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

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.

Die Edit-Operationen

Die hier eingeführten Begriffe werden in [Kurtz, S. 5-7, 14-16] definiert.

Sei $\mathcal{A}$ eine endliche Menge von Buchstaben, die man Alphabet nennt. Für DNA-Sequenzen verwendet man üblicherweise die Menge der Basen, also $\mathcal{A} = \{a, c, g, t\}$. $\mathcal{A}^i$ sei die Menge der Sequenzen der Länge i aus $\mathcal{A}$ und ε sei die leere Sequenz. Formal ausgedrückt ist eine Edit-Operation ein Tupel

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

Eine äquivalente Schreibweise von (α, β) ist $\alpha \rightarrow \beta$. Es gibt drei verschiedene

Edit-Operationen

$$a \to \varepsilon \text{ ist eine Deletion für alle } a \in \mathcal{A}$$
$$\varepsilon \to b \text{ ist eine Insertion für alle } b \in \mathcal{A}$$
$$a \to b \text{ ist eine Substitution für alle } a, b \in \mathcal{A}$$

Dabei ist zu beachten, dass $\varepsilon \to \varepsilon$ keine Edit-Operation darstellt.

Ein Alignment von zwei Sequenzen u und v lässt sich nun als eine Sequenz $(\alpha_1 \to \beta_1, ..., \alpha_h \to \beta_h)$ von Edit-Operationen definieren, sodass $u = \alpha_1 ... \alpha_h$ und $v = \beta_1 ... \beta_h$ gilt.

Die Edit-Distanz

Sei eine Kostenfunktion δ mit $\delta(a \to b) \geq 0$ 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, S. 19-21].

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

2 CIGAR-Strings

Ein Dateiformat, welches zur Speicherung von Alignments verwendet wird, ist das SAM-Format oder die binär komprimierte Version BAM. Dieses codiert ein Alignment in einem sogenannten CIGAR-String der aus einzelnen Zeichen besteht, die jeweils eine Edit-Operation bezeichnen, also M für eine Substitution, I für eine Insertion und D für eine Deletion. Gleiche aufeinanderfolgende Operationen werden als Kombination von Quantität und Symbol geschrieben.

Beispiel 1: Sei $u =$ `actgaact`, $v =$ `actagaat` und das Alignment $A = (a \rightarrow a, c \rightarrow c, t \rightarrow t, ...)$ gegeben.

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

Ein Alignment wird üblicherweise in drei Zeilen geschrieben, wobei in der ersten Zeile die Sequenz u und in der dritten Zeile die Sequenz v geschrieben wird. In der mittleren Zeile symbolisiert das Zeichen '|' eine Substitution, wobei üblicherweise nur ein Match markiert wird. Außerdem wird ein ε aus der Edit-Operation in diesem Fall durch das Zeichen '-' dargestellt.

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

2.1 Komplexität

2.2 Speicherverbrauch

2.2.1 Beispiel

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 `4M1D1M1D1M1D1M2I1M1I1M1D8M1I7M1I5M1D4M` repräsentiert werden kann.

Im Folgenden vergleiche ich für diese Art der Alignment-Repräsentation die Verfahren der naiven binären Kodierung, der unären Kodierung und der Huffman-Kodierung.

Sei das Alphabet $\mathcal{A} = \{M, I, D, 1, 2, 4, 5, 7, 8\}$, welches alle Symbole aus dem CIGAR-String enthält, sowie die relativen Wahrscheinlichkeiten $p(c)$ jeden Symbols $c \in \mathcal{A}$ gegeben.

c	Häufigkeit	$p(c)$
1	13	$\frac{13}{38}$
M	10	$\frac{10}{38}$
D	5	$\frac{5}{38}$
I	4	$\frac{4}{38}$
4	2	$\frac{2}{38}$
2	1	$\frac{1}{38}$
5	1	$\frac{1}{38}$
7	1	$\frac{1}{38}$
8	1	$\frac{1}{38}$

Bei einer naiven binären Kodierung wird jedes Symbol $c \in \mathcal{A}$ mit $\lceil \log_2(n) \rceil, n = |\mathcal{A}|$ Bit kodiert, also $\lceil \log_2(38) \rceil = 6$ Bit pro Symbol. Insgesamt ergibt das somit $38 \cdot 6 = 228$ Bit.

Die unäre Kodierung kodiert jedes Symbol nach der Häufigkeit des Auftretens im Alphabet mit $i-1$ '1'-Bits, gefolgt von einem '0'-Bit, wobei i die Position des Symbols in einer nach der Häufigkeit absteigend sortierten Liste ist. Das am Häufigsten auftretende Symbol des Alphabets wird also mit '0', das zweithäufigste mit '10', das dritthäufigste mit '110' usw. kodiert. [Moffat u. Turpin 2002, S. 29-30]

Der oben genannte CIGAR-String wird demnach wie folgt unär kodiert:

Symbol	Kodierung	Anzahl Bits
1	0	$13 \cdot 1 = 13$
M	10	$10 \cdot 2 = 20$
D	110	$5 \cdot 3 = 15$
I	1110	$4 \cdot 4 = 16$
4	11110	$2 \cdot 5 = 10$
2	111110	$1 \cdot 6 = 6$
5	1111110	$1 \cdot 7 = 7$
7	11111110	$1 \cdot 8 = 8$
8	111111110	$1 \cdot 9 = 9$
Gesamtanzahl:		104

Insgesamt benötigt die unäre Kodierung also 104 Bit.

