Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften
Department Informatik

Bachelorarbeit

Speichereffiziente Methoden zur Repräsentation von paarweisen Sequenz-Alignments

Thorben Wiese

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

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Ein Sequenzalignment wird in der Bioinformatik dazu verwendet, zwei oder mehrere Sequenzen zum Beispiel von 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.

Die effiziente Speicherung der Repräsentation paarweiser Sequenz-Alignments ist von großer Bedeutung, um den Speicherbedarf einer Repräsentation zu verringern. In der Bioinformatik nimmt die Anzahl der zu vergleichenden Sequenzen immer weiter zu. Für viele Sequenzen oder Sequenzabschnitte müssen Alignments gespeichert werden. Um den Speicherbedarf einer solchen Repräsentation zu verringern, ist eine Kodierung sinnvoll. Die zu kodierende Information enthält allgemeine Informationen zu den Sequenzen, etwa die Start- und End-Positionen. Die Sequenzen selber müssen nicht kodiert werden, da sie Teil des Ergebnisses sind.

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 verschiedenen Operationen, um die Symbole der einen Sequenz in die andere zu überführen, können je nach Verfahren unterschiedlich dargestellt und gewichtet werden. Bei Gleichheit wird sie als 'match', bei einer Substitution als 'mismatch', bei einer Löschung als 'deletion' und bei einer Einfügung als 'insertion' dargestellt.

Um die verschiedenen Sequenzen vergleichen zu können, berechnet man für das Alignment 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. Ähnliche Sequenzen haben einen hohen Score und geringe Kosten und unterschiedliche Sequenzen analog einen kleinen Score und hohe Kosten.

Das in dieser Arbeit vorgestellte speichereffiziente Verfahren der Trace Points ist als Python- und C-Implementierung in meinem GitHub-Repository (`https://www.github.com/thorbenwiese/bachelorarbeit_wiese.git`) zu finden.

2 Methoden

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.

Deletionen und Insertionen werden auch Fehler genannt und das Verhältnis der Fehler zur Gesamtanzahl der Edit-Operationen ist die Fehlerrate einer Sequenz.

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.

Ein Alignment wird in drei Zeilen so geschrieben, dass in der ersten Zeile die Sequenz u und in der dritten Zeile die Sequenz v enthalten ist. 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 $A = (a \to a, c \to c, t \to t, \varepsilon \to a, g \to g, a \to a, a \to a, c \to \varepsilon, t \to t)$ 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 $\delta(a \to b) \geqslant 0$ für alle Substitutionen $a \to b$ und $\delta(\alpha \to \beta) > 0$ für alle Einfügungen und Löschungen $\alpha \to \beta$ gegeben. Die Kosten für ein Alignment $A = (\alpha_1 \to \beta_1, ..., \alpha_h \to \beta_h)$ ist die Summe der Kosten aller Edit-Operationen des Alignments.

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

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].

Ein Alignment kann für eine bekannte Edit-Distanz e mit der Einheitskostenfunktion in $O(m + n + e^2)$ Zeit berechnet werden [Kurtz a, S. 41-42].

2.1 Kodierung

In der Informatik ist eine Kodierung eine Zuweisung von Bits zu jedem Symbol a aus einem Alphabet $\mathcal{A}$. Die Kombination von Bits, die einem Symbol zugeordnet wird, wird Codewort genannt. Eine Kodierung, deren Codewörter nie der Anfang eines anderen Codewortes sind, nennt man einen präfixfreien Code. Dieser hat die Eigenschaft, dass alle Codewörter eindeutig sind und die Dekodierung somit deterministisch ist.

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

2.1.1 Unäre Kodierung

Für ein gegebenes Alphabet $\mathcal{A}$ und eine Häufigkeitsverteilung der Symbole aus $\mathcal{A}$ in einer Sequenz kodiert die unäre Kodierung jedes Symbol in Abhängigkeit von seiner Häufigkeit 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 den Wert 1, was den Vorteil hat, dass lediglich die Länge variiert. Somit entspricht die Länge eines Codewortes dem Wert, den es kodiert, bzw. die Position in einer nach der Häufigkeit sortierten Liste. Die Gesamtgröße einer unären Kodierung ist somit die Häufigkeit $h(a)$ eines Symbols $a \in \mathcal{A}$, multipliziert mit der Codewortlänge $c(a)$ des Symbols, summiert über alle Symbole.

$$\sum_{a \in \mathcal{A}} h(a) \cdot c(a)$$

2.1.2 Naive Binäre Kodierung

Bei einer naiven binären Kodierung wird jedes Symbol a aus dem Alphabet $\mathcal{A}$ der zu kodierenden Symbole mit $\lceil \log_2 |\mathcal{A}| \rceil$ Bit kodiert. Dies hat den Vorteil, dass alle Codewörter die gleiche Länge haben. Diese Art der Kodierung berücksichtigt jedoch lediglich die absolute Anzahl der zu kodierenden Symbole und nicht deren Häufigkeitsverteilung, was eine effiziente Kodierung bei Symbolen mit der selben Häufigkeit ermöglicht, bei einer Abweichung der Häufigkeiten aber keinen Vorteil daraus ziehen kann. Die Größe der Kodierung berechnet sich somit aus dem Produkt der Anzahl der Symbole, multipliziert mit der Anzahl der benötigten Bits für jedes Symbol.

