



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:  
Zweitgutachter Universität Hamburg:

Prof. Dr. Stefan Kurtz  
Dr. Giorgio Gonnella



---

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Kodierung</b>	<b>5</b>
2.1	Naive Binäre Kodierung . . . . .	5
2.2	Unäre Kodierung . . . . .	5
2.3	Huffman-Kodierung . . . . .	5
<b>3</b>	<b>Methoden</b>	<b>9</b>
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
<b>4</b>	<b>Resultate</b>	<b>19</b>
4.1	CIGAR Kodierung . . . . .	19
4.2	Differenzen Kodierung . . . . .	21
4.3	Entropie der Repräsentationen . . . . .	25
<b>5</b>	<b>Diskussion</b>	<b>27</b>
5.1	CIGAR-Kodierung . . . . .	27
5.2	Kodierung der Differenzen der Trace Points . . . . .	28
5.3	Entropie beider Methoden . . . . .	29
<b>6</b>	<b>Programm</b>	<b>31</b>
6.1	Funktionalität . . . . .	32
<b>7</b>	<b>Fazit</b>	<b>35</b>
	<b>Literaturverzeichnis</b>	<b>37</b>

---



---

# 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%. . . . .	20
4.2	Anzahl der Bits für die Kodierungen der CIGAR-Strings . . . . .	20
4.3	Häufigkeitsverteilung der Kodierungen der Trace Point Differen- zen für 1 000 DNA-Sequenzen mit je 5 000 Basenpaaren, einer Feh- lerrate von 15% und einem $\Delta$ -Wert von 200. . . . .	22
4.4	Häufigkeitsverteilung der Huffman-Kodierungen der Trace Point Differenzen für 1 000 DNA-Sequenzen mit je 5 000 Basenpaaren, einer Fehlerrate von 15% und $\Delta$ -Werten von 50, 100, 200 und 500. .	23
4.5	Häufigkeitsverteilung der Huffman-Kodierungen der Trace Point Differenzen für 1 000 DNA-Sequenzen mit je 5 000 Basenpaaren, einem $\Delta$ -Wert von 100 und Fehlerraten von 5%, 15%, 30% und 50%. .	24
4.6	Entropie von 1 000 CIGAR-Strings von DNA-Sequenzen mit je 5 000 Basenpaaren, einem $\Delta$ -Wert von 100 und einer Fehlerrate von 15% . . . . .	25
4.7	Anzahl der Bits für die Kodierungen und Entropie . . . . .	26
6.1	UML-Diagramm . . . . .	31

---



# 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 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.

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  $\varepsilon$  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  $(\alpha, \beta)$  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.

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  $\varepsilon$  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  $\delta$  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  $\delta$  die Einheitskostenfunktion ist, so ist  $\text{edist}_\delta(u, v)$  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-Algorithmus 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( $\mathcal{A}, p(a)$ )
2:   assert( $|\mathcal{A}| > 0$ )
3:   assert( $p(a) > 0$ )
4:    $Q \leftarrow \emptyset$  als leere priorisierte Queue
5:   for all  $a \in \mathcal{A}$  do
6:     erzeuge neuen Knoten  $z$ 
7:      $(z.links, z.rechts, z.char, z.count) = (\text{nil}, \text{nil}, a, p(a))$ 
8:     füge  $z$  in  $Q$  ein
9:   end for
10:  while  $|Q| \geq 2$  do
11:     $x \leftarrow \text{extractmin}(Q)$ 
12:     $y \leftarrow \text{extractmin}(Q)$ 
13:    erzeuge neuen Knoten  $z$ 
14:     $(z.links, z.rechts \leftarrow (y, x)$ 
15:     $z.char \leftarrow x.char < y.char ? y.char : x.char$ 
16:     $z.count \leftarrow x.count + y.count$ 
17:    füge  $z$  in  $Q$  ein
18:  end while
19:   $root \leftarrow z$ 
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  $C = s_1c_1s_2c_2\dots s_nc_n$  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 durch 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 separat zu kodieren, um so mit kleineren Alphabeten für die zu kodierenden Symbole arbeiten zu können.