Der durchschnittliche Bitverbrauch für ein Symbol beträgt

$$\frac{104}{38} \approx 2.74 \text{ Bit.}$$

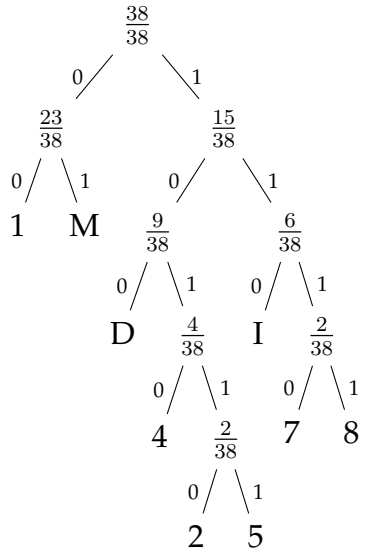
Bei einer *minimalen* binären Kodierung, wie sie auch im Huffman-Alogrithmus verwendet wird, werden die Längen der Codewörter anhand der relativen Wahrscheinlichkeit des Symbols im Alphabet angepasst. Somit lässt sich eine Kodierung ermöglichen, welche im Durchschnitt weniger Bit pro Symbol beansprucht [Moffat u. Turpin 2002, S. 53-57].

Der Huffman-Algorithmus würde bei dem oben genannten Beispiel des CIGAR-Strings wie folgt ablaufen:

Ausführung des Huffman-Algorithmus ergibt nach [Moffat u. Turpin 2002, S. 54]:

Symbol	Kodierung	Anzahl Bits
1	00	$13 \cdot 2 = 26$
M	01	$10 \cdot 2 = 20$
D	100	$5 \cdot 3 = 15$
I	110	$4 \cdot 3 = 12$
4	1010	$2 \cdot 4 = 8$
7	1110	$1 \cdot 4 = 4$
8	1111	$1 \cdot 4 = 4$
2	10110	$1 \cdot 5 = 5$
5	10111	$1 \cdot 5 = 5$
Gesamtanzahl:		99

und der dazugehörige Huffman-Baum:



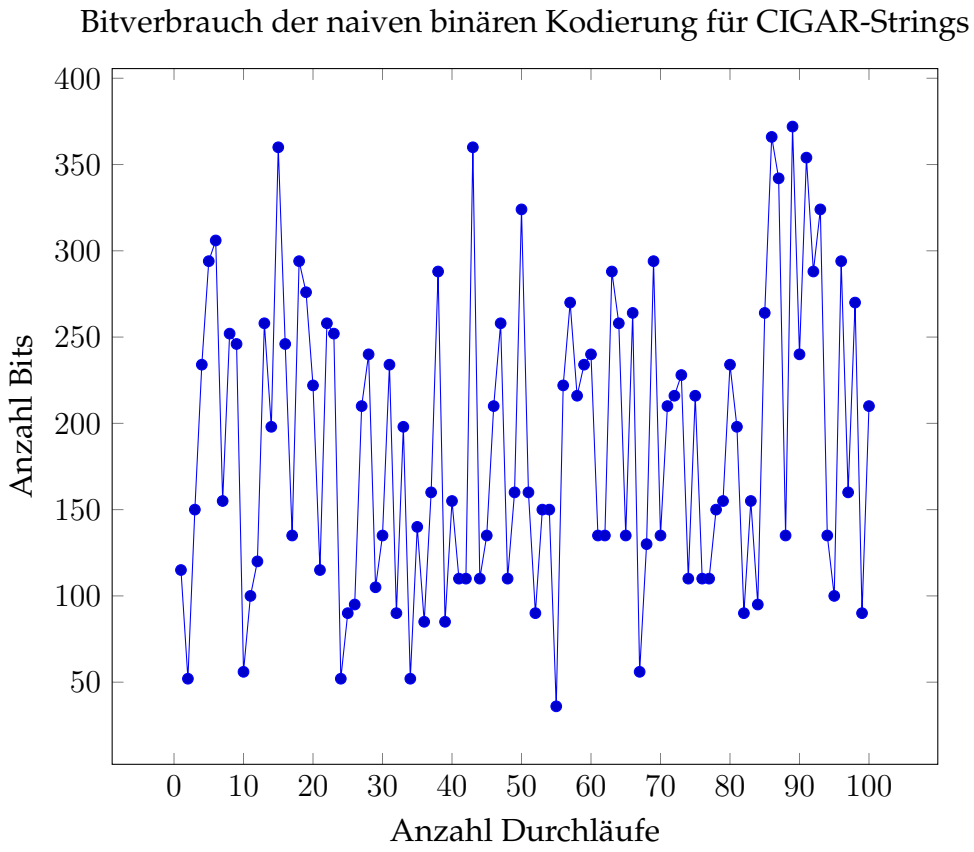
Der gesamte Bitverbrauch dieser Kodierung ist demnach 99 Bit.

