_1 \dots \beta_h$ 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 $A = (\alpha_1 \rightarrow \beta_1, \dots, \alpha_h \rightarrow \beta_h)$ 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:

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

Ein Alignment A ist optimal, wenn $\delta(A) = \text{edist}_\delta(u, v)$ gilt.

Wenn δ die Einheitskostenfunktion ist, so ist $\text{edist}_\delta(u, v)$ 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 CIGAR-Strings

Ein Dateiformat, welches zur Speicherung von Alignments verwendet wird, ist das SAM-Format oder die 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 $u = \text{actgaact}$, $v = \text{actagaat}$ und das Alignment $A = (a \rightarrow a, c \rightarrow c, t \rightarrow t, \dots)$ 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 durch den Cigar-String 3M1I3M1D1M repräsentiert. Das Format benötigt wenig Speicher für Alignments mit einer kleinen Edit-Distanz und deutlich mehr Speicher für Alignments mit einer großen Edit-Distanz [The SAM/BAM Format Specification Group 2015].

2.0.1 Komplexität

2.0.2 Speicherverbrauch

Sei das Alignment A

0	5	0	5	0	5	0	5	0
g	a	g	c	c	c	t	t	t
g	a	c	c	c	t	t	t	t
g	a	g	c	c	c	t	t	t
g	a	c	c	c	t	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g	a	g	c	c	c	t	t	t
g								

```
gaccaagtag--g-cgtggacctt-gctcggt-ctgtaagaga
0      5      0      5      0      5      0      5      0
```

und der dazugehörige CIGAR-String 4M1D1M1D1M1D1M2I1M1I1M1D8M1I7M1I5M1D4M gegeben.

Mit einer naiven binären Kodierung bräuchte man demnach für jedes Symbol $c \in \mathcal{A}$ $\log_2(38) = 6$ Bit pro Symbol, also $38 \cdot 6 = 228$ Bit insgesamt.

Bei einer *minimalen* binären Kodierung, wie sie auch im Huffman-Algorithmus 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 Bits pro Symbol beansprucht [Moffat u. Turpin 2002].

Der Huffman-Algorithmus würde bei dem oben genannten Beispiel der TracePoints wie folgt ablaufen:

Gegeben sei das Alphabet $\mathcal{A} = \{M, I, D, 1, 2, 4, 5, 7, 8\}$, sowie die relativen Wahrscheinlichkeiten $p(i)$ jeden Symbols aus \mathcal{A} mit

$$\begin{aligned}
 p(1) &= \frac{13}{38} \\
 p(M) &= \frac{10}{38} \\
 p(D) &= \frac{5}{38} \\
 p(I) &= \frac{4}{38} \\
 p(4) &= \frac{2}{38} \\
 p(2) &= \frac{1}{38} \\
 p(5) &= \frac{1}{38} \\
 p(7) &= \frac{1}{38} \\
 p(8) &= \frac{1}{38}
 \end{aligned}$$

Ausführung des Huffman-Algorithmus:

$$\begin{array}{ccccccc}
\frac{13}{38} & c_1 = \lambda & \frac{13}{38} & c_1 = \lambda & \frac{13}{38} & c_1 = \lambda & \frac{13}{38} & c_1 = \lambda \\
\frac{10}{38} & c_2 = \lambda & \frac{10}{38} & c_2 = \lambda & \frac{10}{38} & c_2 = \lambda & \frac{10}{38} & c_2 = \lambda \\
\frac{5}{38} & c_3 = \lambda & \frac{5}{38} & c_3 = \lambda & \frac{5}{38} & c_3 = \lambda & \frac{5}{38} & c_3 = \lambda \\
\frac{4}{38} & c_4 = \lambda & \Rightarrow & \frac{4}{38} & c_4 = \lambda & \Rightarrow & \frac{4}{38} & c_4 = \lambda & \Rightarrow \\
\frac{2}{38} & c_5 = \lambda & \frac{2}{38} & c_5 = \lambda & \frac{2}{38} & c_5 = \lambda & \frac{4}{38} & c_5 = 0, c_6 = 10, \\
\frac{1}{38} & c_6 = \lambda & \frac{2}{38} & c_6 = 0, c_7 = 1 & \frac{2}{38} & c_6 = 0, c_7 = 1 & & c_7 = 11 \\
\frac{1}{38} & c_7 = \lambda & \frac{1}{38} & c_8 = \lambda & \frac{2}{38} & c_8 = 0, c_9 = 1 & \frac{2}{38} & c_8 = 0, c_9 = 1 \\
\frac{1}{38} & c_8 = \lambda & \frac{1}{38} & c_9 = \lambda & & & & \\
\frac{1}{38} & c_9 = \lambda & & & & & &
\end{array}$$