$\mathcal{A}_s = \{M,I,D\}$  und  $\mathcal{A}_c = \{s_i \mid 1 \leq i \leq n\}$  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-tgggtcctttgctaggtactgta-gaga
| | | | | | | | | | | | | | | | | |
gaccaagtag--g-cgtggacctt-gctcggg-ctgtaagaga
0      5      0      5      0      5      0      5      0

```

gegeben, welches durch den CIGAR-String  $4M1I1M1I1M1I1M2D1M1D1M1I8M1D7M1D5M1I4M$  repräsentiert werden kann.

Sei außerdem das Alphabet  $\mathcal{A}_s = \{M, I, D\}$ , 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  $0101100001; 00000001$  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

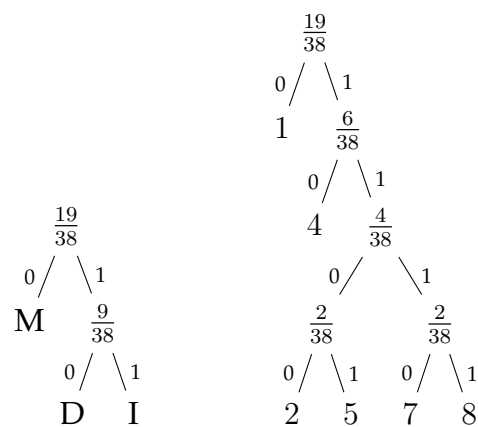


Abbildung 3.1: Huffman-Bäume der Kodierung des CIGAR-Strings

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: Huffman-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  $\Delta$  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  $v[k...l]$  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  $u_0, u_1, \dots, u_{\tau-1}$  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  $i, j, k, l, \Delta$  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 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-gctcggc-ctgtaagaga
0      5      0      5      0      5      0      5      0

```

Beide Sequenzen seien in dem Intervall von 0 bis 37. Das Alignment wird wie oben beschrieben in  $\tau = \lceil \frac{37}{15} \rceil - \lfloor \frac{0}{15} \rfloor = 3 - 0 = 3$  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-gctcggc-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  $L = (a_1, a_2, \dots, a_n)$  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.

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

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-gctcggc-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  $\Delta$ -Wert, damit die Grenzen der Substrings beider Sequenzen berechnet werden können. Hierfür muss also der  $\Delta$ -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

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

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 Huffman-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  $11 + 9 = 20$  Bit und benötigt damit nur  $\frac{20}{70} \approx 28\%$  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

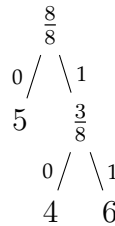


Abbildung 3.2: Huffman-Baum der Differenzen-Kodierung

### 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

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

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



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

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

Um von den Kodierungen des Alignments aus Beispiel 3 auf die Verteilung einer großen Anzahl von Alignments schließen zu können, habe ich die verschiedenen Kodierungen für jeweils CIGAR-Strings und Trace Point Differenzen mit unterschiedlichen Parametern, sowie die Verteilung der Entropie der zwei Verfahren berechnet und grafisch dargestellt.

Die Resultate dieser Berechnungen werden in diesem Kapitel detailliert beschrieben.