Der durchschnittliche Bitverbrauch für ein Symbol beträgt

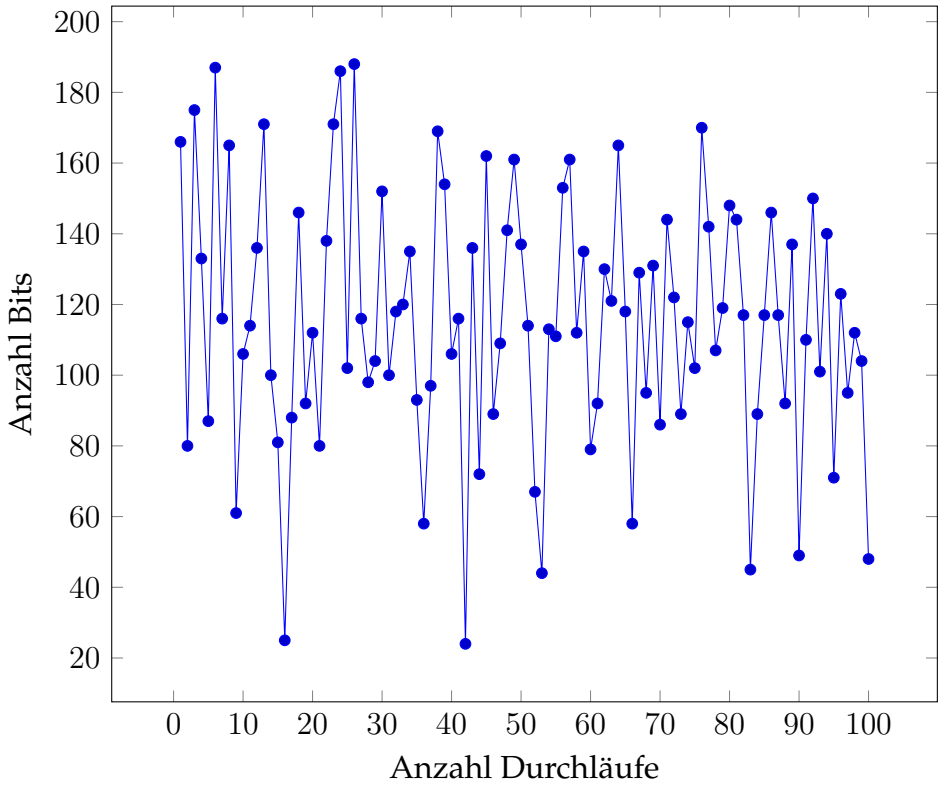
$$\frac{99}{38} \approx 2.61 \text{ Bit.}$$

2.2.2 Testläufe

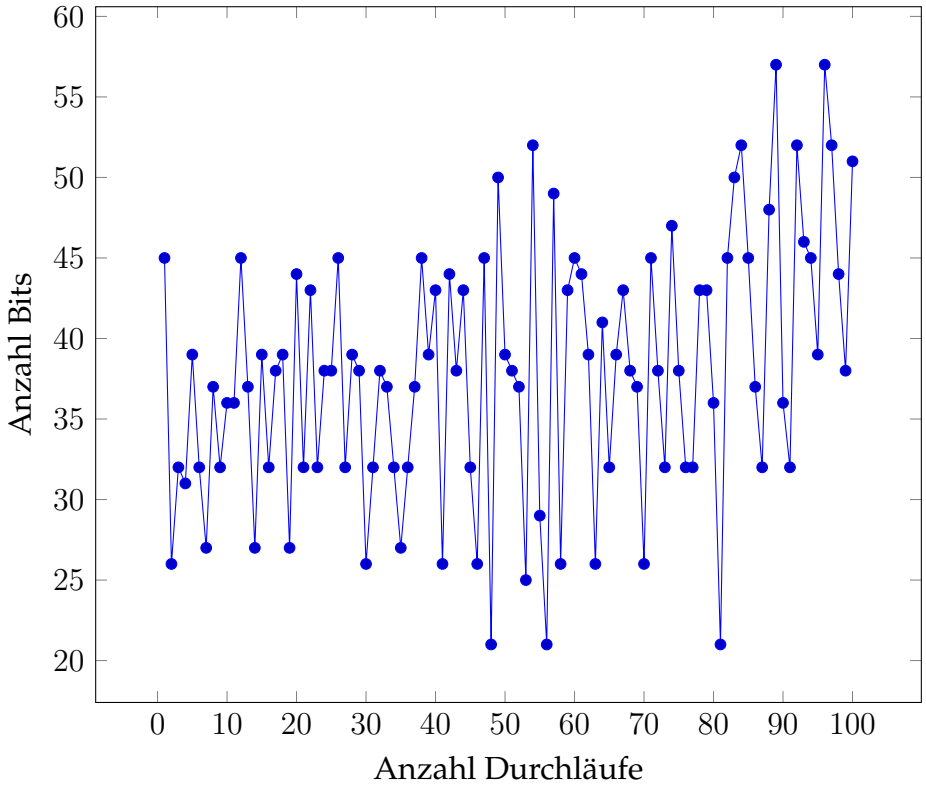
Die folgenden Grafiken wurden mit jeweils 100 zufällig generierte Sequenzpaaren mit je etwa 200 Basen, einer Fehlerrate von 15% und einem Δ-Wert von 10 berechnet.



