INSTITUT FÜR INFORMATIK
Algorithmische Bioinformatik

Universitätsstr. 1 D–40225 Düsseldorf

Heinrich Heine
Universität
Düsseldorf

Das Homo-Edit-Distanz-Problem

Maren Brand

Bachelorarbeit

Beginn der Arbeit:	06. Dezember 2019
Abgabe der Arbeit:	11. März 2020
Gutachter:	Prof. Dr. Gunnar W. Klau
	Dr. Alexander Dilthey

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 11. März 2020

Maren Brand

Zusammenfassung

Das Homo-Edit-Distanz-Problem besteht darin, die Homo-Edit-Distanz zu finden, die sich aus dem Vergleich zweier Zeichenketten ergibt. Sie steht für die minimale Anzahl an notwendigen Löschungen bzw. Einfügungen in beiden Zeichenketten, um auf eine gemeinsame Teilsequenz zu kommen. Eine Löschung bzw. Einfügung kann dabei aus einer beliebigen Anzahl des selben Zeichens bestehen.
In dieser Bachelor-Arbeit werden wir auf zwei Lösungsansätze, welche das Homo-Edit-Distanz-Problem lösen, genauer eingehen. Für beide Ansätze verwenden wir dabei eine neue Methode, welche für jede Teilfolge einer Zeichenfolge berechnet, wie viele Löschungen mindestens notwendig sind, um diese Teilfolge vollständig zu löschen. Die Laufzeit dieser Methode ist polynomiell.
Der erste Ansatz beruht auf der Strategie der vollständigen Enumeration und löst das Homo-Edit-Distanz-Problem in exponentieller Zeit. Dabei findet er alle gemeinsamen Teilsequenzen, mit denen sich die Homo-Edit-Distanz erreichen lässt. Der zweite Ansatz beruht auf der Strategie der dynamischen Programmierung und löst das Homo-Edit-Distanz-Problem in polynomieller Zeit. Um alle (zielführenden) gemeinsamen Teilsequenzen zu finden, wird auch hier der Aufwand exponentiell; eine kann jedoch auch in polynomieller Zeit gefunden werden.
Somit zählt das Problem zur Komplexitätsklasse P.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zielsetzung 1

1.2 Forschungsstand 1

2 Definition **1**

2.1 Eingabe-Instanz 1

2.2 Operationen 2

2.3 Scoring 3

2.4 Anwendungsbeispiel 3

3 Methoden **3**

3.1 Kostenberechnung aller Teilfolgen 4

3.1.1 Berechnung 4

3.1.2 Korrektheitsbeweis 5

3.2 Ansatz der vollständigen Enumeration 7

3.2.1 Finden gemeinsamer Teilsequenzen 8

3.2.2 Bestimmen des Scores 8

3.2.3 Verbesserungen 10

3.3 Ansatz der Dynamischen Programmierung 10

3.3.1 Scoring-Algorithmus 10

3.3.2 Verbesserungen 11

3.3.3 Bestimmen möglicher Lösungssequenzen 13

3.3.4 Komplexität 14

4 Resümee **14**

4.1 Rekapitulation 14

4.2 Ausblick 15

Literatur **16**

1 Einleitung

Das Homo-Edit-Distanz-Problem ist ein Optimierungsproblem, für welches die Homo-Edit-Distanz bestimmt werden soll. Diese entspricht dem optimalen Score, der durch ein optimales Alignment unter den erlaubten Operationen für zwei Zeichenfolgen zu erreichen ist.

1.1 Zielsetzung

Ziel dieser Bachelorarbeit ist, das Homo-Edit-Distanz-Problem zu definieren, verschiedene Wege zu präsentieren, das Problem algorithmisch zu lösen, und es einer Komplexitätsklasse zuzuordnen. Die Forschungsfrage, der wir uns stellen, lautet folglich:

> „Wie lässt sich das Homo-Edit-Distanz-Problem effizient lösen?“

1.2 Forschungsstand

Definiert wird das Problem in [JP04], von Neil C. Jones und Pavel A. Pevzner. Dort wurde allerdings außer Acht gelassen, dass nach einer Löschung Verschmelzungen in den Zeichenfolgen berücksichtigt werden müssen. Ein Beispiel für eine solche Verschmelzung wäre ATA: Nach Löschung des T verschmelzen A und A zu AA, danach könnten sie in einem Schritt gelöscht werden. Wir fanden keine weiteren Veröffentlichungen zu dem Problem und damit bleibt auch bisher ungeklärt, in welche Komplexitätsklasse sich das Problem einordnen lässt.

2 Definition

Alle folgenden Definitionen orientieren sich an [JP04].

2.1 Eingabe-Instanz

Sei im Folgenden $\sum$ ein Alphabet und $\sum^* = \bigcup_{n \in \mathbb{N}_0} \sum^n$ die Menge aller Wörter (Zeichenfolgen) über $\sum$, wobei n der Länge der Wörter entspricht. Eine Eingabeinstanz I besteht aus zwei endlichen Zeichenfolgen $v, w \in \sum^*$, die sich mittels des Alphabets $\sum = \{A, C, G, T\}$ bilden lassen. Eine leere Zeichenfolge repräsentieren wir durch λ.
Der Übersichtlichkeit halber werden wir bei Angabe einer nicht konkreten Zeichenfolge Kommas verwenden. Wir verwenden zwei Darstellungsformen:
Bei der ersten Darstellungsmöglichkeit, die wir Einzel-Darstellung nennen, dürfen gleiche Zeichen benachbart sein, aber jede Position darf nur ein Zeichen enthalten.
Seien in dieser Darstellung unsere Zeichenfolgen v und w anzugeben mit $v = v_1, v_2, \ldots, v_n$ und $w = w_1, w_2, \ldots, w_m$.
Bei der zweiten Darstellungsmöglichkeit, die wir Block-Darstellung nennen, dürfen gleiche Zeichen nicht benachbart sein, aber jede Position enthält einen sogenannten

Block, der mehrere gleiche Zeichen umfassen darf. Wie viele gleiche Zeichen ein Block enthält, ist aus den Exponenten ersichtlich. Seien im Folgenden $a_1, a_2, \ldots, a_p \in \mathbb{N}$ und $b_1, b_2, \ldots, b_q \in \mathbb{N}$.
In dieser Darstellung seien unsere Zeichenfolgen v und w anzugeben mit $v = v_1^{a_1}, v_2^{a_2}, \ldots, v_p^{a_p}$ und $w = w_1^{b_1}, w_2^{b_2}, \ldots, w_q^{b_q}$.
Je nach Situation verwenden wir die erste oder zweite Darstellung. Sei im Folgenden also $b(v_1, v_2, \ldots, v_n) = v_1^{a_1}, v_2^{a_2}, \ldots, v_p^{a_p}$ eine Funktion, die eine Zeichenfolge von ihrer Einzel-Darstellung in ihre Block Darstellung überführt und $b^{-1}(v_1^{a_1}, v_2^{a_2}, \ldots, v_p^{a_p}) = v_1, v_2, \ldots, v_n$ die dazugehörige Umkehrfunktion. Bei der Umwandlung mit b wird ein Mapping der alten Positionen gespeichert.