$$\sum_{a \in \mathcal{A}} |\mathcal{A}| \cdot \lceil \log_2 |\mathcal{A}| \rceil$$

2.1.3 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 bestimmt. Hierbei werden häufig vorkommende Symbole durch kurze Codewörter kodiert und weniger häufige durch längere Codewörter. Somit lässt sich eine Kodierung ermöglichen, die im Durchschnitt weniger Bit pro Symbol beansprucht als etwa die naive binäre Kodierung, bei der die Häufigkeitsverteilung der Symbole nicht berücksichtigt wird. Jedes Codewort ist dabei nie der Anfang eines anderen Codewortes. Dies macht den Code präfixfrei, so dass die Codewörter eindeutig zugeordnet werden können [Moffat u. Turpin 2002, S. 53-57].

Als Datenstruktur der zu kodierenden Symbole wird hierbei eine priorisierte Queue verwendet, welche nach den Häufigkeiten der Symbole sortiert ist. Der Huffman-Algorithmus wählt immer die zwei Symbole mit den geringsten Häufigkeiten aus und fügt sie jeweils als ein Symbol zusammen, bis am Ende alle Symbole zusammengefügt wurden. So wird ein Baum mit den kodierten Symbolen als Blätter aufgebaut. Anschließend können durch das Hinzufügen der Kantenbeschriftungen 0 und 1 für die Kanten eines Knotens die Kodierungen der einzelnen Symbole durch die Kantenbeschriftungen von der Wurzel aus bis zu den Blättern bestimmt werden. Für die Dekodierung der Codewörter wird somit

Algorithmus 1 Pseudocode des Huffman-Algorithmus

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

Ausgabe: Der Huffman-Baum des Alphabets $\mathcal{A}$.

```
 1: function Huffman(A, p : A → ℕ)
 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
```

der Huffman-Baum benötigt, um Pfad... . Der Pseudocode des Algorithmus ist in Algorithmus 1 dargestellt [Kurtz b].

Immer dann, 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. Diese Regel ist im Pseudocode als $extractmin$ in Zeile 11 und 12 beschrieben. Hierdurch wird die Kodierung eindeutig und läuft somit deterministisch ab. Diese Art der Kodierung nennt man eine 'kanonische' Kodierung bzw. einen kanonischen Huffman-Algorithmus. Falls keine Regel eingeführt wird, werden die Symbole zufällig ausgewählt und die Kodierung ist nicht-deterministisch.

Die Gesamtanzahl der Bits für die Huffman-Kodierung ergibt sich, wie bei der unären Kodierung, durch das Produkt der Codewortlänge $c(a)$ eines Symbols $a \in \mathcal{A}$ multipliziert mit der Häufigkeit $h(a)$ des Symbols, aufsummiert über alle

Symbole.

$$\sum_{a \in \mathcal{A}} h(a) \cdot c(a)$$

Der kanonische Huffman-Algorithmus benötigt aufgrund der oben genannten Eigenschaften statt des gesamten Huffman-Baums lediglich die Anzahl der Codewörter für jede vorhandene Codewortlänge, sowie die sortierten Symbole für die Dekodierung. Diese zusätzliche Information muss zu der Gesamtgröße der Kodierung hinzuaddiert werden [Matai u. a. 2014].

2.2 CIGAR-Strings

Ein Dateiformat, das zur Speicherung von Alignments verwendet wird, ist das SAM-Format oder die binär komprimierte Version BAM. Dieses Format codiert ein Alignment in einem sogenannten CIGAR-String, der aus einzelnen Zeichen besteht, die jeweils eine Edit-Operation bezeichnen. Eine vereinfachte Version der CIGAR-Strings, die in dieser Arbeit verwendet wird, kodiert für eine Substitution ein M, für eine Insertion ein I und für eine Deletion ein D. Gleiche aufeinanderfolgende Operationen werden als Kombination von Quantität und Symbol geschrieben. Die Sequenzen selbst werden hierbei nicht mit kodiert, sondern lediglich die Edit-Operationen auf den Sequenzen.

Definition 1. Ein CIGAR-String $C = c_1 s_1 c_2 s_2 ... c_n s_n$ besteht aus Symbolen $s \in$ {M,I,D} und positiven ganzen Zahlen c_i. □

Ein CIGAR-String beschreibt ein Alignment, da mit c_1 Edit-Operationen vom Typ s_1 beginnt, gefolgt von c_2 Edit-Operationen vom Typ s_2 und so weiter. Dabei wird keine Information über die alignierten Sequenzen mitkodiert. Ein CIGAR-String kann daher verschiedene Alignments repräsentieren.

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].

