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 Kodierung **5**
2.1 Naive Binäre Kodierung 5
2.2 Unäre Kodierung 5
2.3 Huffman-Kodierung 5

3 Methoden **9**
3.1 CIGAR-Strings 9
3.1.1 Kodierung eines CIGAR-Strings 9
3.2 Trace Point Konzept 13
3.2.1 Differenzen-Kodierung 15
3.3 Entropie der Methoden 17

4 Resultate **19**
4.1 CIGAR Kodierung 19
4.2 Differenzen Kodierung 20
4.3 Entropie der Repräsentationen 21

5 Diskussion **25**
5.1 Bewertung CIGAR-Kodierung 25
5.2 Bewertung Kodierung der Differenzen der Trace Points 25
5.3 Bewertung Entropie beider Methoden 25

6 Programm **27**
6.1 Aufbau 27
6.2 Funktionalität 28
6.2.1 Informationsverlust bei der encode()-Funktion 28

7 Fazit **31**

Literaturverzeichnis **33**

Abbildungsverzeichnis

3.1 Huffman-Bäume der Kodierung des CIGAR-Strings 11
3.2 Huffman-Baum der Differenzen-Kodierung 17
3.3 Anzahl der Bits für die Kodierung des Beispiel-Alignments 18

4.1 Häufigkeitsverteilung der Kodierungen für 1 000 CIGAR-Strings von DNA Sequenzen mit je 5 000 Basenpaaren und einer Fehlerrate von 15%. 19
4.2 Anzahl der Bits für die Kodierungen der CIGAR-Strings 20
4.3 Entropie des CIGAR-Strings 21
4.4 Entropie der Trace Point Differenzen 22
4.5 Anzahl der Bits für die Kodierungen und Entropie 23

6.1 UML-Diagramm 27

Tabellenverzeichnis

3.1 Unäre Kodierung des CIGAR-Strings 11
3.2 Huffmann-Kodierung des CIGAR-Strings 12
3.3 Unäre Kodierung der Differenzen-Kodierung 16
3.4 Huffman-Kodierung der Differenzen-Kodierung 17

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 a, 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.

Üblicherweise wird ein Alignment in drei Zeilen wie folgt geschrieben. In der ersten Zeile schreibt man die Sequenz u und in der dritten Zeile die Sequenz v. In der mittleren Zeile symbolisiert das Zeichen '|' einen Match. Außerdem wird ein ε aus der Edit-Operation durch das Zeichen '$-$' dargestellt.

Beispiel 1. Sei von u und v das folgende Alignment $A = (a \rightarrow a, c \rightarrow c, t \rightarrow t, \varepsilon \rightarrow a, g \rightarrow g, a \rightarrow a, a \rightarrow a, c \rightarrow \varepsilon, t \rightarrow t)$ gegeben.

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

Die Edit-Distanz

Sei eine Kostenfunktion δ mit $\delta(a \rightarrow b) \geqslant 0$ 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

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

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 a, S. 19-21].

2 Kodierung

Das effiziente Speichern der Alignments wird durch die Kodierung der Daten maßgebend beeinflusst. Drei der gängigsten Kodierungsverfahren, welche in diesem Kapitel beschrieben werden, sind die naive binäre Kodierung, die unäre Kodierung und die Huffman-Kodierung.

2.1 Naive Binäre Kodierung

Bei einer naiven binären Kodierung wird jedes Symbol a aus dem Alphabet der zu kodierenden Symbole $\mathcal{A}$ mit $\lceil \log_2 |\mathcal{A}| \rceil$ Bit kodiert, was den Vorteil hat, dass alle Codewörter die gleiche Länge haben.

2.2 Unäre Kodierung

Die unäre Kodierung kodiert jedes Symbol nach der Häufigkeit des Auftretens im Alphabet mit $i-1$ $'0'$-Bits, gefolgt von einem $'1'$-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 $'1'$, das zweithäufigste mit $'01'$, das dritthäufigste mit $'001'$ usw. kodiert [Moffat u. Turpin 2002, S. 29-30]. Diese Art der Kodierung bietet sich insbesondere dann an, wenn ein zu kodierendes Symbol deutlich häufiger auftritt, als die anderen Symbole. Außerdem hat jedes Codewort die Größe 1, was den Vorteil hat, dass lediglich die Länge variiert.