2.2 Operationen

Für die Definition der Homo-Insertion-Operation verwenden wir die Einzel-Darstellung und für die der Homo-Deletion-Operation die Block-Darstellung.

Homo-Insertion Im Folgenden sei $i \in \mathbb{N}$ mit $1 \leq i \leq n+1$ die Position in einer Zeichenfolge v, an der wir k-mal das Zeichen $z \in \sum$ einfügen wollen. Alle Zeichen ab Position i werden um k Positionen nach rechts verschoben.
Eine Homo-Insertion (kurz: Insertion) ist somit definiert als

$$\delta_{Insertion}(v, i, z^k) = v_1, \ldots, v_{i-1}, z_1, \ldots, z_k, v_{i+k}, \ldots, v_{n-1+k}, v_{n+k}.$$

Es wird eine beliebige Anzahl desselben Zeichens zusammenhängend an einer beliebigen Stelle in eine der Zeichenfolgen eingefügt. Es spielt dabei keine Rolle, ob an dieser Stelle bereits das entsprechende Zeichen vorhanden ist oder nicht.

Homo-Deletion Es werden k Zeichen aus einem beliebigen Block in eine der Zeichenfolgen entfernt. Soll k-mal das Zeichen aus dem Block an Position $i \in \mathbb{N}$ mit $1 \leq i \leq p$ aus v gelöscht werden, muss für diese Operation $a_i \geq k$ gelten: Gilt $a_i > k$, wird der Exponent a_i um k verringert. Gilt hingegen $a_i = k$, wird der Block an Position i komplett gelöscht und es müssen alle folgenden Blöcke um eine Stelle nach links verschoben werden.
Eine Homo-Deletion (kurz: Deletion) ist somit definiert als

$$\delta_{Deletion}(v, i, k) = \begin{cases} v_1^{a_1}, \ldots, v_i^{a_i-k}, \ldots, v_{p-1}^{a_{p-1}}, v_p^{a_p} & , a_i > k, \\ v_1^{a_1}, \ldots, v_{i-1}^{a_{i-1}}, v_{i+1-1}^{a_{i+1}}, \ldots, v_{p-1-1}^{a_{p-1}}, v_{p-1}^{a_p} & , a_i = k. \end{cases}$$

Ist $a_i = k$ zutreffend, kann es zu einer Verschmelzung der Zeichen $v_{i-1}^{a_{i-1}}$ und $v_{i+1-1}^{a_{i+1}}$ kommen. Voraussetzung dafür ist, dass $v_{i-1}^1 = v_{i+1-1}^1$ gilt. In diesem Fall muss die Block-Darstellung korrigiert werden, da es laut Definition nicht erlaubt ist, dass zwei benachbarte Blöcke gleich sind. So verändert sich $v_1^{a_1}, \ldots, v_{i-1}^{a_{i-1}}, v_{i+1-1}^{a_{i+1}}, \ldots, v_{p-1-1}^{a_{p-1}}, v_{p-1}^{a_p}$ zu $v_1^{a_1}, \ldots, v_{i-1}^{a_{i-1}+a_{i+1}}, v_{i+2-2}^{a_{i+2}}, \ldots, v_{p-1-2}^{a_{p-1}}, v_{p-2}^{a_p}$. Eine Verschmelzung wird also immer dort möglich, wo ein Block vollständig gelöscht wird, dessen benachbarte Blöcke dasselbe Zeichen enthalten.

Zu jeder Insertion gibt es eine im Hinblick auf die Distanz äquivalente Deletion. Anstatt also einen Block desselben Zeichens in v einzufügen, könnte ebenso jener Block aus w gelöscht werden. Das bedeutet zwangsläufig, dass sowohl Insertions als auch Deletions bereits in sich ausreichend für die Bestimmung der Homo-Edit-Distanz sind. Um das Problem zu vereinfachen, werden wir daher im Folgenden auf die Insertion-Operation verzichten und uns damit auf die Deletion-Operation beschränken. Es bleibt zu beachten, dass es zwar möglich ist, in gleich vielen Schritten auf eine gemeinsame Teilsequenz zu kommen, egal ob Insertions, Deletions oder beide verwendet werden, diese Teilsequenzen sich jedoch je nach verwendeten Operationen unterscheiden werden.

2.3 Scoring

Im Gegensatz zum Score bei beispielsweise einem globalen Alignment, bei dem ein höherer Wert für eine größere Ähnlichkeit zwischen zwei Zeichenfolgen steht, ist bei der Homo-Edit-Distanz ein höherer Wert ein Indikator für größere Unterschiede. Dies liegt daran, dass durch Anwenden einer Homo-Deletion statt eines Punktes ein Strafpunkt vergeben wird. Daher gilt, dass der Score einer Instanz *Score(v, w)* der Anzahl an notwendigen Operationen entspricht, um in beiden Zeichenfolgen auf eine bestimmte, gemeinsame Teilsequenz zu kommen. Die Homo-Edit-Distanz einer Instanz *Homo-Edit-Distanz(v, w)* ist der minimale Score.

2.4 Anwendungsbeispiel

Betrachten wir nun eine Instanz mit den zwei Zeichenfolgen
$v = CTGCA = C^1T^1G^1C^1A^1$ und
$w = AGAAG = A^1G^1A^2G^1$.
Die Homo-Edit-Distanz erreichen wir über folgende Operationen:
$\delta_{Deletion}(v, 1, 1) = C^1G^1C^1A^1$,
$\delta_{Deletion}(v, 1, 1) = C^2A^1$,
$\delta_{Deletion}(v, 0, 2) = A^1$,
$\delta_{Deletion}(w, 2, 2) = A^1G^2$,
$\delta_{Deletion}(w, 1, 2) = A^1$.
Damit ergibt sich die gemeinsame Teilsequenz $A = A^1$ durch insgesamt 5 Deletions.