2.2.1 Kodierung eines CIGAR-Strings

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

$\mathcal{A}_s = \{\text{M,I,D}\}$ 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
`4M1I1M1I1M1I1M2D1M1D1M1I8M1D7M1D5M1I4M` der Länge 19 repräsentiert werden kann.

Aus dem CIGAR-String ergibt sich das Alphabet $\mathcal{A}_c = \{1, 2, 4, 5, 7, 8\}$, welches alle Zahlen aus dem CIGAR-String enthält.

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 für die Kodierung des CIGAR-Strings.

Die unäre Kodierung des oben genannte CIGAR-Strings ist in Tabelle 2.1 beschrieben. Sie benötigt $32 + 35 = 67$ 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 2.1: Unäre Kodierung des CIGAR-Strings

Der kanonische Huffman-Algorithmus würde bei dem oben genannten Beispiel des CIGAR-Strings nach [Moffat u. Turpin 2002, S. 54] die Symbole wie in Tabelle 2.2 beschrieben kodieren. Die Huffman-Bäume beider Alphabete sind in Abbildung 2.1 dargestellt. Für die Dekodierung sind somit zusätzlich die Listen $(1, 2)$, (M,D,I) und $(1, 1, 0, 4), (1, 4, 2, 5, 7, 8)$ 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	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 2.2: Huffmann-Kodierung des CIGAR-Strings

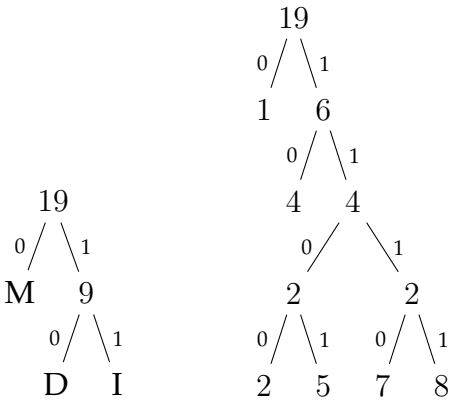

Abbildung 2.1: Huffman-Bäume für die Häufigkeitsverteilung der Symbole des CIGAR-Strings

2.3 Das 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 der Länge Δ zu unterteilen und die Endpunk-

te, sogenannte Trace Points, von Teilabschnitten des Alignments in der zweiten Sequenz abzuspeichern. Zur Rekonstruktion eines Gesamtalignments wird für jeden Teilabschnitt jeweils ein Teilalignment berechnet. Die Teilalignments können dann zu einem Gesamtalignment konkateniert werden.

Das Verfahren bietet den Vorteil, durch die Größe Δ 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 $k < \ell$ und sei $\Delta \in \mathbb{N}$. Sei außerdem p wie folgt definiert:

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

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 für alle $q, 0 \leqslant q \leqslant \tau - 1$, berechnet hierfür jeweils ein optimales Alignment und konkateniert die Alignments von den aufeinanderfolgenden Substring-Paaren zu A'.