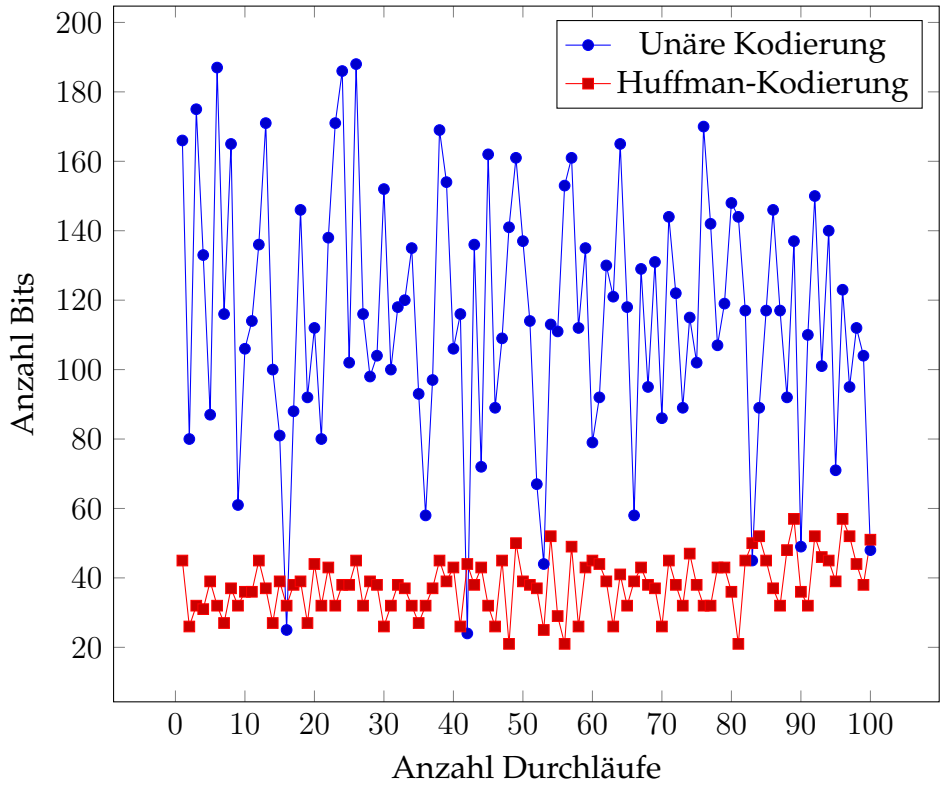
Bitverbrauch der unären binären Kodierung für CIGAR-Strings



Bitverbrauch der Huffman-Kodierung für CIGAR-Strings



Bitverbrauch von unärer und Huffman-Kodierung für CIGAR-Strings



2.3 Bewertung

am besten ist Huffman-Kodierung

Das Format benötigt wenig Speicher für Alignments mit einer kleinen Edit-Distanz und deutlich mehr Speicher für Alignments mit einer großen Edit-Distanz

3 TracePoint Konzept

Ein neuer Ansatz der speichereffizienten Repräsentation von Alignments wurde von Gene Myers in [Myers 2015] beschrieben und basiert auf dem Konzept der Trace Points.

Sei A ein Alignment von $u[i...j]$ und $v[k...\ell]$ mit $i < j$ und $k < l$ und sei $\Delta \in \mathbb{N}$. Sei $p = \lceil \frac{i}{\Delta} \rceil$. Man unterteilt $u[i...j]$ in $\tau = \lceil \frac{j}{\Delta} \rceil - \lfloor \frac{i}{\Delta} \rfloor$ Substrings $u_0, u_1, ..., u_{\tau-1}$ mit

$$u_q = \begin{cases} u[i...p \cdot \Delta] & \text{falls } q = 0 \\ u[(p+q-1) \cdot \Delta + 1...(p+q) \cdot \Delta] & \text{falls } 0 < q < \tau - 1 \\ u[(p+\tau-2) \cdot \Delta...j] & \text{falls } q = \tau - 1 \end{cases}$$

Für alle q mit $0 \leq q < \tau - 1$ sei t_q der letzte Index des Substrings von v, der in A mit u_q aligniert. t_q nennt man Trace Point. Für $q = 0$ aligniert u_0 mit $v_0 = v[k...t_0]$. Für alle q mit $0 < q < \tau - 1$ aligniert u_q mit $v_q = v[t_{q-1} + 1...t_q]$.