3 Methoden

In diesem Kapitel werden Methoden für die Lösung des Problems mittels vollständiger Enumeration und dynamischer Programmierung präsentiert, und darauf aufbauend eine Einordnung des Problems in eine Komplexitätsklasse gegeben.

3.1 Kostenberechnung aller Teilfolgen

In diesem Abschnitt wird erläutert, wie sich die Anzahl notwendiger Homo-Deletions zur kompletten Löschung einer Teilfolge innerhalb einer Zeichenfolge bestimmen lässt. Diese Berechnung wird für alle Teilfolgen durchgeführt. Im Anschluss beweisen wir, dass die resultierenden Werte der Homo-Edit-Distanz zwischen Teilfolge und λ entsprechen.

3.1.1 Berechnung

Die Methode berechnet für jede Teilfolge die Anzahl an notwendigen Deletions, um sie vollständig zu löschen.
Sei im Folgenden ein (zusammenhängendes) Gap eine zu löschende Teilfolge in v mit der Startposition $i \in \mathbb{N}$ und Endposition $j \in \mathbb{N}$, $i \leq j$, sodass $Gap_{i,j}$ der Anzahl an notwendigen Homo-Deletions entspricht. Die Zeichenfolge v muss in Block-Darstellung übergeben werden.
Für diese Methode sind die Exponenten der Block-Darstellung irrelevant, da es für die Anzahl an Homo-Deletions keinen Unterschied macht, wie häufig ein Zeichen in einem Block vorkommt. Daher werden wir in diesem Abschnitt Blöcke als Zeichen bezeichnen.
Der rekursive Ausdruck lautet

$$Gap_{i,j} = \min \begin{cases} Gap_{i,j} = 1 & , i = j, \\ \min\limits_{i \leq k < j} \{Gap_{i,k} + Gap_{k+1,j}\} & , v_i \neq v_j, \\ \min\limits_{i \leq k < j} \{Gap_{i,k} + Gap_{k+1,j} - 1\} & , v_i = v_j. \end{cases}$$

Es wird also für jedes Gap jede Aufteilung auf zwei Gaps betrachtet, also $Gap_{i,i} + Gap_{i+1,j}, Gap_{i,i+1} + Gap_{i+2,j}, \ldots, Gap_{i,j-1} + Gap_{j,j}$. Ist dabei zusätzlich $v_i = v_j$ erfüllt, so wird 1 vom Wert subtrahiert. Dies liegt daran, dass die Zeichen der Positionen i und j in keine der Zweier-Aufteilungen gemeinsam auftauchen und folglich nicht verschmolzen werden können. In $Gap_{i,j}$ wird dies zum ersten Mal möglich sein, somit benötigt man eine Deletion weniger.

Es ist durch das Mapping der Positionen möglich und für die weiteren Methoden notwendig, die gewonnenen Informationen auf die Gaps in $b^{-1}(b(v)) = v$ zu übertragen. Seien i', j' in v die Positionen i, j in $b(v)$, und sei $GapComplete_{i',j'}$ das Gap von i' bis j' für v. Dann gilt $\forall i', j' \in \mathbb{N}, i' \leq j' : GapComplete_{i',j'} = Gap_{i,j}$.

Abbildung 1 zeigt anhand der Beispiel-Zeichenfolge w aus Kapitel 2.4, wie eine Gap-Matrix nach Durchlauf der Kostenberechnung aller Teilfolgen aussehen kann.

j\i	C	T	G	C	A
C	1				
T	2	1			
G	3	2	1		
C	3	3	2	1	
A	4	4	3	2	1

Abbildung 1: Berechnung der zusammenhängenden Gaps in $w = CTGCA$ durch den Algorithmus aus Kapitel 3.1.1. Das Zeichen i steht für den Beginn des Gaps und j für dessen Ende.

Am Ende ergibt sich eine Laufzeit von $\mathcal{O}(n^3)$, da es Gapgrößen von 1 bis n gibt, für die es abhängig von der Gapgröße n bis 1 viele unterschiedliche i und j gibt, für die wiederum abhängig von der Gapgröße 1 bis n viele Zweier-Aufteilungen existieren.

3.1.2 Korrektheitsbeweis

Offensichtlich gilt, dass die Homo-Edit-Distanz, die notwendig ist um eine Zeichenfolge vollständig zu löschen, entweder zunimmt oder gleich bleibt, wenn ein weiteres Zeichen der Zeichenfolge hinzugefügt wird.
Weiterhin definieren wir eine Verschmelzung als *konfliktfrei*, wenn dadurch auf keine andere Verschmelzung verzichtet werden muss. Ein Beispiel für zwei im Konflikt stehende Verschmelzungen ist $TCTC$, entweder wir löschen zuerst das erste C und verschmelzen dann die zwei T miteinander oder wir löschen zuerst das zweite T und verschmelzen die beiden C miteinander, aber wir können niemals beide Verschmelzungen durchführen.
Sei $v \in \sum^*$ eine Zeichenfolge der Länge n in Block-Darstellung und seien $x, y, z \in \sum$ drei beliebige Zeichen mit $x \neq y$, $x \neq z$, $y \neq z$, wobei es unerheblich ist, ob es sich bei ihnen tatsächlich um ein Zeichen oder um einen Block handelt, siehe Kapitel 3.1.1.
Wir werden die Korrektheit des obigen Algorithmus zur Berechnung von Gaps mittels vollständiger Induktion über $|v| = n$ zeigen.

Induktionsanfang:
Für $n = 1$ gilt:
Alle Zeichenfolgen lassen sich mit $v = x$ darstellen und es gilt Homo-Edit-Distanz($v = x, w = \lambda$) $= 1$. Der Algorithmus liefert hier ebenfalls 1, da bei $n = 1$ immer $i = j$ gilt und diese Situation dem eigens dafür definierten Fall entspricht.
Für $n = 2$ gilt:
Alle Zeichenfolgen lassen sich mit $v = xy$ darstellen und es gilt Homo-Edit-Distanz($v =$

$xy, w = \lambda) = 2$. Die einzige Möglichkeit für den Algorithmus, diesen Fall zu berechnen, wäre $Gap_{1,1} + Gap_{2,2}$, da $v_1 \neq v_2$, sodass sich der erwartete Score von 2 ergibt.
Für $n = 3$ gibt es zwei Fälle:
Fall $v = xyx$:
Für diesen Fall gilt offensichtlich Homo-Edit-Distanz$(v, w = \lambda) = 2$. Bei $v = xyx$ würde der Algorithmus $\min\{Gap_{1,1} + Gap_{2,3} - 1, Gap_{1,2} + Gap_{3,3} - 1\}$ berechnen, da $v_1 = v_3$ gilt. Somit kommt der Algorithmus auf einen Score von 2.
Fall $v = xyz$:
Für diesen Fall gilt offensichtlich Homo-Edit-Distanz$(v, w = \lambda) = 3$. Bei $v = xyz$ würde der Algorithmus $\min\{Gap_{1,1} + Gap_{2,3}, Gap_{1,2} + Gap_{3,3}\}$ berechnen, da $v_1 \neq v_3$ gilt. Somit kommt der Algorithmus auf einen Score von 3.