Beispiel 4. Sei $u = u[0...37]$ und $v = v[0...37]$ und das folgende 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
```

Sei $i = k = 0$ und $j = \ell = 37$ und $\Delta = 15$. Das Alignment wird in $\tau = \left\lceil \frac{37}{15} \right\rceil - \left\lfloor \frac{0}{15} \right\rfloor = 3 - 0 = 3$ Abschnitte unterteilt. Der erste Abschnitt aligniert $u[0...14]$ und $v[0...15]$.

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

Der zweite Abschnitt aligniert $u[15...29]$ und $v[16...28]$.

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

Der dritte Abschnitt aligniert $u[30...37]$ und $v[29...37]$.

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

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

2.3.1 Differenzen-Kodierung

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

Anstatt jeden Wert 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. Wir definieren daher

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

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

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 2.2.1 mit den dazugehörigen TracePoints $L = (5, 10, 15, 20, 24, 28, 34)$ gegeben.

Für die Kodierung der Trace Points ist in diesem Beispiel somit eine Differenzen-Kodierung sinnvoll.

Daher ergibt sich $L_{diff} = (5, 5, 5, 5, 4, 4, 6)$.

Um aus den Trace Points ein neues Alignment rekonstruieren zu können, benötigt man zusätzlich mindestens den Δ-Wert, sowie die Start- und End-Positionen der Sequenzen die aligniert werden, damit die Grenzen der Substrings beider Sequenzen berechnet werden können. Hierfür kann der Δ-Wert zu L_{diff} hinzugefügt werden, da er bei Δ großen Teilabschnitten in u wahrscheinlich der dominierende Wert in L_{diff} sein wird.

Für das oben genannten Beispiel ergibt sich somit

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

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 2.3: Unäre Kodierung von $L_{diff} = (5, 5, 5, 5, 4, 4, 6)$ und $\Delta = 5$

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

Tabelle 2.4: Huffman-Kodierung von $L_{diff} = (5, 5, 5, 5, 4, 4, 6)$ und $\Delta = 5$

sowie das Alphabet $\mathcal{A} = \{4, 5, 6\}$ mit den Häufigkeiten aus Tabelle 2.3.

Bei der naiven binären Kodierung von L_{diff} ergibt sich analog zu 2.2.1 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 Größe der binären Kodierung für den CIGAR-String.

Die unäre Kodierung ergibt für dieses Beispiel die in Tabelle 2.3 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.

Der Huffman-Algorithmus kodiert nach [Moffat u. Turpin 2002, S. 54] die Symbole wie in Tabelle 2.4 aufgelistet. Der dazugehörige Huffmann-Baum aus 2.2 verdeutlicht die Kodierung der einzelnen Symbole, muss aber für den kanonischen Huffman-Algorithmus, wie in 2.2.1 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.

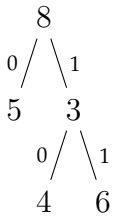

Abbildung 2.2: Huffman-Baum der Differenzen-Kodierung

2.4 Entropie der Methoden