Seien i, j, k, ℓ, Δ und die Trace-Points eines Alignments von u und v gegeben. Dann kann ein Alignment A' von u und v mit $\delta(A') \leq \delta(A)$ konstruiert werden. Danach bestimmt man aus den Trace-Points die Substring-Paare u_q und v_q, berechnet hierfür ein optimales Alignment und konkateniert die Alignments von den aufeinanderfolgenden Substring-Paaren zu A'.

Beispiel 2:

```
Sequenz 1: gagcatgttgcctggtcctttgctaggtactgtagaga
Sequenz 2: gaccaagtaggcgtggaccttgctcggtctgtaagaga
Delta: 15

Gesamtalignment:

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

```
seq1[0...14] aligniert mit seq2[0...15]
gagc-a-t-gttgcc-tgg
||  |  |  |  |   |  |  |||
gaccaagtag--g-cgtgg

seq1[15...29] aligniert mit seq2[16...28]
tcctttgctaggtac
 ||||  |||  |||  |
acctt-gctcggt-c

seq1[30...37] aligniert mit seq2[29...37]
tgta-gaga
||||  ||||
tgtaagaga

Trace Points: [15, 28]
```

3.1 Komplexität

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

3.2 Speicherverbrauch

3.2.1 Delta-Kodierung

3.2.2 Beispiel

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 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 Points ist somit eine Delta-Kodierung möglich. Als neue Liste zu kodierender Werte ergibt sich somit nach 3.2.1 $L = (5, 5, 5, 4, 4, 6)$.

Bei der naiven binären Kodierung

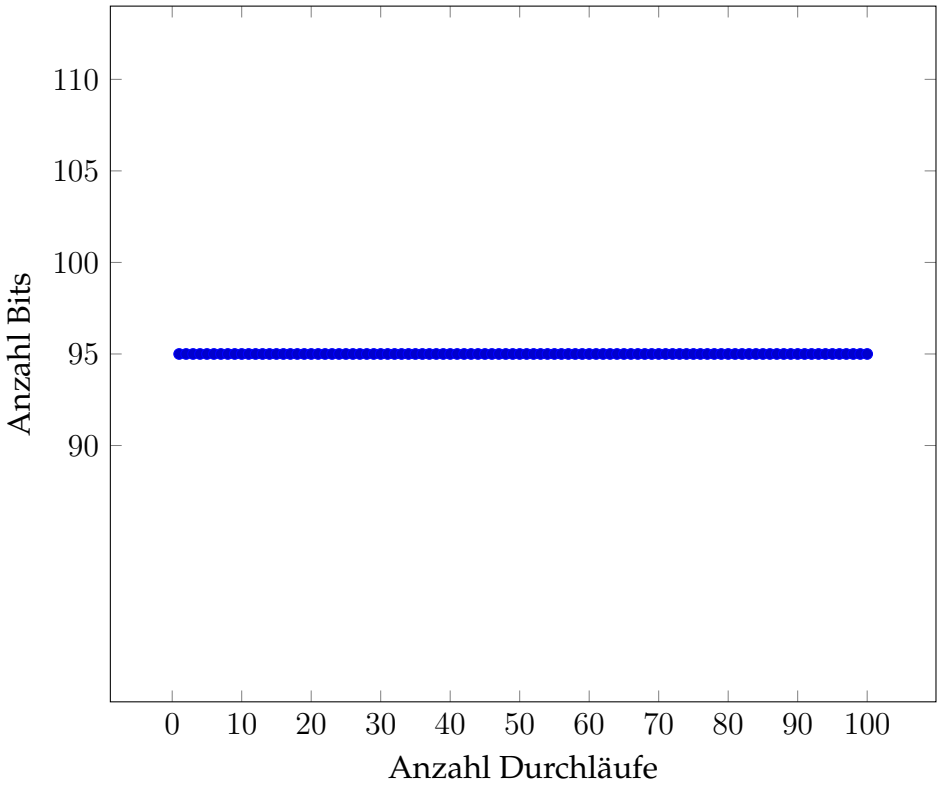
Für die unäre Kodierung

Die Huffman-Kodierung ergibt

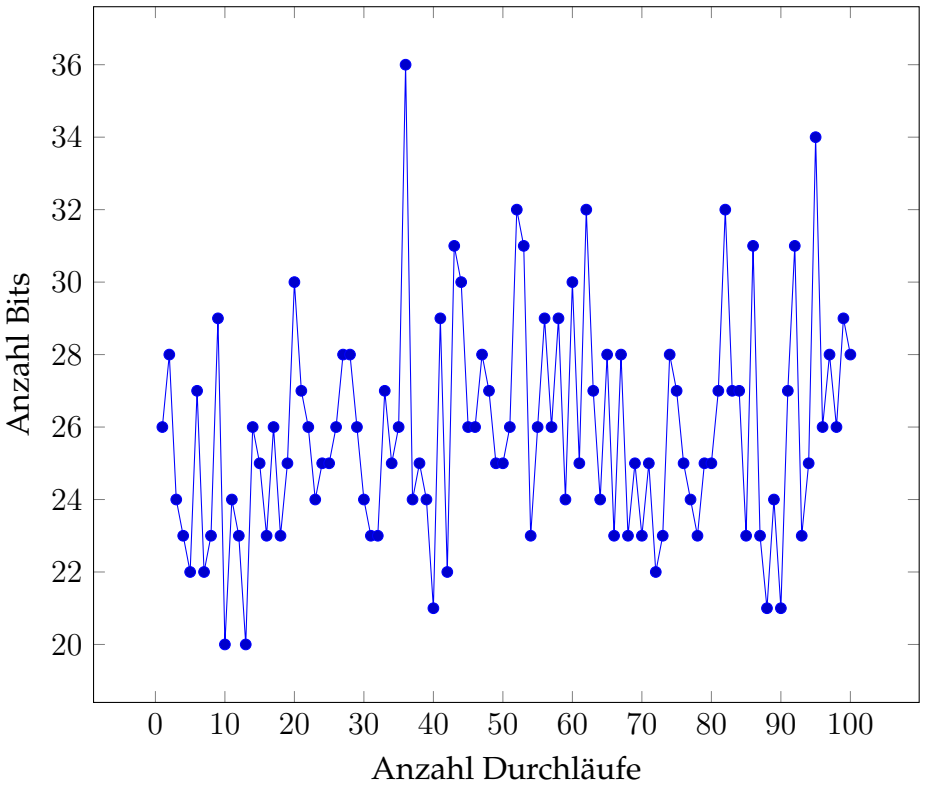