Induktionsschritt $(n \to n + 1)$**:**
An die Zeichenfolge v soll ein neues Zeichen c angehängt werden.

Lemma 1 *Sei $v \in \sum^*$. Für den Score, der vom Algorithmus zur Bestimmung zusammenhängender Gaps geliefert wird, gilt: Score$(v, w = \lambda) \leq$ Homo-Edit-Distanz$(v, w = \lambda)$.*

Wir werden im Folgenden zwei Fälle unterscheiden:
Fall $v_1 = c$:
Angenommen, Score$(v_1, v_2, \ldots, v_n, c) >$ Homo-Edit-Distanz$(v_1, v_2, \ldots, v_n, c)$.
$\Rightarrow \forall k \in \mathbb{N}, 1 \leq k \leq n$: Score$(v_1, \ldots, v_k)$ + Score$(v_{k+1}, \ldots, c) - 1 >$ Homo-Edit-Distanz$(v_1, v_2, \ldots, v_n, c)$
$\Rightarrow$ Es muss eine konfliktfreie Verschmelzung geben, die in keines der Gaps Gap_{v_1,v_k} bzw. $Gap_{v_{k+1},c}$ bisher möglich war. Nicht möglich bedeutet hier, dass die zwei zu verschmelzenden Zeichen sich nicht innerhalb desselben Gaps befinden. Wäre die Verschmelzung in eines der Gaps möglich gewesen, so hätten wir vorher bereits einen niedrigeren Score für jenes erhalten.
$\Rightarrow$ Die einzige Verschmelzung, die in keines der Gaps Gap_{v_1,v_k} bzw. $Gap_{v_{k+1},c}$ möglich war, ist die Verschmelzung von v_1 und c. Das wird jedoch durch die Subtraktion von 1 berücksichtigt, da es keine Situation geben kann, in der ein Konflikt das Verschmelzen von v_1 und c verhindern würde.
$\Rightarrow$ Es kommt zum Widerspruch, es folgt Lemma 1 für Fall $v_1 = c$.

Fall $v_1 \neq c$:
Angenommen, Score$(v_1, v_2, \ldots, v_n, c) >$ Homo-Edit-Distanz$(v_1, v_2, \ldots, v_n, c)$.
$\Rightarrow \forall k \in \mathbb{N}, 1 \leq k \leq n$: Score$(v_1, \ldots, v_k)$ + Score$(v_{k+1}, \ldots, c) >$ Homo-Edit-Distanz$(v_1, v_2, \ldots, v_n, c)$
$\Rightarrow$ Es muss eine konfliktfreie Verschmelzung geben, die in keines der Gaps Gap_{v_1,v_k} bzw. $Gap_{v_{k+1},c}$ bisher möglich war. Nicht möglich bedeutet hier, dass die zwei zu verschmelzenden Zeichen sich nicht innerhalb desselben Gaps befinden. Wäre die Verschmelzung in eines der Gaps möglich gewesen, so hätten wir vorher bereits einen niedrigeren Score für jenes erhalten.
$\Rightarrow$ Die einzige Verschmelzung, die in keines der Gaps Gap_{v_1,v_k} bzw. $Gap_{v_{k+1},c}$ möglich war, ist die Verschmelzung von v_1 und c. Da diese beiden Zeichen jedoch ungleich sind,

lassen sie sich nicht verschmelzen.
$\Rightarrow$ Es kommt zum Widerspruch, es folgt Lemma 1 für Fall $v_1 \neq c$.

Lemma 2 *Sei $v \in \sum^*$. Für den Score, der vom Algorithmus zur Bestimmung zusammenhängender Gaps geliefert wird, gilt: Score($v, w = \lambda$) $\geq$ Homo-Edit-Distanz($v, w = \lambda$).*

Wir werden im Folgenden zwei Fälle unterscheiden:
Fall $v_1 = c$:
Angenommen, Score($v_1, v_2, \ldots, v_n, c$) < Homo-Edit-Distanz($v_1, v_2, \ldots, v_n, c$).
$\Rightarrow \exists k \in \mathbb{N}, 1 \leq k \leq n$: Score($v_1, \ldots, v_k$) + Score($v_{k+1}, \ldots, c$) $-$ 1 $<$ Homo-Edit-Distanz($v_1, v_2, \ldots, v_n, c$)
$\Rightarrow$ Der Algorithmus findet mindestens eine Verschmelzung mehr, als möglich ist.
$\Rightarrow$ Es muss eine Verschmelzung geben, die in Konflikt mit mindestens einer anderen steht. Dieser Konflikt kann in keines der Gaps Gap_{v_1,v_k} bzw. $Gap_{v_{k+1},c}$ bisher aufgetreten sein.
$\Rightarrow$ Die einzige Verschmelzung, die in keines der Gaps Gap_{v_1,v_k} bzw. $Gap_{v_{k+1},c}$ bisher möglich war und nun hinzukommt, ist die Verschmelzung von v_1 und c. Da es sich um die Verschmelzung der Zeichen am Rand handelt, kann es jedoch zu keiner Überschneidung und folglich zu keinem Konflikt kommen.
$\Rightarrow$ Es kommt zum Widerspruch, es folgt Lemma 2 für Fall $v_1 = c$.

Fall $v_1 \neq c$:
Angenommen, Score($v_1, v_2, \ldots, v_n, c$) < Homo-Edit-Distanz($v_1, v_2, \ldots, v_n, c$).
$\Rightarrow \exists k \in \mathbb{N}, 1 \leq k \leq n$: Score($v_1, \ldots, v_k$) + Score($v_{k+1}, \ldots, c$) $<$ Homo-Edit-Distanz($v_1, v_2, \ldots, v_n, c$)
$\Rightarrow$ Der Algorithmus findet mindestens eine Verschmelzung mehr, als möglich ist.
$\Rightarrow$ Es muss eine Verschmelzung geben, die in Konflikt mit mindestens einer anderen steht. Dieser Konflikt kann in keines der Gaps Gap_{v_1,v_k} bzw. $Gap_{v_{k+1},c}$ bisher aufgetreten sein.
$\Rightarrow$ Die einzige Verschmelzung, die in keines der Gaps Gap_{v_1,v_k} bzw. $Gap_{v_{k+1},c}$ bisher möglich war, ist die Verschmelzung von v_1 und c. Da jedoch $v_1 \neq z$ gilt, kommt es zu keiner Verschmelzung und damit auch zu keinem Konflikt.
$\Rightarrow$ Es kommt zum Widerspruch, es folgt Lemma 2 für Fall $v_1 \neq c$.