2.3 Huffman-Kodierung

Bei einer *minimalen* binären Kodierung, wie sie auch im Huffman-Alogrithmus verwendet wird, werden die Längen der Codewörter anhand der Häufigkeiten der Symbole in der zu kodierenden Sequenz angepasst. Hierbei werden häufig vorkommende Symbole kurzen Codewörtern zugeordnet und weniger häufige

Symbole längeren Codewörtern. Somit lässt sich eine Kodierung ermöglichen, welche im Durchschnitt weniger Bit pro Symbol beansprucht. Jedes Codewort ist dabei nie der Anfang eines anderen Codewortes, was den Code präfixfrei macht und die Codewörter somit eindeutig zugeordnet werden können [Moffat u. Turpin 2002, S. 53-57].

Als Datenstruktur wird hierbei eine priorisierte Queue verwendet, welche nach den Häufigkeiten der Symbole sortiert ist. Der Algorithmus wählt immer die zwei Symbole mit den geringsten Häufigkeiten aus und fügt sie als ein Symbol zusammen, bis am Ende alle Symbole zusammengefügt wurden. So wird rekursiv ein Baum aufgebaut, welcher am Ende durch das Hinzufügen von den Kantengewichten 0 und 1 die Kodierungen der einzelnen Symbole bestimmt. Der Pseudocode des Algorithmus ist in Algorithmus 1 dargestellt [Kurtz b].

Immer wenn die Wahl zwischen mehreren Symbolen mit der selben Häufigkeit getroffen werden muss, kann eine Regel eingeführt werden. Beispielsweise können immer die zwei Symbole, die im Alphabet oder numerisch vor den anderen stehen, ausgewählt werden. Hierdurch wird die Kodierungen eindeutig. Diese Art der Kodierung nennt man eine 'kanonischen' Kodierung.

Der 'kanonische' Huffman-Algorithmus benötigt aufgrund der oben genannten Eigenschaften lediglich die Anzahl der Codewörter für jede vorhandene Codewortlänge, sowie die sortierten Symbole benötigt, um die Informationen zu dekodieren [Matai u. a. 2014].

Algorithmus 1 Pseudocode des Huffman-Algorithmus

Parameter: $\mathcal{A}$ ist das Alphabet der zu kodierenden Symbole mit den Häufigkeiten $p(a)$ für alle $a \in \mathcal{A}$.

```
1:  function Huffman(A, p(a))
2:      assert(| A | > 0)
3:      assert(p(a) > 0)
4:      Q ← ∅ als leere priorisierte Queue
5:      for all a ∈A do
6:          erzeuge neuen Knoten z
7:          (z.links, z.rechts, z.char, z.count) = (nil, nil, a, p(a))
8:          füge z in Q ein
9:      end for
10:     while |Q| ⩾ 2 do
11:         x ← extractmin(Q)
12:         y ← extractmin(Q)
13:         erzeuge neuen Knoten z
14:         (z.links, z.rechts ← (y, x)
15:         z.char ← x.char < y.char ? y.char : x.char
16:         z.count ← x.count + y.count
17:         füge z in Q ein
18:     end while
19:     root ← z
20: end function
```

3 Methoden

3.1 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.

Definition 1. Eine Folge von abwechselnd Quantitäten und Edit-Operationen $C = s_1c_1s_2c_2...s_nc_n$ ist ein CIGAR-String mit der Länge $|C| = 2n$ eines Alignments für alle Symbole $s \in$ {M,I,D} und alle Quantitäten $c \in \mathbb{N}$.

Beispiel 2. Sei folgendes Alignment aus Beispiel 1 gegeben.

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

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

3.1.1 Kodierung eines CIGAR-Strings

Für die Kodierung eines CIGAR-Strings ist es sinnvoll, alle Quantitäten und Symbole seperat zu kodieren, um so mit kleineren Alphabeten für die zu kodierenden Symbole arbeiten zu können.

$\mathcal{A}_s$ = {M,I,D} und $\mathcal{A}_c = \{s_i \mid 1 \leqslant i \leqslant n\}$ sind somit die Alphabete der zu kodierenden Symbole eines CIGAR-Strings.

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

Beispiel 3. 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.

Sei außerdem 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 Häufigkeiten jeden Symbols gegeben.

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

Die unäre Kodierung des oben genannte CIGAR-Strings ist in Tabelle 3.1 beschrieben. Sie benötigt folglich $32 + 35 = 67$ Bit.