3.2.3 Testläufe

Die folgenden Grafiken wurden mit jeweils 100 zufällig generierte Sequenzpaare mit je 200 Basen, einer Fehlerrate von 15% und einem Δ-Wert von 10 berechnet.

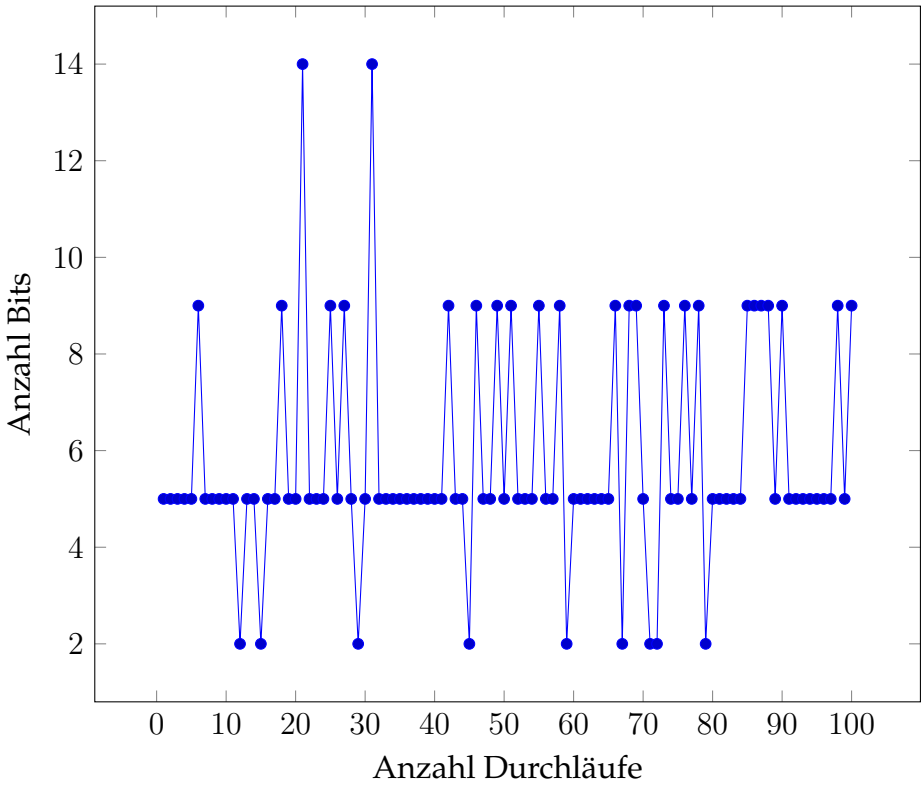
Bitverbrauch der naiven binären Kodierung für die Delta-Kodierung



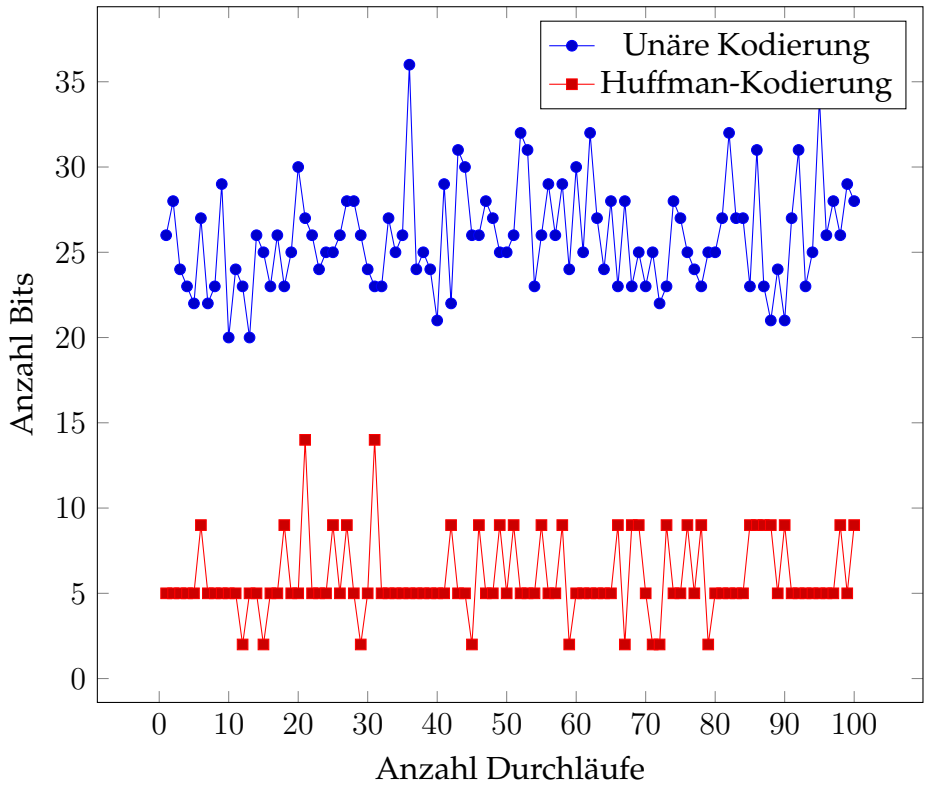
Bitverbrauch der unären binären Kodierung für die Delta-Kodierung



Bitverbrauch der Huffman-Kodierung für die Delta-Kodierung



Bitverbrauch von unärer und Huffman-Kodierung für die Delta-Kodierung



3.3 Bewertung

Die Repräsentation des Alignments A benötigt als CIGAR-String mit einer naiven binären Kodierung 228 Bit und als unäre Kodierung XX Bit, wobei die Kodierung der Trace Points mit $\Delta = 5$ mit einer naiven binären Kodierung 21 Bit und als unäre Kodierung XX Bit benötigt.

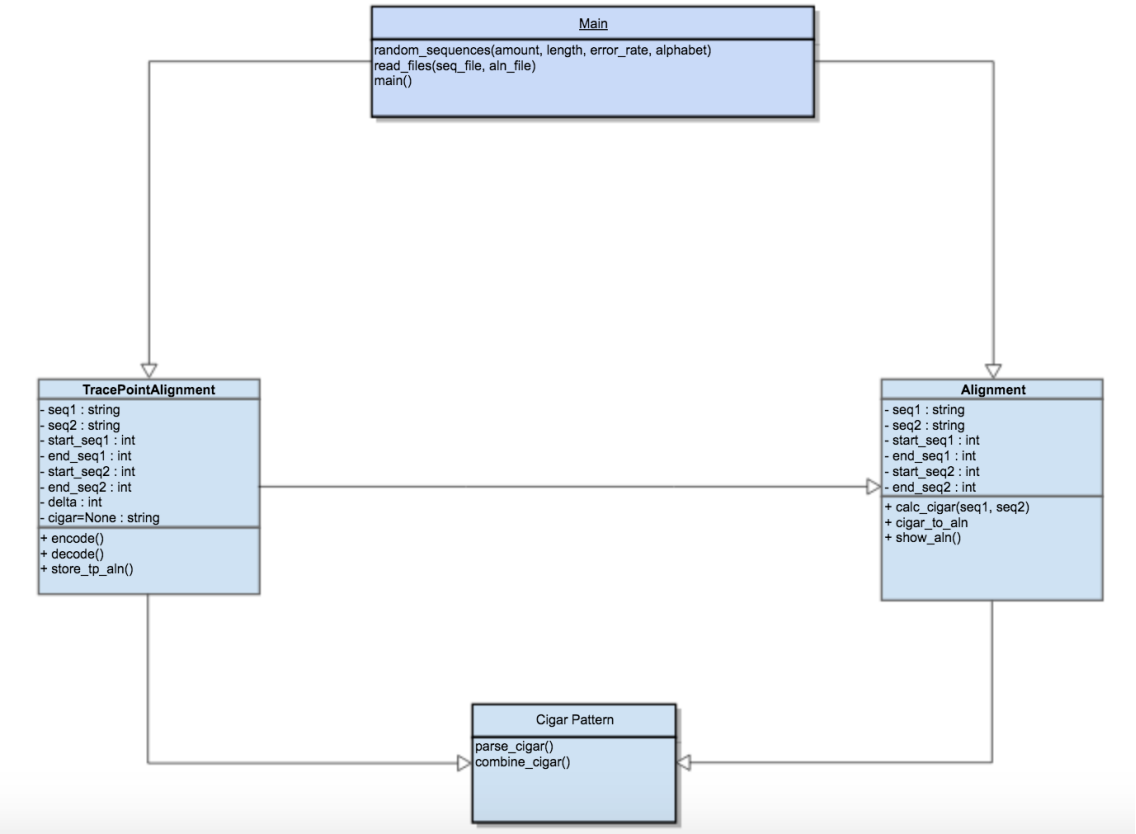
Es ist somit zu erkennen, dass die Kodierung der Trace Points in diesem Fall mit einem relativ klein gewählten Δ weniger Speicher benötigt, als die Kodierung des CIGAR-Strings. Dennoch hängt der Speicherverbrauch der Trace Points Kodierung von der Wahl des Δ ab, da bei einem kleinen Δ mehr Trace Points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

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.

4 Programm

4.1 Aufbau



4.2 Funktionalität

4.2.1 Informationsverlust bei 'encode()'