Lemma 1 und Lemma 2 implizieren den folgenden Satz.

Satz 1 *Sei $v \in \sum^*$. Für den Score, der vom Algorithmus zur Bestimmung zusammenhängender Gaps geliefert wird, gilt: ($v, w = \lambda$) = Homo-Edit-Distanz($v, w = \lambda$).*

3.2 Ansatz der vollständigen Enumeration

Der erste Ansatz zur Lösung des Problems ist, den Lösungsraum vollständig zu untersuchen.

3.2.1 Finden gemeinsamer Teilsequenzen

Für jede der beiden Zeichenfolgen v und w gibt es maximal 2^n bzw. 2^m viele Teilsequenzen. Jede dieser Teilsequenzen lässt sich über einen Binärvektor darstellen, in dem jedes Bit eine Position in der Folge v bzw. w repräsentiert. Steht an einer Position eine 1 im Vektor, so bedeutet dies, dass das Zeichen an dieser Position in der Teilsequenz vorkommt. Folglich würde es bei einer 0 nicht in der Teilsequenz vorkommen.
Aus der obigen Binärkodierung entstehen also Teilsequenzen der ursprünglichen Zeichenfolge. Dabei gibt es mehr Binärkodierungen als Teilsequenzen, so gibt es beispielsweise für die Zeichenfolge $ACAC$ und die Teilsequenz AC zwei Binärkodierungen, 1100 und 0011. Den entstehenden Teilsequenzen werden daher alle Binärkodierungen, aus denen sie sich bilden lassen, zugeordnet.
Da sich nicht alle Teilsequenzen von v aus Binärkodierungen von w bilden lassen, können wir die Menge der für die Lösung in Frage kommenden Binärkodierungen auf die Schnittmenge aller Teilsequenzen von v und w reduzieren.

Wir werden im Folgenden nicht das Beispiel aus Kapitel 2.4 verwenden, da es zu viele mögliche Teilsequenzen für dieses gibt.
Betrachten wir nun eine Instanz mit den zwei Zeichenfolgen
$v = CGC$ und
$w = \lambda$.
Es ergeben sich folgende Zeichensequenzen aus den Binärkodierungen von v:
$CGC,\ CG-,\ C-C,\ -GC,\ C--,\ -G-,\ --C,\ ---$.
Nach Teilsequenzen sortiert erhalten wir:
$CGC \rightarrow \{CGC\}$,
$CG \rightarrow \{CG-\}$,
$CC \rightarrow \{C-C\}$,
$GC \rightarrow \{-GC\}$,
$C \rightarrow \{C--, --C\}$,
$G \rightarrow \{-G-\}$,
$\lambda \rightarrow \{---\}$.
Es ergibt sich folgende Zeichensequenz aus den Binärkodierungen von w:
λ.
Auf Teilsequenzen sortiert erhalten wir:
$\lambda \rightarrow \{\lambda\}$.
Damit erhalten wir als einzige gemeinsame Teilsequenz von v und w:
λ.

Am Ende ergibt sich eine Laufzeit von $\mathcal{O}(2^{n+m})$, die durch das Erstellen der Binärkodierungen und das Finden der gemeinsamen Teilsequenzen zustande kommt.

3.2.2 Bestimmen des Scores

Nachdem die möglichen Lösungen auf die Schnittmenge gemeinsamer Teilsequenzen von v und w reduziert wurden, sollen nun die Teilsequenzen und die dazugehörigen Binärkodierungen aus beiden Zeichenfolgen gefunden werden, welche die wenigsten

Deletions benötigen.
Dabei wird für jede einzelne gemeinsame Teilsequenz überprüft, wie viele Deletions mindestens nötig sind, um v und w in diese zu überführen. Da es wie bereits erwähnt mehrere Binärkodierungen geben kann, die für dieselbe Teilsequenz kodieren, wird zunächst für beide Zeichenfolgen einzeln überprüft, welche Binärkodierung für jede Teilsequenz die wenigsten Deletions benötigt. Hierzu werden alle zusammenhängenden Gaps aus der Binärkodierung aufaddiert. Die Gaps müssen zuvor einmalig je Zeichenfolge berechnet werden, siehe Kapitel 3.1.1. Ist für beide Zeichenfolgen die Binärkodierung für die aktuelle Teilsequenz mit dem kleinsten Score gefunden, so addiert man beide Kosten auf.

Die Menge S speichere die Teilsequenzen, die bisher den besten Score liefern. Erreichen wir mit einer Teilsequenz t einen günstigeren Score, so setzen wir $S = t$.
Erreichen wir mit einer Teilsequenz t denselben Score, so setzen wir $S = S \cup t$.
Erreichen wir mit einer Teilsequenz t einen schlechteren Score, als den bisher niedrigsten, so bleibt S unverändert.

Sind alle gemeinsamen Teilsequenzen abgearbeitet, so entspricht die Homo-Edit-Distanz dem gefundenen Score. Alle Teilsequenzen, die sich zu diesem Zeitpunkt in S befinden, bezeichnen wir als Lösungssequenzen.

Setzen wir nun das Beispiel aus Kapitel 3.2.1 fort. Als einzige gemeinsame Teilsequenz von v und w ergab sich dort λ. Da $w = \lambda$ ist, müssen wir nur noch berechnen, wie viele Homo-Deletions benötigt werden, um v auf λ zu bringen. Dafür werden wir den Algorithmus aus Kapitel 3.1.1 verwenden.
Da in w nichts gelöscht werden muss und in v nur ein zusammenhängendes Gap $\lambda \rightarrow \{- - -\}$ gelöscht wird, erhalten wir einen Score von 2 für diese gemeinsame Teilsequenz, siehe Abbildung 2. Somit ist die Homo-Edit-Distanz$(v, w) = 2$ und die einzige Lösungssequenz λ.

j\i	C	G	C
C	1		
G	2	1	
C	(2)	2	1

Abbildung 2: Berechnung der zusammenhängenden Gaps in $v = CGC$ durch den Algorithmus aus Kapitel 3.1.1. Das Zeichen i steht für den Beginn des Gaps und j für dessen Ende. Der eingekreiste Eintrag entspricht dem für unser Beispiel gesuchten Gap.