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 3.2 beschrieben kodieren. Die Huffman-Bäume beider Alphabete sind in Abbildung 3.1 dargestellt. Für die Dekodierung sind somit zusätzlich die Listen $(1, 2)$, (M,D,I) und $(1, 1, 0, 4), (1, 4, 2, 5, 7, 8)$ zu speichern. Diese können als sogenannter 'Header' am Anfang der kodierten Datei unär kodiert werden. In diesem Fall würde der Header als $01001; 0001$ und $0101100001; 0000001$ 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 $28 + 33 + 9 + 17 = 87$ Bit.

Symbol	Häufigkeit	Kodierung	Anzahl Bits
M	10	1	$10 \cdot 1 = 10$
D	5	01	$5 \cdot 2 = 10$
I	4	001	$4 \cdot 3 = 12$
Gesamtanzahl:			32

Symbol	Häufigkeit	Kodierung	Anzahl Bits
1	13	1	$13 \cdot 1 = 13$
4	2	01	$2 \cdot 2 = 4$
2	1	001	$1 \cdot 3 = 3$
5	1	0001	$1 \cdot 4 = 4$
7	1	00001	$1 \cdot 5 = 5$
8	1	000001	$1 \cdot 6 = 6$
Gesamtanzahl:			35

Tabelle 3.1: Unäre Kodierung des CIGAR-Strings

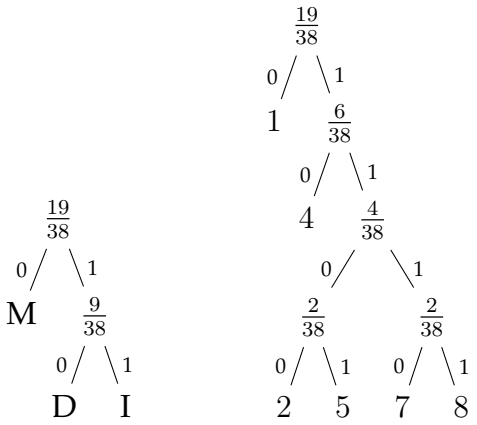

Abbildung 3.1: Huffman-Bäume der Kodierung des CIGAR-Strings

Symbol	Häufigkeit	Kodierung	Anzahl Bits
M	10	0	$10 \cdot 1 = 10$
D	5	10	$5 \cdot 2 = 10$
I	4	11	$4 \cdot 2 = 8$
Gesamtanzahl:			28

Symbol	Häufigkeit	Kodierung	Anzahl Bits
1	13	0	$13 \cdot 1 = 13$
4	2	10	$2 \cdot 2 = 4$
2	1	1100	$1 \cdot 4 = 4$
5	1	1101	$1 \cdot 4 = 4$
7	1	1110	$1 \cdot 4 = 4$
8	1	1111	$1 \cdot 4 = 4$
Gesamtanzahl:			33

Tabelle 3.2: Huffmann-Kodierung des CIGAR-Strings

3.2 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.

Der Grundgedanke dieser Methode ist es, das Alignment zweier gegebener Sequenzen nicht als solches kodiert abzuspeichern, sondern stattdessen die erste Sequenz in gleich große Abschnitte zu unterteilen und die Endpunkte, sogenannte Trace Points, von Teilabschnitten des Alignments in der zweiten Sequenz abzuspeichern. So kann für jeden Teilabschnitt jeweils ein Teilalignment berechnet werden. Für die Unterteilung der ersten Sequenz wird der positive Parameter Δ verwendet. Die Teilalignments können dann zu einem Gesamtalignment konkateniert werden.

Das Verfahren bietet den Vorteil, durch die Größe der Teilabschnitte die Anzahl der Trace Points und somit den Speicherbedarf, sowie die Zeit, die für die Berechnung der Teilalignments benötigt wird, anpassen zu können.

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 außerdem für die dynamische Anpassung der Intervallgrenzen der Parameter p gegeben mit

$$p = \begin{cases} \lceil \frac{i}{\Delta} \rceil & \text{falls } i > 0 \\ 1 & \text{falls } i = 0. \end{cases}$$

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 $0 \leqslant q < \tau - 1$ 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 $\delta(A') \leqslant \delta(A)$ konstruiert werden.