### 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.

---

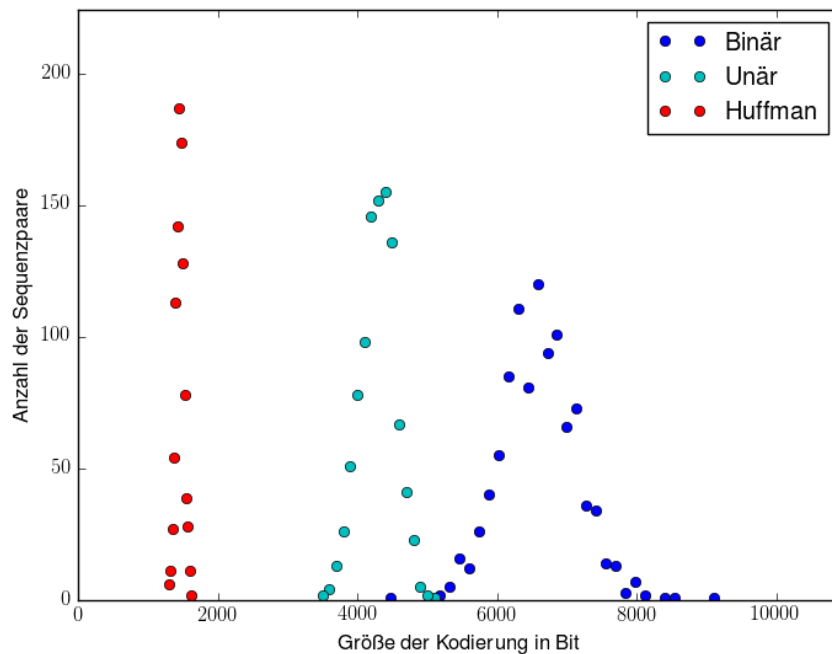


Abbildung 4.1: Häufigkeitsverteilung der Kodierungen für 1 000 CIGAR-Strings von DNA Sequenzen mit je 5 000 Basenpaaren und einer Fehler-rate von 15%.

	Minimum	Maximum	$\emptyset$
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 4 200 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 Mit-

telwert 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  $\Delta$ -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.

Für die binäre Kodierung der Trace Point Differenzen werden bei einem  $\Delta$ -Wert von 200 bei jedem Durchlauf für 24 Trace Points und den  $\Delta$ -Wert selbst insgesamt  $\lceil \log_2 25 \rceil = 125$  Bit benötigt.

Die unäre und Huffman-Kodierungen der Trace Point Differenzen sind Abbildung 4.3 dargestellt. Es ist deutlich zu erkennen, dass beide Verteilungen ähnlich verlaufen. Die unäre Kodierung benötigt zwischen 60 und 132 Bit und somit 91.19 Bit im Mittel. Bei der Huffman-Kodierung liegen die Werte zwischen 42 und 126 Bit bei einem Mittelwert von 84.18 Bit. Somit benötigt die Huffman-Kodierung etwa 33% weniger Speicher als die binäre Kodierung und etwa 8% weniger als die unäre Kodierung.

Um die Auswirkungen der Wahl des  $\Delta$ -Parameters messen zu können, wurden, wie oben beschrieben, die Häufigkeitsverteilung der Huffman-Kodierung der Trace Point Differenzen für jeweils 1 000 zufällige Sequenzpaare mit je 5 000 Basenpaaren, einer Fehlerrate von 15% und  $\Delta$ -Werten von 50, 100, 200 und 500 berechnet. Wie in Abbildung 4.4 zu erkennen ist, liegt die Größe der Kodierung für  $\Delta = 50$  zwischen 45 und 130 Bit bei einem Mittelwert von 86.8 Bit. Für  $\Delta = 100$  liegen die Werte zwischen 42 und 126 Bit und der Mittelwert bei 80.22 Bit. Weiterhin werden für  $\Delta = 200$  Werte zwischen 42 und 126 Bit erreicht, bei einem Mittelwert von 84.18 Bit. Abschließend erreicht die Kodierung für  $\Delta = 500$  Werte im Bereich von 25 und 90 Bit. Hier liegt der Mittelwert bei 57.95 Bit.