Insgesamt ergibt sich eine Laufzeit von $\mathcal{O}(\min\{2^n, 2^m\} \cdot (n+m+1))$, darin enthalten sind die Aufwendungen, um in jeder gemeinsamen Teilsequenz die zusammenhängenden Gaps zu finden, und das Durchgehen aller gemeinsamen Teilsequenzen zur Berechnung ihrer Scores.

Mit dem Abschluss dieses Kapitels lässt sich nun die Gesamtlaufzeit des Verfahrens der vollständigen Enumeration auf $\mathcal{O}(2^{n+m})$ beschränken.

3.2.3 Verbesserungen

Während der Suche nach einem schnelleren Algorithmus, der das Homo-Edit-Distanz-Problem löst, ergab sich eine verbesserte Variante der vollständigen Enumeration.
So ist es möglich, den Schritt, in dem alle Binärkodierungen für beide Zeichenfolgen erstellt werden, zu überspringen und stattdessen von Anfang an nur Binärkodierungen zu bilden bzw. aufzunehmen, die für Teilsequenzen kodieren, die sich aus Binärkodierungen beider Zeichenfolgen bilden lassen. Alle Binärkodierungen, auf die diese Beschreibung zutrifft, lassen sich aus allen optimalen globalen Alignments bilden, die bei Anwendung des Needleman-Wunsch-Algorithmus entstehen mit den folgenden Werten: Indel-Kosten $= 0$, Match-Prämie $= 0$ und Mismatch-Kosten $= \infty$.
Diese Verbesserung erwies sich in der Praxis als deutlich effizienter, da nur für die Teilsequenzen Laufzeitkosten entstehen, die auch tatsächlich zu den Lösungssequenzen gehören könnten.

3.3 Ansatz der Dynamischen Programmierung

Der zweite Ansatz ist, das Problem mittels dynamischer Programmierung (DP) zu lösen. In den folgenden Abschnitten stellen wir einen rekursiven Algorithmus vor, mit dem die DP-Matrix gefüllt wird. Außerdem präsentieren wir weitere mögliche Verbesserungen für diesen Algorithmus, schätzen seine Laufzeit ab und zeigen mögliche Wege auf, mit denen sich die Lösungssequenzen einer Instanz finden lassen. Zum Schluss gehen wir genauer auf die Komplexität des Problems ein.

3.3.1 Scoring-Algorithmus

Die Idee ist, in einer Matrix mit den Dimensionen $n+1$ und $m+1$ in jedem Eintrag $M_{y,x}$ zu speichern, was die Homo-Edit-Distanz der Teilfolgen 0 bis $y-1$ in v und 0 bis $x-1$ in w beträgt. So ist für jeden Eintrag $M_{y,x}$ bekannt, dass alle bis dahin verwendeten Zeichen in beiden Zeichenfolgen v und w mittels $M_{y,x}$ vielen Operationen auf eine gemeinsame Teilsequenz gebracht werden können.
Für jeden Eintrag $M_{y+1,x+1}$ gibt es dabei $y+x+1$ viele Möglichkeiten, auf den günstigsten lokalen Score zu kommen.
Die ersten y Möglichkeiten entstehen durch die Option, von jedem Eintrag $M_{i,x}$ darüber, $0 \leq i \leq y-1$, den lokalen Wert zu nehmen und dann die neu entstehenden Kosten für ein Gap in v zwischen den Positionen i und $y-1$, also $GapV_{i,y-1}$, aufzuaddieren.
Die zweiten x Möglichkeiten entstehen durch die Option, von jedem Eintrag $M_{y,j}$ links, $0 \leq j \leq x-1$, den lokalen Wert zu nehmen und dann die neu entstehenden Kosten für ein Gap in w zwischen den Positionen j und $x-1$, also $GapW_{j,x-1}$, aufzuaddieren.
Die letzte, zusätzliche Möglichkeit wäre, den Wert aus dem Eintrag $M_{y-1,x-1}$ zu übernehmen, was allerdings nur dann möglich ist, wenn das Zeichen in v an Position $y-1$ dasselbe ist wie das Zeichen in w an Position $x-1$.

Der rekursive Ausdruck lautet

$$M_{y,x} = \min \begin{cases} 0 & , y = 0 \text{ und } x = 0, \\ M_{y-1,x-1} & , v_{y-1} = w_{x-1}, \\ \min\limits_{0 \le i < y} \{M_{i,x} + GapV_{i,y-1}\} & , \\ \min\limits_{0 \le j < x} \{M_{y,j} + GapW_{j,x-1}\} & . \end{cases}$$

Am Ende befindet sich die Homo-Edit-Distanz in dem Eintrag $M_{n,m}$.

Abbildung 3 zeigt anhand des Beispiels aus Kapitel 2.4, wie eine Matrix nach Durchlauf des Algorithmus aussehen kann. Es wird deutlich, dass die einzelnen Einträge ihre besten Scores über mehrere Vorgänger-Einträge erreichen können, die nicht zwangsläufig zu ihren direkten Nachbarn zählen müssen.

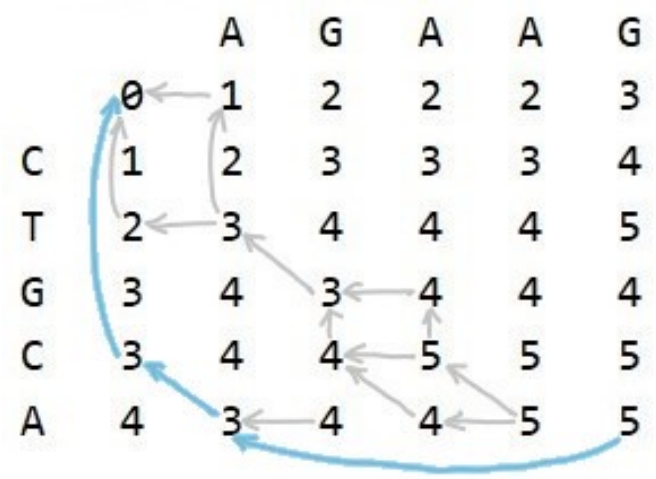

Abbildung 3: Matrix nach Anwendung des Scoring-Algorithmus auf die Beispiel-Zeichenfolgen $v = CTGCA$ und $w = AGAAG$. Die türkisen Pfeile zeigen den einzigen möglichen Weg, um auf den Score in $M_{n,m}$ zu kommen. Die grauen Pfeile zeigen Wege, die zwischenzeitlich auf einen besseren Score kommen. Es gilt zu beachten, dass der Übersichtlichkeit halber nur die wichtigsten Wege in Form von Pfeilen eingezeichnet wurden.