Sei S eine Sequenz der Länge n in der die Symbole $a \in \mathcal{A}$ mit der Häufigkeit $h(a)$ auftreten. Die Entropie $H(S)$ beschreibt den durchschnittlichen Informationsgehalt für alle Symbole aus S in der Einheit $\frac{\text{Bit}}{\text{Symbol}}$.

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

Sie ist maximal, wenn $h(a) = h(b)$ für jeweils zwei verschiedene Symbole $a, b \in \mathcal{A}$ gilt [Mézard u. Montanari 2009].

Für den in 2.2.1 genannten CIGAR-String S mit der Häufigkeitsverteilung aus Tabelle 2.3 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 2.3.1 genannten Trace Point Differenzen des selben Alignments inklusive des Δ-Wertes ergeben analog

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

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

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

Abbildung 2.3: Anzahl der Bits für die Kodierung des Beispiel-Alignments

Zusammenfassend ergibt sich für die Größen der Kodierungen des Alignments aus Beispiel 1 Tabelle 2.3.

3 Resultate

Die Verfahren, um die in diesem Kapitel beschriebenen Resultate zu berechnen, wurden in Python implementiert.

Für die empirische Untersuchung der Größe der einzelnen Kodierungen wurden 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. Hierfür wurden DNA-Sequenzen aus zufällig aneinander gereihten Basen und daraus abgewandelte Sequenzen mit der jeweiligen Fehlerrate durch Austauschen oder Löschen bzw. Einfügen von Basen berechnet. Aus jedem Sequenzpaar wurde dann ein optimales Alignment berechnet, aus dem der CIGAR-String extrahiert wurde. Dieser konnte dann mit den oben beschriebenen Verfahren kodiert und die Größe der Kodierung bestimmt werden.

Die gemessenen Werte der Häufigkeitsverteilung der Kodierung der CIGAR-Strings sind in Tabelle 3.6 enthalten und in Abbildung 3.1 grafisch dargestellt.

Für die Kodierung der Trace Point Differenzen werden die Trace Points anhand des Δ-Wertes berechnet und die Differenzen der Trace Points bestimmt. Diese werden dann mit den oben beschriebenen Verfahren kodiert und dabei die Größe der Kodierung bestimmt.

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

Die Werte der unären und Huffman-Kodierung der Trace Point Differenzen für $\Delta = 200$ und einer Fehlerrate von 15% sind in Tabelle 3.6 beschrieben und in Abbildung 3.2 dargestellt.

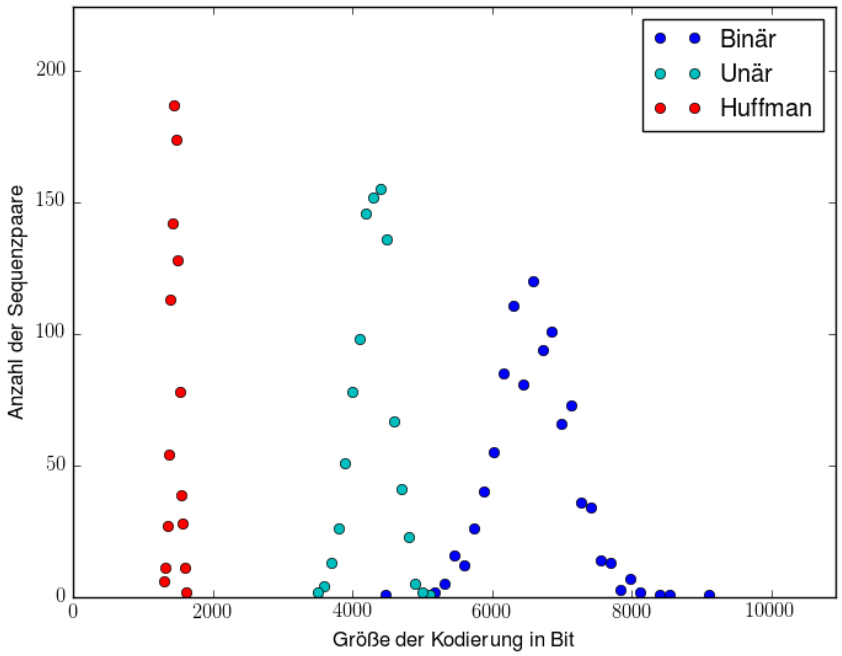

Abbildung 3.1: Häufigkeitsverteilung der Kodierungen für 1 000 CIGAR-Strings von DNA Sequenzen mit je 5 000 Basenpaaren und einer Fehlerrate von 15%.

Die Entropie für die CIGAR-String Repräsentation liegt für die gegebenen Sequenzpaare, wie in Abbildung 3.5 zu erkennen ist, zwischen 1 280 und 2 280 Bit, wobei der Durchschnittswert bei 1 755.79 Bit liegt. Die Werte sind normalverteilt, wobei das Maximum bei 225 Sequenzpaaren mit etwa 1 800 Bit liegt.

Die Differenzen der Trace Points weisen hingegen eine Entropie im Bereich von 91 bis 212 Bit auf, wobei der Durchschnittswert von 134.69 Bit nur etwa 7.6% des Durchschnittwertes der CIGAR-String Repräsentation ausmacht. Das Maximum der Entopie-Werte liegt hier bei etwa bei 220 Sequenzpaaren mit 130 Bit.