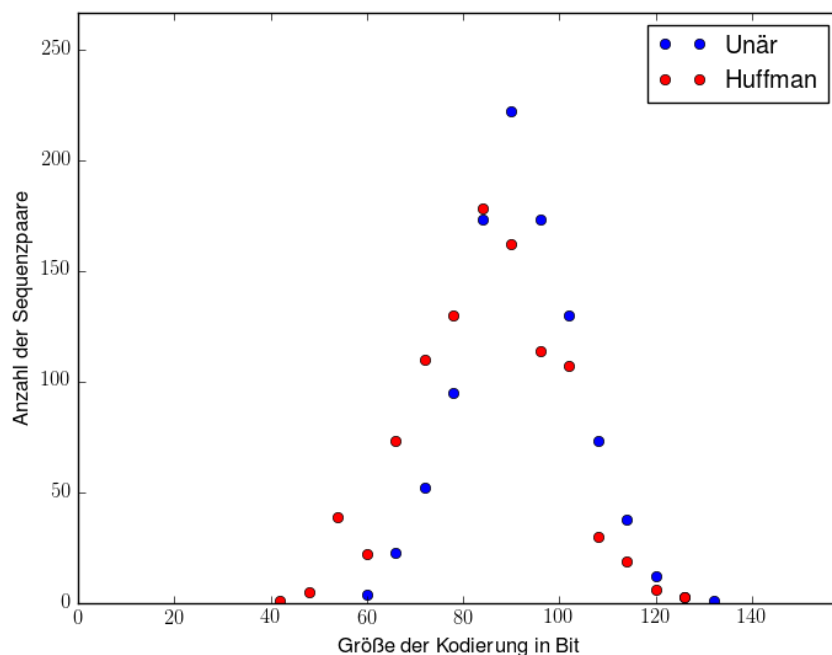


Abbildung 4.3: Häufigkeitsverteilung der Kodierungen der Trace Point Differenzen für 1 000 DNA-Sequenzen mit je 5 000 Basenpaaren, einer Fehlerrate von 15% und einem  $\Delta$ -Wert von 200.

Somit benötigt die Huffman-Kodierung bei einem groß gewählten  $\Delta$  von 500 etwa ein Drittel weniger Speicher als bei einem klein gewählten  $\Delta$ -Wert von 50.

Analog zu den verschiedenen  $\Delta$ -Werten wurde außerdem die Auswirkung der Fehlerrate gemessen. Hierfür wurde die Häufigkeitsverteilung der Huffman-Kodierung für 1 000 Sequenzpaare mit je 5 000 Basenpaaren, einem  $\Delta$ -Wert von 100 und Fehlerraten von 5%, 15%, 30% und 50% berechnet und in Abbildung 4.5 grafisch dargestellt.

Wie in der Abbildung zu erkennen ist, werden für die Huffman-Kodierung mit einer Fehlerrate von 5% 24 bis 96 Bit benötigt, wobei der Mittelwert bei 50.99 Bit liegt. Die Kodierung bei einer Fehlerrate von 15% benötigt zwischen 48 und 128 Bit bei einem Mittelwert von 83.03 und bei einer Fehlerrate von 30% werden 56 bis 168 Bit benötigt, wobei der Mittelwert bei 114.28 Bit liegt. Abschließend werden für eine Kodierung mit einer Fehlerrate von 50% zwischen 90 und 210 und im Mittel 151.88 Bit benötigt.

Bei einer geringen Fehlerrate von 5% werden somit 66.5% weniger Speicher benötigt, als bei einer hohen Fehlerrate von 50%.

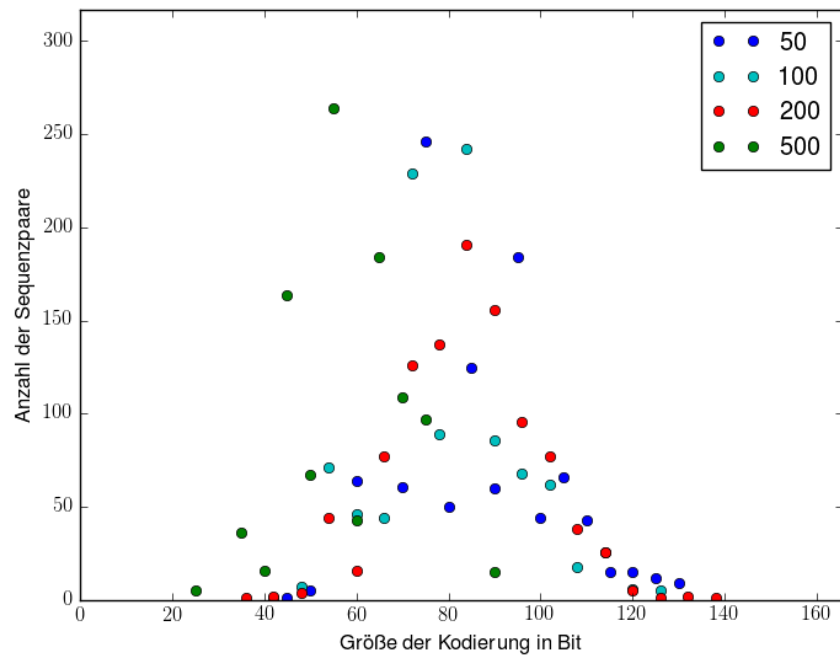


Abbildung 4.4: Häufigkeitsverteilung der Huffman-Kodierungen der Trace Point Differenzen für 1 000 DNA-Sequenzen mit je 5 000 Basenpaaren, einer Fehlerrate von 15% und  $\Delta$ -Werten von 50, 100, 200 und 500.