$$\begin{array}{ccc}
\frac{13}{38} & c_1 = \lambda & \frac{13}{38} & c_1 = \lambda \\
\frac{10}{38} & c_2 = \lambda & \frac{10}{38} & c_2 = \lambda \\
\frac{5}{38} & c_3 = \lambda & \Rightarrow & \frac{6}{38} & c_4 = 0, c_8 = 10, c_9 = 11 & \Rightarrow \\
\frac{4}{38} & c_4 = \lambda & \frac{5}{38} & c_3 = \lambda \\
\frac{4}{38} & c_5 = 0, c_6 = 10, c_7 = 11 & \frac{4}{38} & c_5 = 0, c_6 = 10, c_7 = 11 \\
\frac{2}{38} & c_8 = 0, c_9 = 1 & &
\end{array}$$

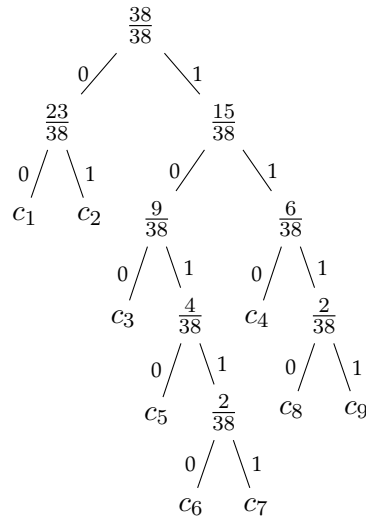
$$\begin{array}{ccc}
\frac{13}{38} & c_1 = \lambda & \frac{15}{38} & c_3 = 00, c_5 = 010, c_6 = 0110, c_7 = 0111, \\
\frac{10}{38} & c_2 = \lambda & & c_4 = 10, c_8 = 110, c_9 = 111 \\
\frac{9}{38} & c_3 = 0, c_5 = 10, c_6 = 110, c_7 = 111 & \Rightarrow & \frac{13}{38} & c_1 = \lambda \\
\frac{6}{38} & c_4 = 0, c_8 = 10, c_9 = 11 & & \frac{10}{38} & c_2 = \lambda & \Rightarrow
\end{array}$$

$$\begin{array}{ccc}
\frac{23}{38} & c_1 = 0, c_2 = 1 & \frac{38}{38} & c_1 = 00, c_2 = 01, c_3 = 100, c_4 = 110, \\
\frac{15}{38} & c_3 = 00, c_5 = 010, c_6 = 0110, c_7 = 0111, & \Rightarrow & c_5 = 1010, c_6 = 10110, c_7 = 10111, \\
& c_4 = 10, c_8 = 110, c_9 = 111 & & c_8 = 1110, c_9 = 1111
\end{array}$$

Somit ergibt sich die Menge C der Codewörter mit

$$C = "00", "01", "100", "110", "1010", "10110", "10111", "1110", "1111"$$

und der dazugehörige Huffman-Baum:



Der gesamte Bitverbrauch dieser Kodierung ist demnach 32 Bit.

Der durchschnittliche Bitverbrauch für ein Symbol beträgt

$$\frac{38}{38} + \frac{23}{38} + \frac{15}{38} + \frac{9}{38} + \frac{6}{38} + \frac{4}{38} + \frac{2}{38} + \frac{2}{38} \approx 2.61 \text{ Bit.}$$

Die Entropie beträgt

$$\begin{aligned}
 H(X) &= -\left(\frac{13}{38} \cdot \log_2\left(\frac{13}{38}\right) + \frac{10}{38} \cdot \log_2\left(\frac{10}{38}\right) + \frac{5}{38} \cdot \log_2\left(\frac{5}{38}\right) + \right. \\
 &\quad \left. \frac{4}{38} \cdot \log_2\left(\frac{4}{38}\right) + \frac{2}{38} \cdot \log_2\left(\frac{2}{38}\right) + \frac{1}{38} \cdot \log_2\left(\frac{1}{38}\right) + \right. \\
 &\quad \left. \frac{1}{38} \cdot \log_2\left(\frac{1}{38}\right) + \frac{1}{38} \cdot \log_2\left(\frac{1}{38}\right) + \frac{1}{38} \cdot \log_2\left(\frac{1}{38}\right)\right) \\
 &\approx 2.72 \text{ Bit je Symbol}
 \end{aligned}$$