Zusammenfassend ergibt sich für die Resultate die Tabelle 3.6.

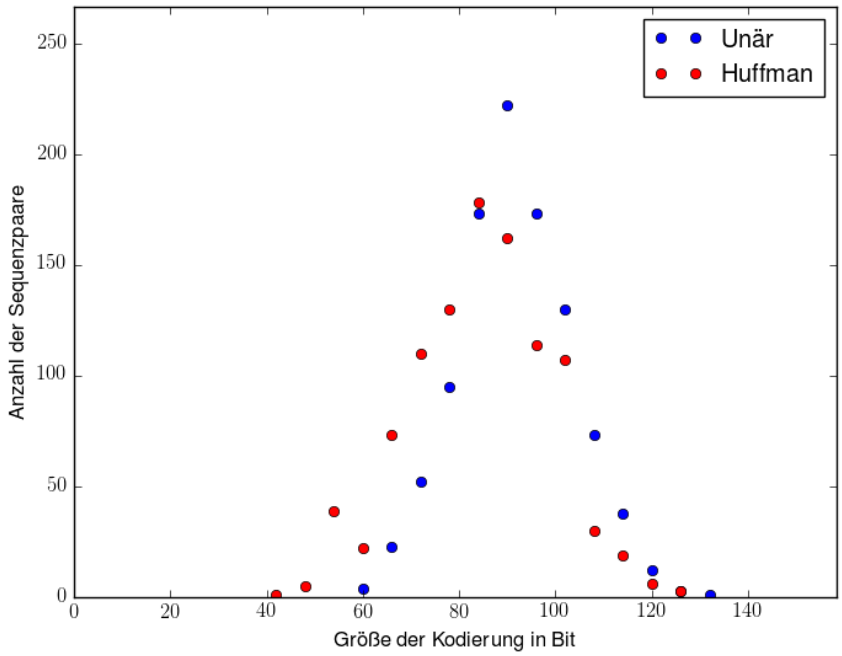

Abbildung 3.2: Häufigkeitsverteilung der Kodierungen der Trace Point Differenzen für 1 000 DNA-Sequenzen mit je 5 000 Basenpaaren, einer Fehlerrate von 15% und einem Δ-Wert von 200.

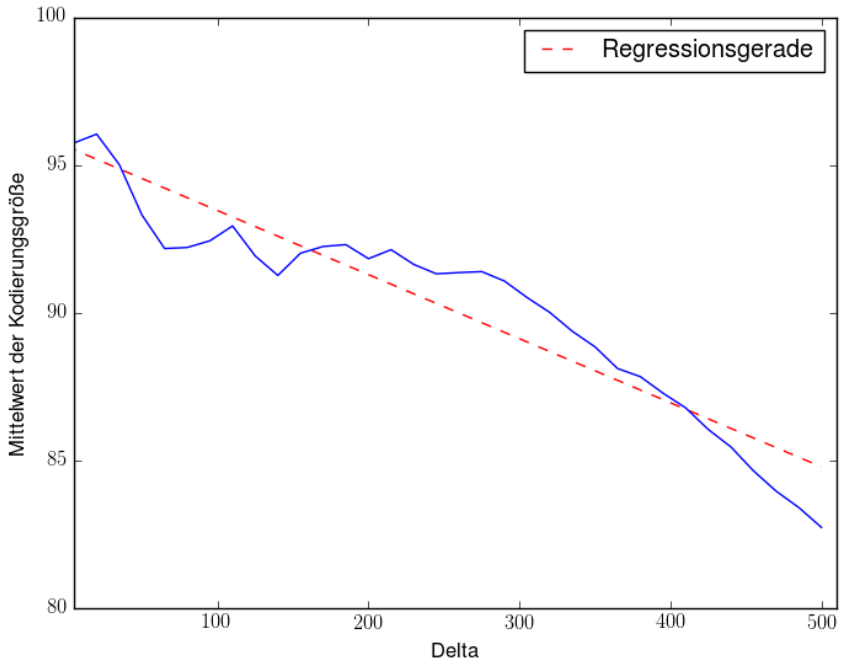

Abbildung 3.3: Mittelwerte der Huffman-Kodierung der Trace Point Differenzen für 5 000 Basenpaaren, einer Fehlerrate von 15% und Δ-Werten von 5 bis 500.

	Delta		Fehlerrate	
Wert	35	500	5%	95%
Größe	95	76	44	227

Delta	35	500
Größe	95	76

Fehlerrate	5%	95%
Größe	44	227

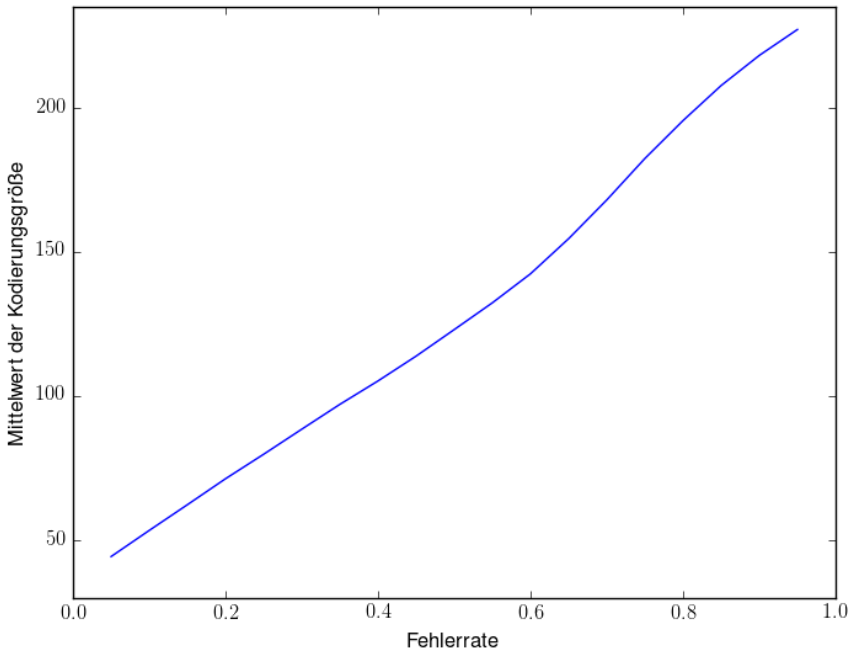

Abbildung 3.4: Mittelwerte der Huffman-Kodierung der Trace Point Differenzen für 5 000 Basenpaaren, einem Δ-Wert von 100 und Fehlerraten von 5% bis 95%.

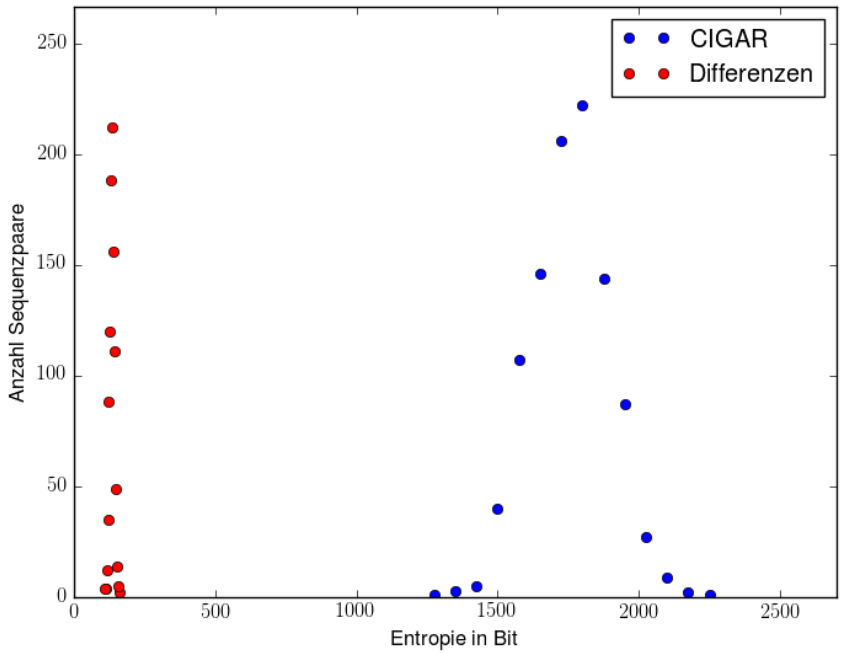

Abbildung 3.5: Entropie von 1 000 CIGAR-Strings von DNA-Sequenzen mit je 5 000 Basenpaaren, einem Δ-Wert von 100 und einer Fehlerrate von 15%

	CIGAR-String			Trace Point Differenzen ($\Delta = 200$)		
	Min	Max	∅	Min	Max	∅
Binäre Kodierung	4 480	9 100	6 595	125	125	125
Unäre Kodierung	3 500	5 100	4 290	60	132	90
Huffman-Kodierung	1 300	1 625	1 458	42	126	84
Entropie	1 280	2 280	1 755	91	212	134

Abbildung 3.6: Größe der Kodierungen und Entropie in Bit

4 Diskussion

In diesem Kapitel werden die Resultate aus Kapitel 3 interpretiert und die Qualität der Kodierungen in Bezug auf eine speichereffiziente Repräsentation eines Alignments betrachtet.

4.1 CIGAR-Kodierung

Die Kodierungen der CIGAR-Strings, wie sie in Abbildung 3.1 grafisch und in Tabelle 3.6 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. Insertions und Deletions treffen dazu selten hintereinander auf, was einen Schwerpunkt der Quantität '1' zufolge hat. Für diese 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 z.B. 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 und es zeigt sich, dass die Huffman-Kodierung wie oben beschrieben nur etwa 34% des Speicherbedarfs der unären Kodierung benötigt.

Die Huffman-Kodierung ist somit für CIGAR-Strings die speichereffizienteste der drei Kodierungen.

4.2 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 3.2 grafisch dargestellt und in der Tabelle 3.6 numerisch beschrieben. In Abbildung **??** sind die Auswirkungen der Wahl des Δ-Parameters dargestellt und in Abbildung **??** die Auswirkungen der Fehlerrate der Sequenzen.

Für einen Δ-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 Δ großen Abschnitten der u-Sequenz aligniert werden, sind diese üblicherweise auch genauso oder ähnlich groß wie das Δ. 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 Δ-Wertes kann die Größe der Kodierung deutlich beeinflussen, da dieser Wert die Anzahl der zu speichernden Symbole bestimmt. Bei einem großen Δ müssen weniger Symbole gespeichert werden und bei einem kleinen Δ analog mehr. So benötigt die Huffman-Kodierung für ein $\Delta = 500$ folglich nur etwa 66% des Speichers im Vergleich zur Kodierung mit $\Delta = 50$.