Danach bestimmt man aus den Trace-Points die Substring-Paare u_q und v_q, berechnet hierfür jeweils ein optimales Alignment und konkateniert die Alignments von den aufeinanderfolgenden Substring-Paaren zu A'.

Beispiel 4. Sei $\Delta = 15$ und folgendes Alignment gegeben:

```
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
```

Beide Sequenzen seien in dem Intervall $0...37$. Das Alignment wird wie oben beschrieben in $\tau = \left\lceil \frac{37}{15} \right\rceil - \left\lfloor \frac{0}{15} \right\rfloor = 3 - 0 = 3$ Abschnitte unterteilt. Der erste Abschnitt ist in u von 0 bis 14 und in v von 0 bis 15.

```
gagc-a-t-gttgcc-tgg
||  |  |  |  |   |  |  |||
gaccaagtag--g-cgtgg
```

Der zweite Abschnitt ist in u von 15 bis 29 und in v von 16 bis 28.

```
tcctttgctaggtac
||||  |||  |||  |
acctt-gctcggt-c
```

Der dritte Abschnitt ist in u von 30 bis 37 und in v von 29 bis 37.

```
tgta-gaga
||||  ||||
tgtaagaga
```

Somit ergeben sich als Endpunkte aller Abschnitte außer dem letzten Abschnitt die Trace Points 15 und 28.

3.2.1 Differenzen-Kodierung

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

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 5. 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 3.1.1 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 in diesem Beispiel somit eine Differenzen-Kodierung möglich.

Als neue Liste zu kodierender Werte ergibt sich nach 3.2.1 $L_{diff} = (5, 5, 5, 5, 4, 4, 6)$.

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.

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

Tabelle 3.3: Unäre Kodierung der Differenzen-Kodierung

Für das oben genannten Beispiel ergibt sich somit

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

sowie das Alphabet $\mathcal{A} = \{4, 5, 6\}$.

Bei der naiven binären Kodierung von L_{diff} ergibt sich analog zu 3.1.1 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 3.3 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.

Die Ausführung des Huffman-Algorithmus kodiert nach [Moffat u. Turpin 2002, S. 54] die Symbole wie in Tabelle 3.4 aufgelistet. Der dazugehörige Huffmann-Baum aus 3.2 verdeutlicht die Kodierung der einzelnen Symbole, muss aber für den kanonischen Huffman-Algorithmus, wie in 3.1.1 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 $01001; 0001$ 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.

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 3.4: Huffman-Kodierung der Differenzen-Kodierung

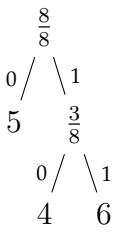

Abbildung 3.2: Huffman-Baum der Differenzen-Kodierung

3.3 Entropie der Methoden

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

$$H(S) = -\sum_{a \in S} p(a) \cdot \log_2 p(a)$$

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

Für den in 3.1.1 genannten CIGAR-String ergibt sich eine Entropie von

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

und somit eine Gesamtentropie von $2.54 \cdot 38 = 96.52$ Bit.

Die in 3.2.1 genannten Trace Point Differenzen des selben Alignments ergeben analog

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

und somit eine Gesamtentropie von $1.30 \cdot 8 = 10.4$ Bit.

Zusammenfassend ergibt sich für das Alignment aus Beispiel 1 die folgende Tabelle:

	CIGAR-String	Trace Point Differenzen
Binäre Kodierung	95	24
Unäre Kodierung	67	12
Huffman-Kodierung	87	21
Entropie	96.52	10.4

Abbildung 3.3: Anzahl der Bits für die Kodierung des Beispiel-Alignments

4 Resultate

4.1 CIGAR Kodierung

Für die Kodierung der CIGAR-Strings wird zunächst eine zufällig angeordnete DNA-Sequenz und daraus mit der jeweiligen Fehlerrate eine abgewandelte Sequenz berechnet. Dieser Vorgang wiederholt sich mehrfach. Aus einem Sequenzpaar wird dann ein Alignment berechnet, aus dem der CIGAR-String extrahiert wird. Dieser wird dann mit den oben beschriebenen Verfahren kodiert und die Anzahl der Bits, die für die jeweilige Kodierung benötigt werden, werden berechnet und deren Verteilung in Abbildung 4.1 grafisch dargestellt. Die gemessenen Werte sind in Tabelle 4.2 verdeutlicht.