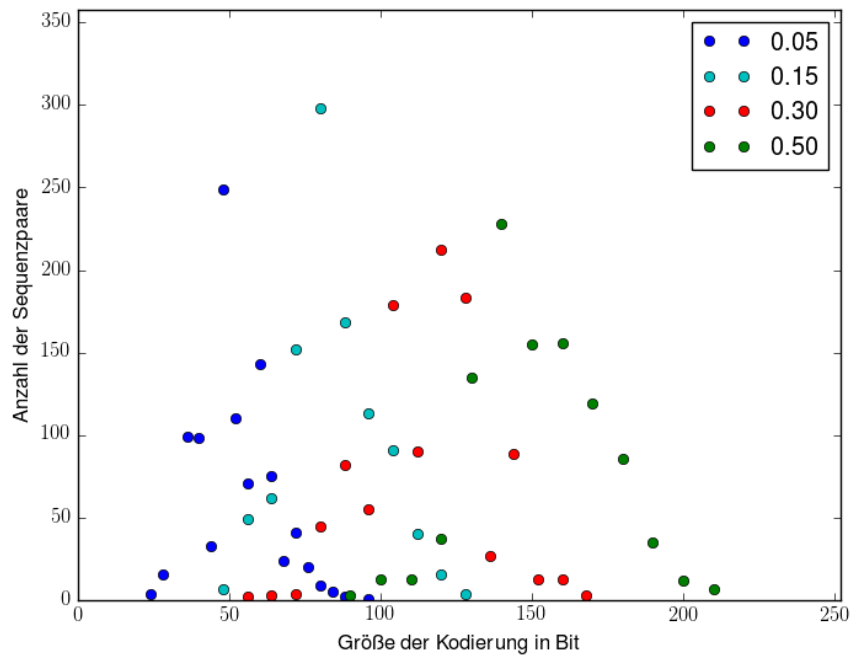


Abbildung 4.5: Häufigkeitsverteilung der Huffman-Kodierungen der Trace Point Differenzen für 1 000 DNA-Sequenzen mit je 5 000 Basenpaaren, einem  $\Delta$ -Wert von 100 und Fehlerraten von 5%, 15%, 30% und 50%.





	CIGAR-String			Trace Point Differenzen ( $\Delta = 200$ )		
	Min	Max	$\emptyset$	Min	Max	$\emptyset$
Binäre Kodierung	4 480	9 100	6 595.50	125	125	125
Unäre Kodierung	3 500	5 100	4 290.94	60	132	90.19
Huffman-Kodierung	1 300	1 625	1 458.02	42	126	84.18
Entropie	1 280	2 280	1 755.79	91	212	134.69

Abbildung 4.7: Anzahl der Bits für die Kodierungen und Entropie

## 5 Diskussion

In diesem Kapitel werden die Resultate aus Kapitel 4 interpretiert und die Qualität der Kodierungen in Bezug auf eine speichereffiziente Repräsentation eines Alignments betrachtet.

### 5.1 CIGAR-Kodierung

Die Kodierungen der CIGAR-Strings, wie sie in Abbildung 4.1 grafisch und in Tabelle 4.7 numerisch dargestellt sind, zeigen, dass die Huffman-Kodierung mit 1 458.02 Bit im Mittel deutlich weniger Speicher benötigt, als die unäre Kodierung mit 4 290.94 Bit oder die binäre Kodierung mit 6 595.50 Bit.

In CIGAR-Strings von Alignments mit einer geringen Fehlerrate ist die Anzahl der Matches deutlich höher als die der Insertions oder Deletions, was zu großen Quantitäten der 'M'-Operation und kleinen Quantitäten der 'I'- und 'D'-Operationen führt. Insertions und Deletions treffen dazu selten hintereinander auf, was einen Schwerpunkt der Quantität '1' zufolge hat. Für so eine Art der Verteilung ist die binäre Kodierung weniger gut geeignet als die unäre oder Huffman-Kodierung, da sie keine Speicherersparnis aus häufig auftretenden Symbolen ziehen kann, sondern alle Symbole mit der selben Anzahl an Bits kodiert.