Die Fehlerrate der Sequenzen hat ebenfalls einen großen Einfluss auf die Größe der Kodierung. Je größer die Fehlerrate, umso größer ist die Anzahl der Insertions und Deletions im Alignment. Bei einer kleinen Fehlerrate ist somit eine hohe Anzahl an Matches vorhanden. Da die Trace Points ein Vielfaches von Δ sind und somit die Differenzen der Trace Points ebenfalls Δ groß sind, müssen daher deutlich weniger unterschiedliche Symbole kodiert werden. 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. Damit ist die Huffman-Kodierung, wie bei den CIGAR-Strings, das effizienteste Kodierungs-Verfahren.

4.3 Entropie beider Methoden

Die Entropie der CIGAR-Strings und der Trace Point Differenzen sind in Abbildung 3.5 grafisch dargestellt und in der Tabelle 3.6 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 Bit auf. Dies bedeutet, dass sie etwa 13 mal so viel wie die Entropie der Trace Point Differenzen ausmacht, welche bei durchschnittlich 134 Bit liegt.

5 Implementierung

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.

Der Aufbau der Implementierung ist als UML-Klassendiagramm in Abbildung 5.1 dargestellt.

Die Implementierung besteht aus der Main-Datei, welche als Eingabeparameter die Sequenzen, deren Start- und Endpositionen, sowie der Δ-Wert entgegennimmt und diese in einer 'Trace Point Liste' speichert. Anschließend wird mithilfe eines dynamischen Programmieralgorithmus' ein optimales lokales Alignment der zu alignierenden Teilabschnitte der Sequenzen erzeugt und als Liste von Edit-Operationen gespeichert. Die Trace Point Liste und die Liste der Edit-Operationen werden dann an die encode-Funktion übergeben, die in einem eigenen TracePoint-Modul definiert ist.

Diese Funktion unterteilt die Sequenzen wie in Kapitel 2.3 beschrieben und bestimmt die Trace Points, welche dann ebenfalls in der Trace Point Liste gespeichert werden. Der Pseudocode der encode-Funktion ist in Algorithmus 2 beschrieben. Hierbei geht die Information, wie die jeweiligen Intervalle zwischen den Trace Points zu den komplementären Intervallen in der Ursprungssequenz aligniert werden, verloren.

Die Dekodierung der Trace Points erfolgt in der decode-Funktion, welche ebenfalls im TracePoint-Modul definiert ist. Sie nimmt eine Trace Point Liste als Parameter entgegen und berechnet für jedes Paar von zu alignierenden Teilabschnitten eine neue Liste von Edit-Operationen und konkateniert diese abschließend zu einer Gesamtliste und gibt sie als Rückgabewert zurück. Der Pseudocode der decode-Funktion in Algorithmus 3 zu finden. Hierbei ist jedoch nicht gewährleistet, dass das neue Alignment dem alten entspricht. Das neue Alignment hat jedoch mindestens die gleiche Edit-Distanz wie das alte Alignment.

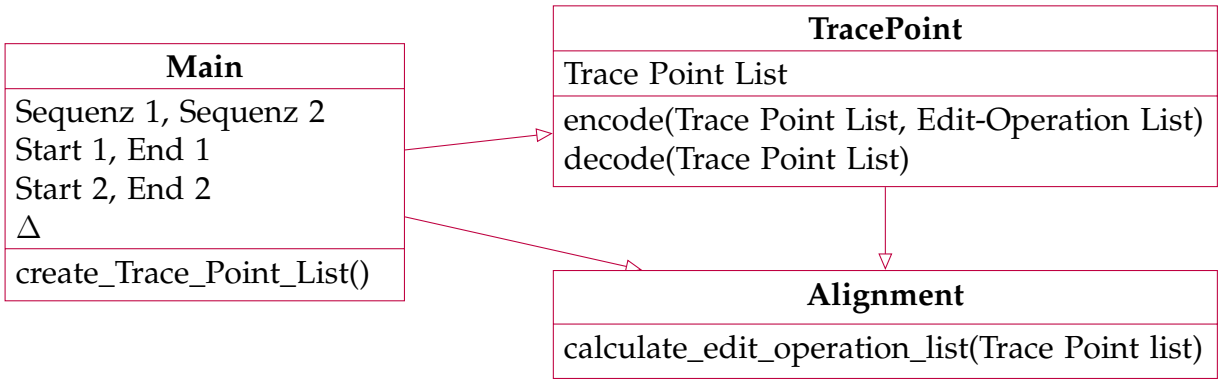

Abbildung 5.1: UML-Diagramm

Algorithmus 2 Berechnung der Trace Points aus einer gegebenen Liste von Edit-Operationen

Parameter: Der Funktion wird eine Referenz auf eine Trace Point Liste $*tp\_list$ und eine Referenz auf die Liste der Edit-Operationen $*eoplist$ übergeben.