Es ergibt sich eine Laufzeit von $\mathcal{O}(n \cdot m \cdot (n + m + 1))$, da es $n \cdot m$ viele Einträge in der Matrix gibt, für die es jeweils bis zu $n + m + 1$ viele Möglichkeiten gibt.

3.3.2 Verbesserungen

Bestimmen einer oberen Schranke Sei im Folgenden ein möglicher vertikaler Starteintrag ein Eintrag, der sich in derselben Spalte oberhalb von dem Eintrag befindet, auf den er sich bezieht. Sei dementsprechend ein möglicher horizontaler Starteintrag ein Eintrag, der sich in derselben Zeile links von dem Eintrag befindet, auf den er sich bezieht. Wir

nennen sie deswegen Starteinträge, weil sie für den Start eines zusammenhängenden Gaps in v (vertikaler Starteintrag) bzw. w (horizontaler Starteintrag) in Frage kommen. In dieser Variante speichert jeder Eintrag zusätzlich zu seinem besten lokalen Score eine Liste vertikaler Starteinträge und eine Liste horizontaler Starteinträge, mit denen man diesen Score erreichen kann.
Statt für jeden Eintrag $M_{y+1,x+1}$ alle Einträge darüber und links daneben zu betrachten, werden wir in dieser Variante nur die Scores für die in den angrenzenden Einträgen bereits gefundenen Starteinträge berechnen. Wir betrachten also die vertikalen Starteinträge aus $M_{y+1,x}$, die horizontalen Starteinträge aus $M_{y,x+1}$ und addieren jeweils die Gap-Kosten von diesen Starteinträgen bis zu dem aktuellen Eintrag. Zusätzlich kommen die Einträge $M_{y+1,x}$, $M_{y,x+1}$ und $M_{y,x}$ als neue Starteinträge für $M_{y+1,x+1}$ in Frage, für letzteren Starteintrag muss allerdings $v_{y-1} = w_{x-1}$ gelten. Nachdem für jeden möglichen Startwert der Score für den aktuellen Eintrag berechnet ist, werden die Starteinträge, mit denen man den niedrigsten Score erreichen kann, in den entsprechenden Listen gespeichert.
Der Wert, der am Ende in $M_{n,m}$ gespeichert ist, muss nicht zwangsläufig der Homo-Edit-Distanz entsprechen. Das liegt daran, dass einige Starteinträge, die sich im späteren Verlauf als günstig(er) erweisen, lokal schlechtere Werte liefern können. Daraus resultierend werden sie in dieser Variante verworfen, weil immer nur die lokal besten Starteinträge gespeichert werden. Wir wissen nun aber, dass die Homo-Edit-Distanz nicht größer sein kann als $M_{n,m}$, damit ist eine obere Schranke gefunden.

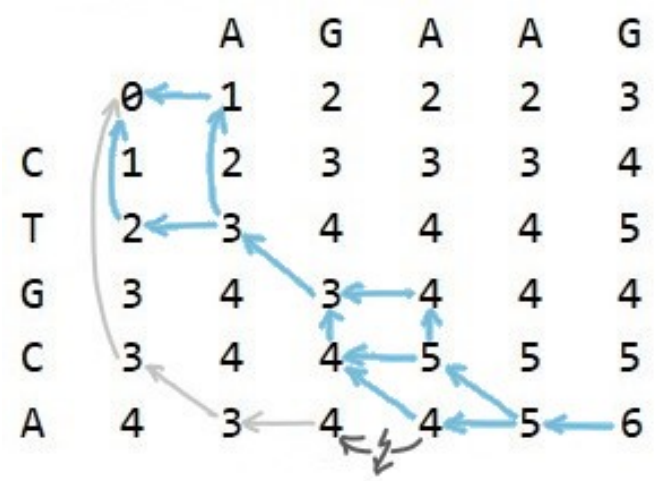

Abbildung 4: Matrix nach Anwendung des Algorithmus zum Bestimmen einer oberen Schranke auf die Beispiel-Zeichenfolgen $v = CTGCA$ und $w = AGAAG$. Die türkisen Pfeile stehen für die möglichen Wege, um auf den Score in $M_{n,m}$ zu kommen. Die grauen Pfeile zeigen den Weg, der eigentlich für $M_{n,m}$ zum besten Score führen würde, jedoch geht dieser beim dunkelgrau eingezeichneten Pfeil verloren.

Abbildung 4 zeigt anhand des Beispiels aus Kapitel 2.4, wie die Matrix zur Bestimmung der oberen Schranke aussieht. Sie verdeutlicht, warum dieser Schritt allein nicht ausreicht, um die Homo-Edit-Distanz zu bestimmen. Da immer nur die lokal besten Starteinträge in den Listen gespeichert werden, geht der optimale Weg der Instanz verloren. Dies liegt daran, dass er zwischenzeitlich einen lokal schlechteren Score liefert.

Prüfen auf günstigere Fälle Nachdem wir nun eine obere Schranke bestimmt haben, wiederholen wir obiges Vorgehen mit dem Unterschied, dass in den horizontalen/vertikalen Listen nicht nur die Starteinträge gespeichert werden, die den besten lokalen Score liefern, sondern auch all jene, die einen Score liefern, der kleiner gleich der oberen Schranke ist. Soll es später möglich sein, alle Lösungssequenzen zu finden, mit denen man die Homo-Edit-Distanz erreichen kann, müssen nach der Berechnung der Matrix alle Listen der Einträge aktualisiert werden. Dies geschieht, indem nur jene Starteinträge in den Listen gespeichert bleiben, die auch den besten Score für den entsprechenden Eintrag in der Matrix liefern.
Die resultierende Matrix für das Beispiel aus Kapitel 2.4 entspricht Abbildung 3.

3.3.3 Bestimmen möglicher Lösungssequenzen

In diesem Kapitel beziehen wir uns ausschließlich auf den Scoring-Algorithmus aus Kapitel 3.3.1 Auf diesem aufbauend werden wir beschreiben, welche Möglichkeiten wir haben, um alle Lösungssequenzen einer Instanz zu finden.