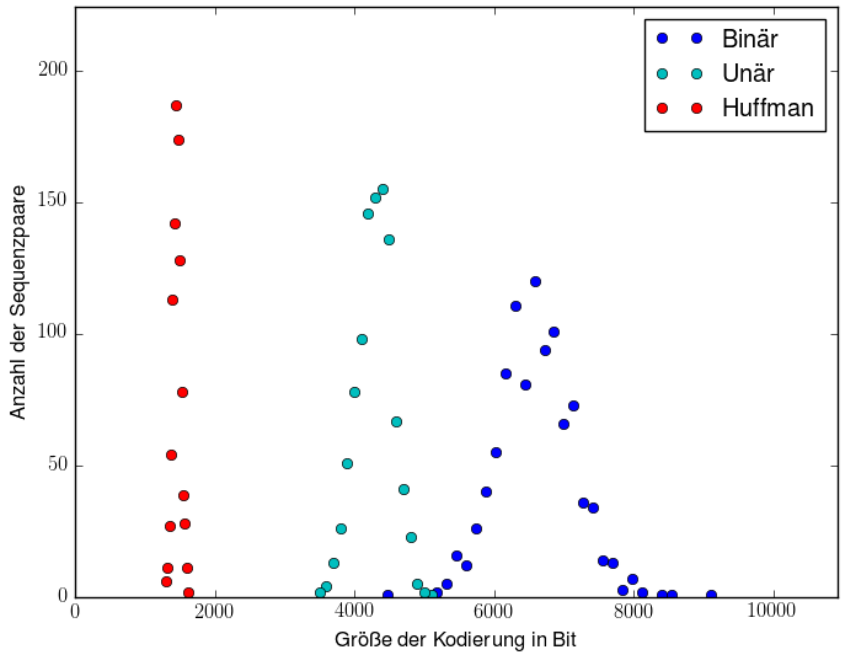

Abbildung 4.1: Häufigkeitsverteilung der Kodierungen für 1 000 CIGAR-Strings von DNA Sequenzen mit je 5 000 Basenpaaren und einer Fehlerrate von 15%.

	Minimum	Maximum	∅
Binäre Kodierung	4 480	9 100	6 595,50
Unäre Kodierung	3 500	5 100	4 290,94
Huffman-Kodierung	1 300	1 625	1 458,02

Abbildung 4.2: Anzahl der Bits für die Kodierungen der CIGAR-Strings

Die naive binäre Kodierung der CIGAR-Strings benötigt, wie in Abbildung 4.2 dargestellt, für 1 000 Sequenzpaare mit je 5 000 Basen im Mittel 6 595,50 Bit, wobei Werte zwischen 4 480 und 9 100 Bit erreicht werden. Diese bilden ein Maximum bei 120 Sequenzpaaren mit jeweils etwa 6 500 Bit.

Für die unäre Kodierung der selben CIGAR-Strings werden Werte zwischen 3 500 und 5 100 Bit erreicht, wobei der Durchschnitt bei 4 290,94 und damit etwa 35% unter dem der naiven binären Kodierung liegt. Das Maximum liegt hier bei 155 Sequenzpaaren, deren CIGAR-Strings mit je etwa 4200 Bit kodiert werden.

Die Huffman-Kodierung kodiert die CIGAR-Strings zwischen 1 300 und 1 625 Bit, wobei das Maximum von 180 Sequenzpaaren mit 1 500 Bit erreicht wird. Der Mittelwert liegt hier bei 1 458,02 Bit und damit etwa 78% unter dem der naiven binären Kodierung und etwa 66% unter dem der unären Kodierung.

4.2 Differenzen Kodierung

Für die Kodierung der Trace Point Differenzen wird zunächst eine zufällig angeordnete DNA-Sequenz und daraus mit der jeweiligen Fehlerrate eine abgewandelte Sequenz berechnet. Dieser Vorgang wiederholt sich mehrfach. Aus einem Sequenzpaar wird dann ein Alignment berechnet, aus dem der CIGAR-String extrahiert wird. Aus diesem werden die Trace Points anhand des Δ-Wertes ausgelesen und die Differenzen der Trace Points bestimmt. Diese werden dann mit den oben beschriebenen Verfahren kodiert und die Anzahl der Bits, die für die jeweilige Kodierung benötigt werden, werden berechnet und deren Verteilung grafisch dargestellt.