```
1:  function encode(*tp_list, *eoplist)
2:      assert(tp_list.start1, tp_list.start2 ⩾ 0)
3:      assert(tp_list.start1 < tp_list.end1)
4:      p ← MAX(1, ⌈start1/Δ⌉)
5:      τ ← ⌈end1/Δ⌉ − ⌊start1/Δ⌋
6:      uTP ← Array for interval termini in the first sequence
7:      for q ← 0 upto τ do
8:          uTP[q] ← (p + q) · Δ − 1
9:      end for
10:     uChars, vChars, count ← 0
11:     TP ← Array for Trace Points
12:     for each Operation in eoplist do
13:         for i ← 0 upto eop_count − 1 do
14:             if eop_type = 'Insertion' then
15:                 increment uChars
16:             else if eop_type = 'Deletion' then
17:                 increment vChars
18:             else
19:                 increment uChars, vChars
20:             end if
21:             if uChars = uTP[count] then
22:                 tp_list.TP.append(vChars)
23:             end if
24:             if count = τ − 1 then
25:                 tp_list.TP_len ← |tp_list.TP|
26:                 break
27:             else
28:                 increment count
29:             end if
30:         end for
31:     end for
32: end function
```

Algorithmus 3 Berechnung einer Liste von Edit-Operationen aus einer gegebenen Trace Point Liste

Parameter: Der Funktion wird eine Referenz auf eine Trace Point Liste $*tp\_list$ übergeben.

Ausgabe: Die Funktion liefert ein konkatenierte Liste von Edit-Operationen zurück.

1: **function** *decode*($*tp\_list$)
2: $eoplist \leftarrow$ empty list of edit operations
3: **for** $k \leftarrow 0$ **upto** $tp\_list.TP\_len - 1$ **do**
4: **if** k = 0 **then**
5: $eoplist.append(seq1[0...\Delta], seq2[0...tp\_list.TP[k] + 1])$
6: **else if** k = | TP | - 1 **then**
7: $eoplist.append(seq1[k{\cdot}\Delta...|seq1|], seq2[tp\_list.TP[k{-}1]{+}1...|seq2|]))$
8: **else**
9: $eoplist.append(seq1[k \cdot \Delta...(k + 1) \cdot \Delta],$
10: $seq2[tp\_list.TP[k - 1] + 1]...tp\_list.TP[k] + 1))$
11: **end if**
12: **end for**
13: **return** $eoplist$
14: **end function**

6 Fazit

Das Verfahren der Trace Point Differenzen stellt im Vergleich zu der üblichen Repräsentation von Alignments als CIGAR-String eine deutlich speichereffizientere Methode dar. Insbesondere durch eine Huffman-Kodierung der Differenzwerte kann bei den in dieser Arbeit errechneten Größen eine Speicherersparnis von etwa 94% gegenüber den CIGAR-Strings erzielt werden. Hierbei spielt jedoch der vorher definierte positive Parameter Δ eine entscheidende Rolle.

Je größer Δ gewählt ist, umso 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 Programmierung für 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

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: _____