2.0.3 Grafiken

3 TracePoint 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 $v[k...l]$ 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 $u_0, u_1, \dots, u_{\tau-1}$ 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 2:

Sequenz 1: gagcatgttgccctggctcctttgctaggtactgtagaga

Sequenz 2: gaccaagtaggcgtggaccttgctcggctctgtaagaga

Delta: 15

Gesamtalignment:

```

0      5      0      5      0      5      0      5      0
gagc-a-t-gttgcc-tggctcctttgctaggtactgta-gaga
|| | | | | | | ||| ||| ||| ||| ||| |||
gaccaagtag--g-cgtggacctt-gctcggctctgtaagaga
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]

Berechnung der Intervalle anhand der Trace Points:

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

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

3.0.1 Komplexität

Für die Trace-Point Repräsentation wird für eine Edit-Distanz e mit Einheitskosten als Kostenfunktion δ wie oben beschrieben lediglich $O(e^2)$ Zeit pro Teilalignment benötigt, wobei bei einer erwarteten Fehlerrate ε des Alignments die Edit-Distanz immer höchstens so groß ist wie die Anzahl der Fehler im Teilalignment. [Kurtz, S.41-42]

3.0.2 Speicherverbrauch

Sei $\Delta = 5$ und das Alignment \mathcal{A}