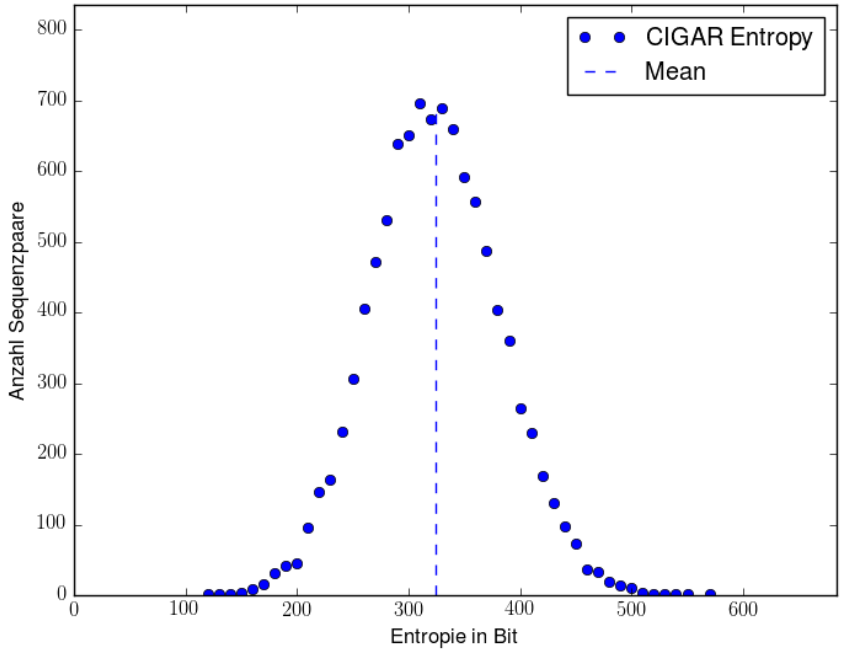

Abbildung 4.3: Entropie des CIGAR-Strings

4.3 Entropie der Repräsentationen

Die folgenden Grafiken zeigen 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 4.3 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.

Die Differenzen der Trace Points weisen hingegen, wie in Abbildung 4.4 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.

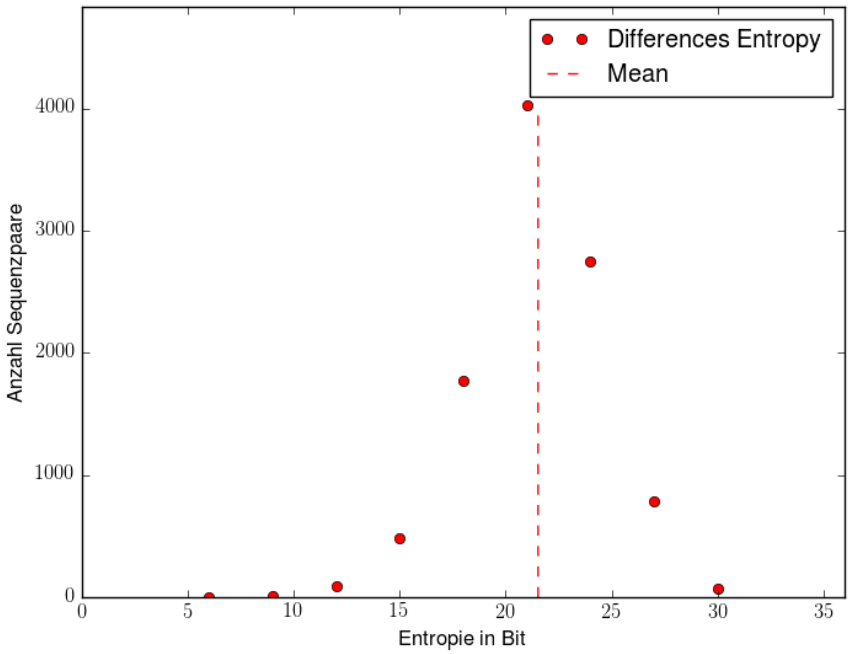

Abbildung 4.4: Entropie der Trace Point Differenzen

Zusammenfassend ergibt sich für die Resultate folgende Tabelle:

	CIGAR-String			Trace Point Differenzen		
	Min	Max	∅	Min	Max	∅
Binäre Kodierung						
Unäre Kodierung						
Huffman-Kodierung						
Entropie						