Die unäre Kodierung benötigt bei kürzeren Sequenzen, wie bei denen aus Beispiel 3, weniger Speicher als die Huffman-Kodierung, da diese für die Dekodierung jedes Mal den 'Header' mitspeichern muss und sich dieser zusätzliche Speicherverbrauch bei kürzeren Sequenzen stärker auswirkt, als bei längeren. In den Resultaten wurden deutlich längere Sequenzen verwendet, weshalb hier die Huffman-Kodierung wie oben beschrieben nur etwa 34% des Speicherbedarfs der unären Kodierung benötigt.

Da kürzere Sequenzen hier nur für das Beispiel verwendet wurden und Alignments üblicherweise von längeren Sequenzabschnitten gebildet werden, ist somit die Huffman-Kodierung die speichereffizienteste der drei Verfahren.

---

## 5.2 Kodierung der Differenzen der Trace Points

Die Kodierung der Differenzen der Trace Points ist für ein  $\Delta = 200$  und eine Fehlerrate von 15% in Abbildung 4.3 grafisch dargestellt und in der Tabelle 4.7 numerisch beschrieben. In Abbildung 4.4 sind die Auswirkungen der Wahl des  $\Delta$ -Parameters dargestellt und in Abbildung 4.5 die Auswirkungen der Fehlerrate der Sequenzen.

Für einen  $\Delta$ -Wert von 200 benötigt die Huffman-Kodierung im Durchschnitt nur 84.18 Bit und damit etwa 9.3% weniger Speicher, als die unäre Kodierung mit 90.19 Bit. Die binäre Kodierung liegt mit durchschnittlich 125 Bit deutlich über dem Bedarf der anderen beiden Kodierungen. Da die einzelnen Abschnitte in der  $v$ -Sequenz zu  $\Delta$  großen Abschnitten der  $u$ -Sequenz aligniert werden, sind diese üblicherweise auch genauso oder ähnlich groß wie das  $\Delta$ . Das hat zur Folge, dass die Differenzen der Trace Points in einem ähnlichen Wertebereich liegen und die unäre und Huffman-Kodierung aus dieser Verteilung einen Vorteil ziehen können und daher speichereffizienter kodieren, wobei der Huffman-Algorithmus etwas besser abschneidet als die unäre Kodierung.

Die Wahl des  $\Delta$ -Wertes kann die Größe der Kodierung deutlich beeinflussen, da dieser Wert die Anzahl der zu speichernden Symbole bestimmt. Bei einem großen  $\Delta$  müssen weniger Symbole gespeichert werden und bei einem kleinen  $\Delta$  analog mehr. So benötigt die Huffman-Kodierung für ein  $\Delta = 500$  folglich nur etwa 66% des Speichers als für ein  $\Delta = 50$ .

Die Fehlerrate der Sequenzen hat ebenfalls einen großen Einfluss auf die Größe der Kodierung, da für sehr ähnliche Sequenzen mit einer hohen Anzahl an Matches die Trace Points ein Vielfaches von  $\Delta$  sind und somit die Differenzen der Trace Points ebenfalls  $\Delta$  groß sind und daher deutlich weniger unterschiedliche Symbole kodiert werden müssen. So benötigt die Huffman-Kodierung mit  $\Delta = 200$  für eine kleine Fehlerrate von 5% im Durchschnitt 50.99 Bit und damit im Verhältnis zu einer hohen Fehlerrate von 50%, welche im Schnitt 151.88 Bit benötigt, lediglich etwa ein Drittel des Speichers.

---

---

## 5.3 Entropie beider Methoden

Die Entropie der CIGAR-Strings und der Trace Point Differenzen sind in Abbildung 4.6 grafisch dargestellt und in der Tabelle 4.7 numerisch beschrieben.