Die encode-Funktion extrahiert aus dem gegebenen CIGAR-String die Trace Points, welche dann zusammen mit dem Δ-Wert und den Start- und Endpositionen der Sequenzabschnitte gespeichert werden. Hierbei geht die Information, wie die jeweiligen Intervalle zwischen den Trace Points zu den komplementären Intervallen in der Ursprungssequenz aligniert werden, verloren. Für die Rückgewinnung dieser Information muss in der 'decode()'-Funktion zunächst ein neues

Alignment der jeweiligen Intervall-Paare errechnet werden.

Algorithm 1 Computation of Trace Points from a given CIGAR-String

Input: $seq1, seq2, start\_seq1, end\_seq1, start\_seq2, \Delta, cigar$ mit
$|seq1|, |seq2|, |cigar| > 0$;
$start\_seq1, start\_seq2 \geq 0$;
$start\_seq1 < end\_seq1$ und
$\Delta > 0$

Output: Array TP of Trace Points

function *encode*($seq1, seq2, start\_seq1, end\_seq1, start\_seq2, \Delta, cigar$)
  $itv\_size \leftarrow MAX(1, \lceil start\_seq1/\Delta \rceil)$
  $itv\_count \leftarrow MIN(\lceil |seq1|/\Delta \rceil, \lceil |seq2|/\Delta \rceil)$
  for $i \leftarrow 0$ **upto** $|itv\_count|$ **do**

$$itv[i] \leftarrow \begin{cases} start\_seq1, itv\_size \cdot \Delta - 1 & \text{if } i = 0 \\ (itv\_size + i - 1) \cdot \Delta, (itv\_size + i) \cdot \Delta - 1 & \text{if } 0 < i < |itv\_count| \\ (itv\_size + i - 1) \cdot \Delta, end\_seq1 - 1 & \text{else.} \end{cases}$$

  end for
  $count1, count2, count3 \leftarrow 0$
  $TP \leftarrow$ Array for Trace Points
  for each (cig_count, cig_symbol) in cigar **do**