```

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

```

wie in Abschnitt 2.0.2 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 einer naiven binären Kodierung bräuchte man demnach für jedes Symbol $c \in \mathcal{A}$ $\lceil \log_2(7) \rceil = 3$ Bit pro Symbol, also $7 \cdot 3 = 21$ Bit insgesamt.

Der Huffman-Algorithmus würde bei dem oben genannten Beispiel der TracePoints wie folgt ablaufen:

Gegeben sei das Alphabet $\mathcal{A} = \{5, 10, 15, 20, 24, 28, 34\}$, sowie die relativen Wahrscheinlichkeiten $p(i)$ jeden Symbols aus \mathcal{A} mit

$$\begin{aligned}
 p(2) &= \frac{3}{13} \\
 p(5) &= \frac{2}{13} \\
 p(1) &= \frac{2}{13} \\
 p(0) &= \frac{2}{13} \\
 p(4) &= \frac{2}{13} \\
 p(8) &= \frac{1}{13} \\
 p(3) &= \frac{1}{13}
 \end{aligned}$$

Ausführung des Huffman-Algorithmus:

$$\begin{array}{lll}
\frac{3}{13} & c_1 = \lambda & \frac{3}{13} & c_1 = \lambda & \frac{4}{13} & c_2 = 0, c_3 = 1 \\
\frac{2}{13} & c_2 = \lambda & \frac{2}{13} & c_2 = \lambda & \frac{3}{13} & c_1 = \lambda \\
\frac{2}{13} & c_3 = \lambda & \frac{2}{13} & c_3 = \lambda & \frac{2}{13} & c_4 = \lambda \\
\frac{2}{13} & c_4 = \lambda \Rightarrow & \frac{2}{13} & c_4 = \lambda & \Rightarrow & \frac{2}{13} & c_5 = \lambda & \Rightarrow \\
\frac{2}{13} & c_5 = \lambda & \frac{2}{13} & c_5 = \lambda & \frac{2}{13} & c_6 = 0, c_7 = 1 \\
\frac{1}{13} & c_6 = \lambda & \frac{2}{13} & c_6 = 0, c_7 = 1 \\
\frac{1}{13} & c_7 = \lambda
\end{array}$$

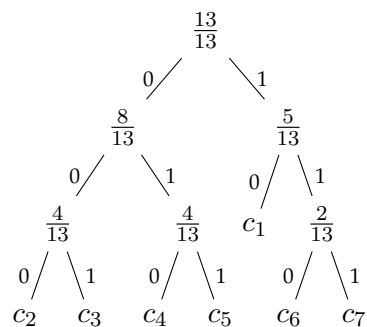
$$\begin{array}{lll}
\frac{4}{13} & c_2 = 0, c_3 = 1 & \frac{5}{13} & c_1 = 0, c_6 = 10, c_7 = 11 & \frac{8}{13} & c_2 = 00, c_3 = 01, c_4 = 10, \\
\frac{4}{13} & c_4 = 0, c_5 = 1 \Rightarrow & \frac{4}{13} & c_2 = 0, c_3 = 1 & \Rightarrow & c_5 = 11 \\
\frac{3}{13} & c_1 = \lambda & \frac{4}{13} & c_4 = 0, c_5 = 1 & \frac{5}{13} & c_1 = 0, c_6 = 10, c_7 = 11 \\
\frac{2}{13} & c_6 = 0, c_7 = 1
\end{array}$$

$$\Rightarrow \frac{13}{13} \quad c_1 = 10, c_2 = 000, c_3 = 001, c_4 = 010, c_5 = 011, c_6 = 110, c_7 = 111$$

Somit ergibt sich die Menge C der Codewörter mit

$$C = \{ "10", "000", "001", "010", "011", "110", "111" \}$$

und der dazugehörige Huffman-Baum:



Der gesamte Bitverbrauch dieser Kodierung ist demnach 20 Bit.

Der durchschnittliche Bitverbrauch für ein Symbol beträgt

$$\frac{13}{13} + \frac{8}{13} + \frac{5}{13} + \frac{4}{13} + \frac{4}{13} + \frac{2}{13} \approx 2.77 \text{ Bit.}$$

Die Entropie beträgt

$$\begin{aligned}
H(X) &= -\left(\frac{3}{13} \cdot \log_2\left(\frac{3}{13}\right) + \frac{2}{13} \cdot \log_2\left(\frac{2}{13}\right) + \frac{2}{13} \cdot \log_2\left(\frac{2}{13}\right) + \frac{2}{13} \cdot \log_2\left(\frac{2}{13}\right) + \right. \\
&\quad \left. \frac{2}{13} \cdot \log_2\left(\frac{2}{13}\right) + \frac{1}{13} \cdot \log_2\left(\frac{1}{13}\right) + \frac{1}{13} \cdot \log_2\left(\frac{1}{13}\right)\right) \\
&\approx 2.72 \text{ Bit je Symbol}
\end{aligned}$$

Die Repräsentation des Alignments \mathcal{A} benötigt als CIGAR-String mit einer naiven binären Kodierung 228 Bit und als Huffman-Kodierung 32 Bit, wobei die Kodierung der Trace Points mit $\Delta = 5$ mit einer naiven binären Kodierung 21 Bit und als Huffman-Kodierung 20 Bit benötigt.

Es ist somit zu erkennen, dass die Kodierung der Trace Points in diesem Fall mit einem relativ klein gewählten Δ weniger Speicher benötigt, als die Kodierung des CIGAR-Strings. Dennoch hängt der Speicherverbrauch der Trace Points Kodierung von der Wahl des Δ ab, da bei einem kleinen Δ mehr Trace Points und somit Symbole gespeichert werden müssen, als bei einem großen Δ und kann somit bei einer sehr ungünstig gewählten Größe mehr Speicher verbrauchen als ein CIGAR-String.

3.0.3 Bewertung

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.

3.0.4 Grafiken

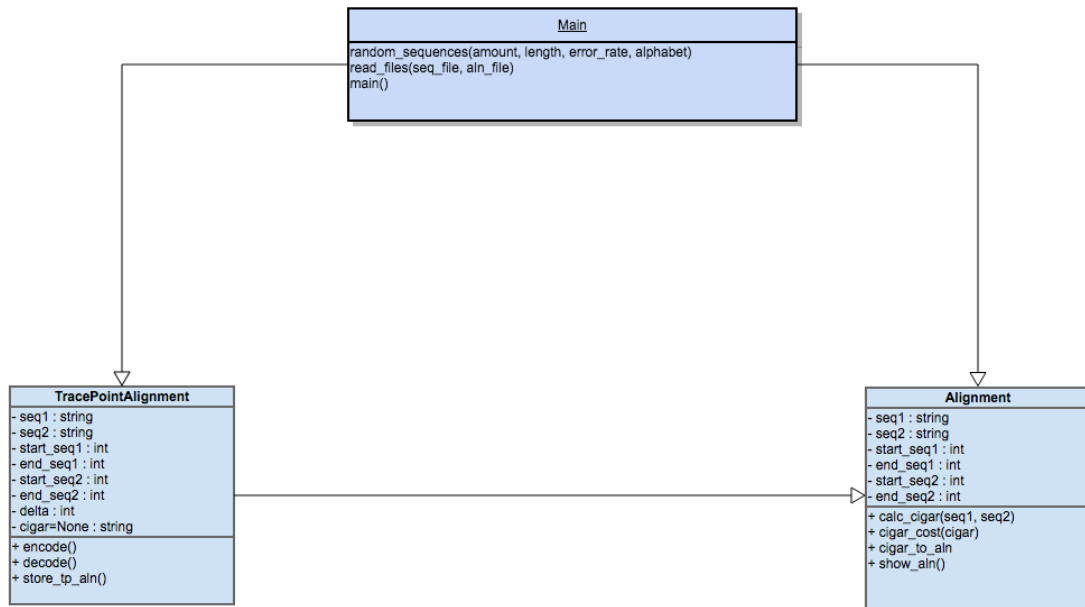
4 Optimierung

4.0.1 Delta-Kodierung

4.0.2 Grafiken Vergleich

5 Programm

5.0.1 Aufbau



5.0.2 Funktionalität

Informationsverlust bei 'encode()'

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 der jeweiligen Intervall-Paare errechnet werden.

5.0.3 Grafiken Geschwindigkeit, Speicher

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