Sie gibt den Informationsgehalt einer Sequenz in Relation zu der Verteilung der Symbole an und wird deshalb in Bit pro Symbol berechnet und dann später mit der Anzahl der zu kodierenden Symbole multipliziert. Da ein CIGAR-String alle Edit-Operationen eines Alignments beschreibt und die Trace Point Differenzen lediglich die Abstände von einem zu alignierenden Abschnitt zu dem nächsten beschreiben, müssen in der Regel für CIGAR-Strings mehr Symbole kodiert werden, als für die Trace Point Differenzen. Diese Eigenschaft spiegelt sich auch in der Entropie beider Verfahren wieder. Die CIGAR-Strings weisen eine Entropie von durchschnittlich 1 755.79 Bit auf, was etwa 13 mal so viel ist wie die Entropie der Trace Point Differenzen, welche bei durchschnittlich 134.69 Bit liegt.

---



## 6 Programm

Die Umsetzung des Trace Point Verfahrens, sowie die Berechnung der verschiedenen Größen der Kodierung ist im Rahmen dieser Arbeit zunächst als Python Implementierung und anschließend als C Implementierung realisiert worden [Wiese 2016]. Der Aufbau als UML-Klassendiagramm ist in Abbildung 6.1 zu sehen.

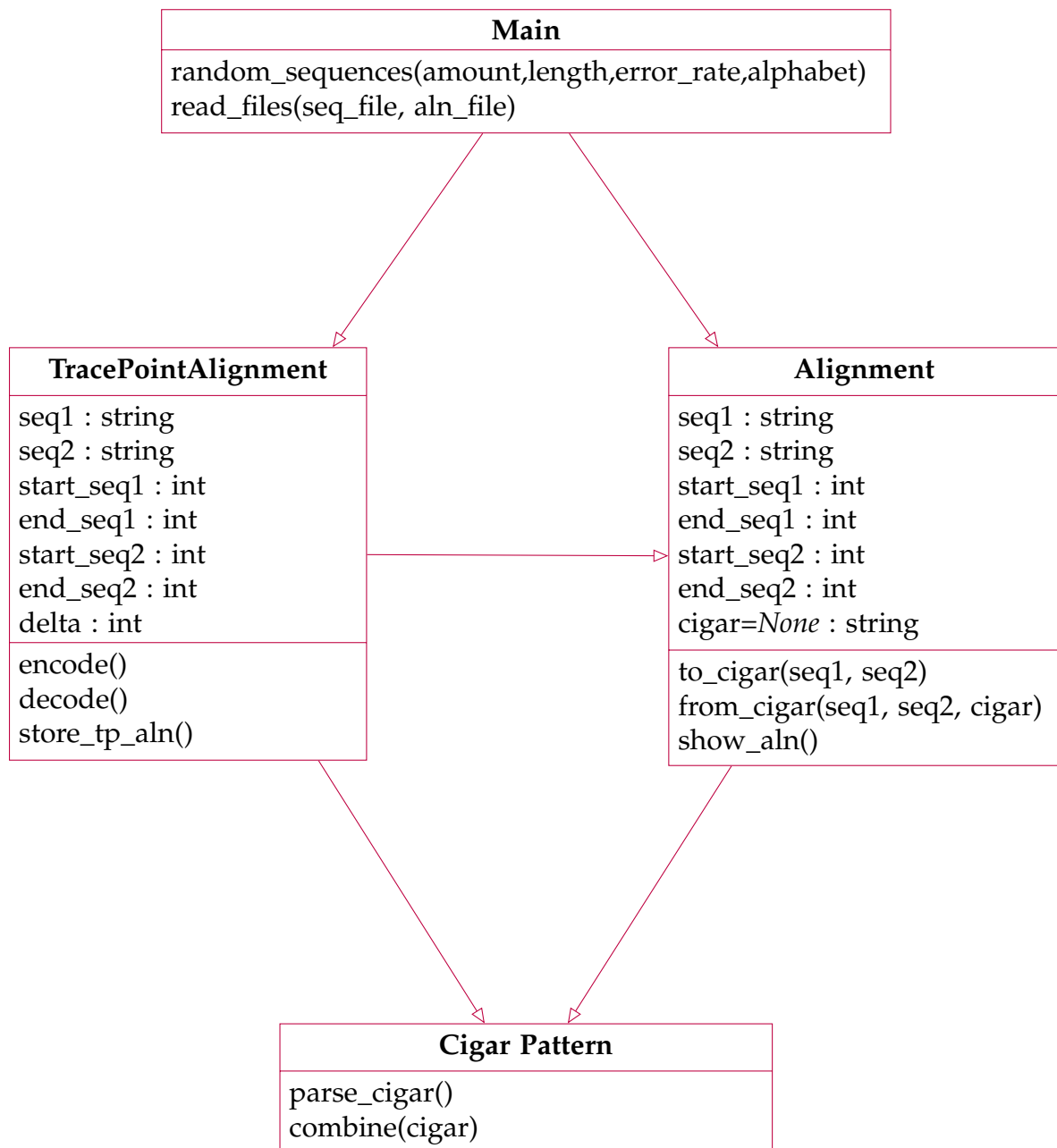


Abbildung 6.1: UML-Diagramm

## 6.1 Funktionalität

Die entscheidenden Funktionen dieser Implementierung sind die `encode()`-Funktion, welche aus einem gegebenen CIGAR-String die Trace Points mit Hilfe des  $\Delta$ -Wertes extrahiert und die `decode()`-Funktion, welche für gegebene Trace Points und ein  $\Delta$  einen CIGAR-String rekonstruiert. Der Pseudocode der `encode()`-Funktion ist in Algorithmus 2 und der Pseudocode der `decode()`-Funktion in Algorithmus 3 beschrieben.

Die `encode`-Funktion extrahiert aus dem gegebenen CIGAR-String die Trace Points, welche dann zusammen mit dem  $\Delta$ -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  $\Delta$ -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  $\geq 0$ )
4:   assert(start_seq1 < end_seq1)
5:    $p \leftarrow MAX(1, \lceil start\_seq1 / \Delta \rceil)$ 
6:    $\tau \leftarrow \lfloor end\_seq1 / \Delta \rfloor - \lfloor start\_seq2 / \Delta \rfloor$ 