Abbildung 4.5: Anzahl der Bits für die Kodierungen und Entropie

5 Diskussion

TODO

5.1 Bewertung CIGAR-Kodierung

5.2 Bewertung Kodierung der Differenzen der Trace Points

5.3 Bewertung Entropie beider Methoden

6 Programm

6.1 Aufbau

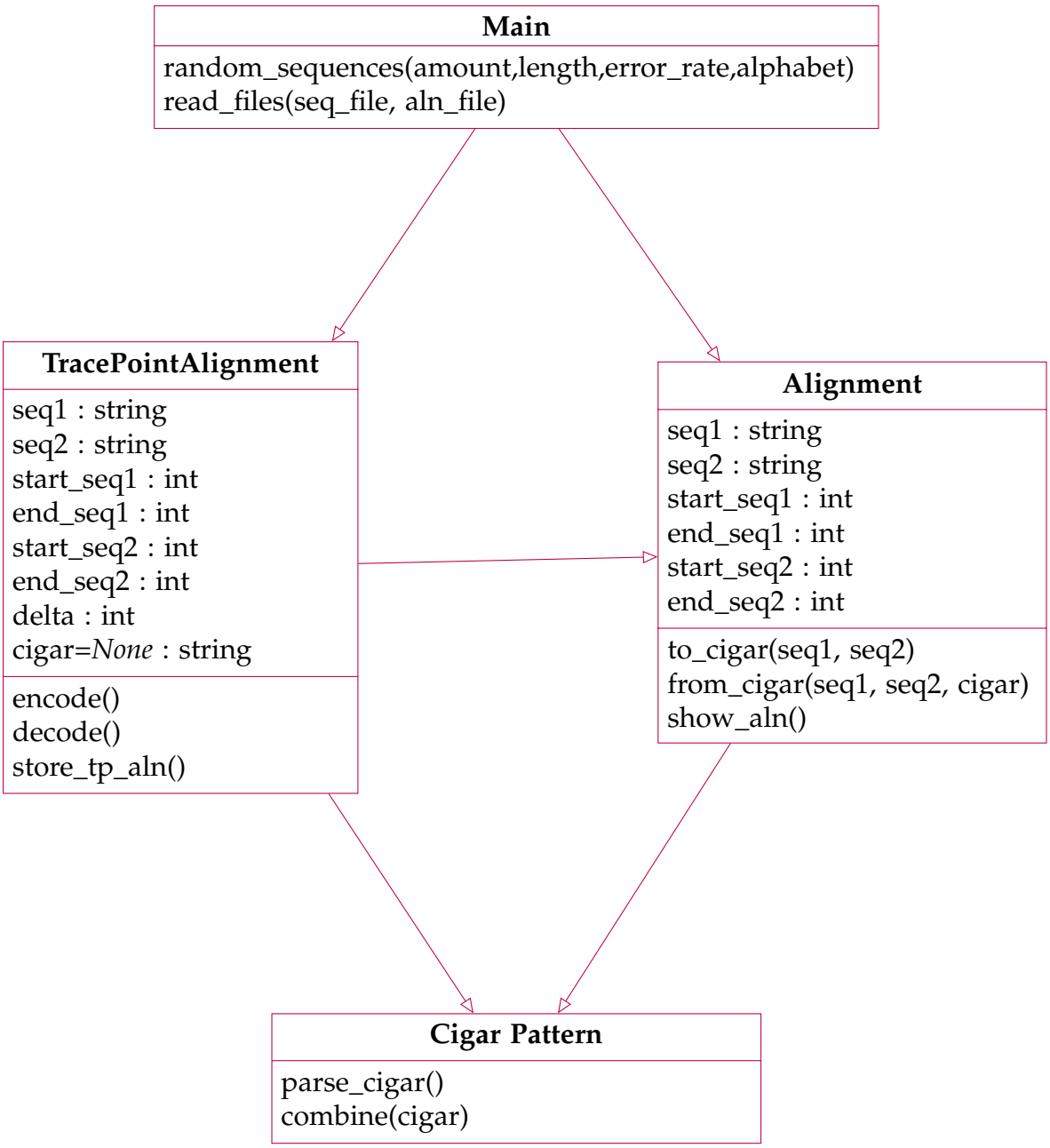

Abbildung 6.1: UML-Diagramm

6.2 Funktionalität

6.2.1 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. Um bestimmen zu können, wie die einzelnen Abschnitte zueinander aligniert werden, muss in der decode()-Funktion zunächst ein neues Alignment des jeweiligen Intervall-Paares errechnet werden und alle Teilalignments zu einem Gesamtalignment konkateniert werden. Hierbei ist jedoch nicht gewährleistet, dass das neue Alignment dem alten entspricht. Da das neue Alignment aus konkatenierten optimalen Alignments besteht, ist es ebenfalls optimal und hat daher mindestens die gleiche Edit-Distanz wie das alte Alignment.