Backtracking Für diese Möglichkeit ist es wichtig, sich für jeden Eintrag der Matrix zu speichern, über welche Wege bzw. von welchen Einträgen man den minimalen Score erreichen konnte. Dies kann man am besten mit einer Liste lösen, in der man alle Positionen derjenigen Einträge, auf die die obige Beschreibung zutrifft, abspeichert, zum Beispiel als Tupel.
Danach startet man das Backtracking in der Liste des Eintrags $M_{n,m}$. Da alle Listenelemente wiederum Einträge in der Matrix repräsentieren, springt der Algorithmus zu allen Positionen aus der Liste und macht dann dasselbe für deren Listen. Nur wenn von einem beliebigen Eintrag der Matrix $M_{y+1,x+1}$ zu seinem diagonalen, direkten Nachbarn $M_{y,x}$ gesprungen wird, fügen wir der bis dahin gefundenen Teilsequenz das Zeichen $v_{y-1}(= w_{x-1})$ hinzu. So gelangt man an alle Lösungssequenzen. Es bleibt jedoch zu beachten, dass es sich nur um alle Lösungssequenzen handelt, die sich aus den beiden Zeichenfolgen v und w per Homo-Deletions bilden lassen.
Dieses Verfahren hat eine exponentielle Laufzeit.

Alternative zum Backtracking Beim Backtracking haben wir für jeden Eintrag der Matrix alle Einträge gespeichert, über welche wir den minimalen Score erreichen konnten. Nun werden wir für jeden Eintrag der Matrix alle bis dahin möglichen Teilsequenzen speichern, mit denen wir auf den lokal niedrigsten Score kommen können. Dafür ist es jedoch immer noch relevant, über welche Einträge wir diesen Score erreichen konnten.

Konnten wir den Score beispielsweise über Einträge links oder oberhalb unseres aktuellen Eintrags erreichen, so müssen nur die für diese Einträge zuvor bestimmten Teilsequenzen kopiert werden, denn aus diesen Richtungen kommend werden nur Gaps erweitert. Lediglich wenn der Score auch über den Nachbareintrag oben links erreicht wurde, wird dessen Liste mit Teilsequenzen kopiert, wobei an jeden Eintrag dieser Liste das an dieser Stelle matchende Zeichen angehängt wird. Man sollte jedoch bei diesem Vorgehen sicherstellen, dass keine Teilsequenzen mehrfach gespeichert werden, da dies nur die Laufzeit erhöhen würde. Wendet man dieses Verfahren bei allen Einträgen der Matrix an, so befinden sich am Ende alle Lösungssequenzen in der Liste des Eintrags $M_{n,m}$.
Dieses Verfahren hat ebenfalls eine exponentielle Laufzeit, allerdings werden Mehrfachvorkommen von Teilsequenzen für jeden Eintrag vermieden. Bei einem Alphabet, welches aus 4 Zeichen besteht, sind Mehrfachvorkommen ab einer Zeichenfolgenlänge von 5 unumgänglich, da man nur auf 2^5 distinkte Teilsequenzen kommen kann, wenn es kein Zeichen in der Zeichenfolge gibt, welches mehr als einmal vorkommt. Mit steigender Zeichenfolgenlänge steigt somit auch der Unterschied in den Laufzeiten der zwei Varianten.

3.3.4 Komplexität

Um das Problem einer Komplexitätsklasse zuzuordnen, beziehen wir uns im Folgenden auf den schnelleren Ansatz der dynamischen Programmierung aus Kapitel 3.3. Da aus dem Kapitel 3.3.1 bekannt ist, dass die Homo-Edit-Distanz mit einer Laufzeit von $\mathcal{O}(n \cdot m \cdot (n+m+1))$ berechnet werden kann und dies offensichtlich ein polynomieller Ausdruck ist, können wir das Homo-Edit-Distanz-Problem der Komplexitätsklasse P zuordnen.

4 Resümee

Dieses Kapitel beinhaltet eine Kurzfassung aller gewonnenen Erkenntnisse und soll einen Ausblick zu weiteren Forschungsmöglichkeiten im Bereich des Homo-Edit-Distanz-Problems geben.

4.1 Rekapitulation

Ziel war es, eine Antwort auf die Forschungsfrage zu finden, wie sich das Homo-Edit-Distanz-Problem effizient lösen lässt. Diesem Ziel sind wir im Verlaufe der Arbeit Stück für Stück näher gekommen.
Zu Beginn haben wir das Problem definiert, danach haben wir begonnen, das Problem praktisch zu lösen. Hauptbestandteil der praktischen Lösung ist eine Methode, die für alle Teilfolgen einer Zeichenfolge berechnet, wie viele Homo-Deletions notwendig sind, um diese vollständig zu löschen. Diese Methode, deren Korrektheit wir bewiesen haben, wird in beiden darauffolgenden Lösungsansätzen verwendet. Die Erste basiert auf vollständiger Enumeration und die Zweite auf dynamischer Programmierung. Mit dem ersten Ansatz ließ sich eine Lösung des Problems in exponentieller Zeit finden. Es stellte

sich jedoch heraus, dass es mit dem zweiten Ansatz möglich ist, das Problem in polynomieller Zeit zu lösen. Dadurch konnten wir wiederum schlussfolgern, dass das Problem der Komplexitätsklasse P zuzuordnen ist.

4.2 Ausblick

Nachdem wir eine polynomielle Lösung des Homo-Edit-Distanz-Problems für ein beschränktes Alphabet geliefert haben, welche sich auf jedes beliebige andere beschränkte Alphabet übertragen ließe, liegt der Fokus weiterer Forschungen weniger auf der Eingabe-Instanz, als viel eher auf den möglichen Modifikationen der Operationen. So könnte man an Stelle einzelner Zeichen Repeats als Zeichen aus dem Alphabet betrachten und Operationen definieren, bei denen gleiche Repeats zusammen gelöscht werden. Im Besonderen bleibt also die Frage, auf welche Stringprobleme sich die Operationen übertragen lassen, unbeantwortet. Des Weiteren beschränken sich die Lösungssequenzen auf die Teilsequenzen, die sich mittels Homo-Deletions bilden lassen. Daher bleibt es offen, wie man auf Lösungssequenzen kommt, die sich nur durch Homo-Insertions oder einer Kombination aus beiden Operationen bilden lassen.

Abschließend lässt sich sagen, dass wir die Forschungsfrage durch den gefundenen polynomiellen Algorithmus beantworten konnten, sodass eine Grundlage für weitere Forschungsmöglichkeiten in diesem Gebiet geboten ist.

Literatur

[JP04] JONES, Neil C. ; PEVZNER, Pavel A.: *An Introduction to Bioinformatics Algorithms*. MIT Press, 2004