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  $\Delta$ -Wert und das Trace Point Array übergeben.

```

1: function decode( $seq1, seq2, \Delta, TP$ )
2:   assert( $|seq1|, |seq2|, \Delta, |TP| > 0$ )
3:    $cig \leftarrow$  empty String
4:   for  $i \leftarrow 0$  upto  $|TP|$  do
5:     if  $i = 0$  then
6:        $cig.append(cigar(seq1[0... \Delta], seq2[0...TP[i] + 1]))$ 
7:     else if  $i = |TP| - 1$  then
8:        $cig.append(cigar(seq1[i \cdot \Delta...|seq1|], seq2[TP[i - 1] + 1...|seq2|]))$ 
9:     else
10:       $cig.append(cigar(seq1[i \cdot \Delta...(i + 1) \cdot \Delta], seq2[TP[i - 1] + 1]...TP[i] +$ 
11:         $1))$ 
12:     end if
13:   end for
14:    $cig \leftarrow combine(cig)$ 
15:   return  $cig$ 
16: end function
17: function combine( $cigar$ )
18:    $cig \leftarrow$  empty String
19:    $tmp \leftarrow 0$ 
20:   for each  $cig\_count, cig\_symbol$  in  $cigar$  do
21:      $tmp \leftarrow tmp + previous\_cig\_count$ 
22:     if  $cig\_symbol = previous\_cig\_symbol$  then
23:       if not last element in  $cigar$  then
24:          $tmp \leftarrow 0$ 
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  $\Delta$  ist, desto weniger Trace Points werden gespeichert und umso länger dauert die Berechnung, um die Teil-Alignments zu rekonstruieren. Bei einem kleinen  $\Delta$  werden analog mehr Trace Points gespeichert, aber die Rekonstruktionszeit der Teil-Alignments ist geringer.

Mithilfe von  $\Delta$  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 Programmierung*  $\tilde{A}^{\frac{1}{4}}r$  *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](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
- [Wiese 2016] WIESE, Thorben: *Implementierung des Trace Point Verfahrens im Rahmen der Bachelorarbeit "Speichereffizienten Methoden zur Repräsentation von paarweisen Sequenz-Alignments"*. [https://github.com/thorbenwiese/bachelorarbeit\\_wiese.git](https://github.com/thorbenwiese/bachelorarbeit_wiese.git). Version: Oktober 2016
-