    for $i \leftarrow 0$ **upto** cig_count **do**
      if cig_symbol = 'I' **then**
        increment $count1$
      else if cig_symbol = 'D' **then**
        increment $count2$
      else
        increment $count1, count2$
      end if
      if count1 = intervals[count3][1] + 1 **and** count1 $\neq |seq1|$ **then**
        append (count2 - 1 + start_seq2) to TP
      end if
      if count $\neq |itv| - 1$ **then**
        increment $count3$
      end if
    end for
  end for
  return TP
end function

Algorithm 2 Computation of a CIGAR-String from a given Trace Point Array

Input: $seq1, seq2, \Delta, TP$ mit
$|seq1|, |seq2|, \Delta, |TP| > 0$

Output: CIGAR-String

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

5 Fazit

Literaturverzeichnis

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

[Moffat u. Turpin 2002] MOFFAT, Alistair ; TURPIN, Andrew: *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002

[Myers 2015] MYERS, Eugene: *Recording Alignments with Trace Points*. `https://dazzlerblog.wordpress.com/2015/11/05/trace-points/`. Version: November 2015

[The SAM/BAM Format Specification Group 2015] THE SAM/BAM FORMAT SPECIFICATION GROUP: *Sequence Alignment/Map Format Specification*. `https://samtools.github.io/hts-specs/SAMv1.pdf`. Version: November 2015

Eidesstattliche Erklärung

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

Hamburg, den _____ Unterschrift: _____