```

1: function encode( $seq1, seq2, start\_seq1, end\_seq1, start\_seq2, \Delta, cigar$ )
2:    $itv\_size \leftarrow MAX(1, \lceil start\_seq1 / \Delta \rceil)$ 
3:    $itv\_count \leftarrow MIN(\lceil |seq1| / \Delta \rceil, \lceil |seq2| / \Delta \rceil)$ 
4:   for  $i \leftarrow 0$  upto  $|itv\_count|$  do
5:      $itv[i] \leftarrow \begin{cases} start\_seq1, itv\_size \cdot \Delta - 1 & \text{if } i = 0 \\ (itv\_size + i - 1) \cdot \Delta, (itv\_size + i) \cdot \Delta - 1 & \text{if } 0 < i < |itv\_count| \\ (itv\_size + i - 1) \cdot \Delta, end\_seq1 - 1 & \text{else.} \end{cases}$ 
6:   end for
7:    $count1, count2, count3 \leftarrow 0$ 
8:    $TP \leftarrow$  Array for Trace Points
9:   for each ( $cig\_count, cig\_symbol$ ) in  $cigar$  do
10:    for  $i \leftarrow 0$  upto  $cig\_count$  do
11:      if  $cig\_symbol = 'T'$  then
12:        increment  $count1$ 
13:      else if  $cig\_symbol = 'D'$  then
14:        increment  $count2$ 
15:      else
16:        increment  $count1, count2$ 
17:      end if
18:      if  $count1 = intervals[count3][1] + 1$  and  $count1 \neq |seq1|$  then
19:        append ( $count2 - 1 + start\_seq2$ ) to  $TP$ 
20:      end if
21:      if  $count \neq |itv| - 1$  then
22:        increment  $count3$ 
23:      end if
24:    end for
25:  end for
26:  return  $TP$ 
27: 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:     append to cig:
       
$$\begin{cases} \text{cigar}(seq1[0... \Delta], seq2[0...TP[i] + 1]) & \text{if } i = 0 \\ \text{cigar}(seq1[i \cdot \Delta...|seq1|], seq2[TP[i - 1] + 1...|seq2|]) & \text{if } i = |TP| - 1 \\ \text{cigar}(seq1[i \cdot \Delta...(i + 1) \cdot \Delta], seq2[TP[i - 1] + 1]...TP[i] + 1) & \text{else.} \end{cases}$$

5:   end for
6:   cig  $\leftarrow$  combine(cig)
7:   return cig
8: end function
9:
10: function combine(cigar)
11:   cig  $\leftarrow$  empty String
12:   tmp  $\leftarrow 0$ 
13:   for each (cig_count, cig_symbol) in cigar do
14:     tmp  $\leftarrow tmp + previous\_cig\_count$ 
15:     if cig_symbol = previous_cig_symbol then
16:       if not last element in cigar then
17:         tmp  $\leftarrow 0$ 
18:       end if
19:     end if
20:     if last element in cigar then
21:       append (tmp + cig_count, cig_symbol) to cig
22:     end if
23:   end for
24:   return cig
25: end function

```

6 Fazit

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

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Hamburg, den _____ Unterschrift: _____