Algorithmus 2 Berechnung der Trace Points von einem gegebenen CIGAR-String

Parameter: $seq1$ und $seq2$ sind die beiden Sequenzen mit den jeweiligen Start- und Endpositionen $start\_seq1, end\_seq1$ bzw. $start\_seq2, end\_seq2$. Zusätzlich werden der Funktion der Δ-Wert und der CIGAR-String übergeben.

1: **function** *encode*($seq1, seq2, start\_seq1, end\_seq1, start\_seq2, \Delta, cigar$)
2: assert($|seq1|, |seq2|, |cigar|, \Delta > 0$)
3: assert($start\_seq1, start\_seq2 \geqslant 0$)
4: assert($start\_seq1 < end\_seq1$)
5: $p \leftarrow MAX(1, \lceil start\_seq1/\Delta \rceil)$
6: $\tau \leftarrow \lceil end\_seq1/\Delta \rceil - \lfloor start\_seq2/\Delta \rfloor$
7: $uTP \leftarrow$ Array for interval termini in the first sequence
8: **for** $i \leftarrow 0$ **upto** $|\tau|$ **do**
9: $uTP[i] \leftarrow (p + i) \cdot (\Delta - 1)$
10: **end for**
11: $uChars, vChars, count \leftarrow 0$
12: $TP \leftarrow$ Array for Trace Points
13: **for** each (cig_count, cig_symbol) in cigar **do**
14: **for** $i \leftarrow 0$ **upto** cig_count **do**
15: **if** cig_symbol = 'I' **then**
16: increment $uChars$
17: **else if** cig_symbol = 'D' **then**
18: increment $vChars$
19: **else**
20: increment $uChars, vChars$
21: **end if**
22: **if** $uChars = uTP[count]$ **then**
23: TP.append($vChars$)
24: **end if**
25: **if** $count \neq |uTP| - 1$ **then**
26: **return** TP
27: **else**
28: increment $count$
29: **end if**
30: **end for**
31: **end for**
32: **end function**

Algorithmus 3 Berechnung eines CIGAR-Strings von einem gegebenen Trace Point Array

Parameter: $seq1$ und $seq2$ sind die beiden Sequenzen.
Zusätzlich werden der Funktion der Δ-Wert und das Trace Point Array übergeben.

```
1: function decode(seq1, seq2, Δ, TP)
2:     assert(|seq1|, |seq2|, Δ, |TP| > 0)
3:     cig ← empty String
4:     for i ← 0 upto |TP| do
5:         if i = 0 then
6:             cig.append(cigar(seq1[0...Δ], seq2[0...TP[i] + 1]))
7:         else if i = |TP| - 1 then
8:             cig.append(cigar(seq1[i · Δ...|seq1|], seq2[TP[i − 1] + 1...|seq2|]))
9:         else
10:            cig.append(cigar(seq1[i·Δ...(i+1)·Δ], seq2[TP[i−1]+1]...TP[i]+
   1))
11:        end if
12:    end for
13:    cig ← combine(cig)
14:    return cig
15: end function
16:
17: function combine(cigar)
18:    cig ← empty String
19:    tmp ← 0
20:    for each cig_count, cig_symbol in cigar do
21:        tmp ← tmp + previous_cig_count
22:        if cig_symbol = previous_cig_symbol then
23:            if not last element in cigar then
24:                tmp ← 0
25:            end if
26:        end if
27:        if last element is in cigar then
28:            cig.append(tmp + cig_count, cig_symbol)
29:        end if
30:    end for
31:    return cig
32: end function
```

7 Fazit

TODO

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

[Kurtz b] KURTZ, Stefan: *Project Description for Projekt ProgrammierungfÃ¼r Naturwissenschaften, Summer 2014*

[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