# Kürzeste-Wege-Algorithmen für Sequence Alignment

Bachelorarbeit in Mathematik
eingereicht an der Fakultät für Mathematik und Informatik
der Georg-August-Universität Göttingen
am 1. Februar 2013

von

Philipp Seemann
Matrikelnummer: 20979060

Erstgutachter:
Jun.-Prof. Dr. Stephan Westphal

Zweitgutachterin:
Dr. Marie Schmidt

# Inhaltsverzeichnis

# 1 Einführung

Überfall auf eine Bank, die Täter sind geflüchtet und die Überwachungsanlage liefert keine brauchbaren Bilder. Die Spurensicherung rückt an, findet keine Fingerabdrücke, aber genetisches Material, das mit hoher Wahrscheinlichkeit von den Tätern stammen muss. Darin enthalten ist die DNA, der sogenannte genetische Fingerabdruck. Die Experten werten diese DNA jetzt aus und vergleichen sie mit den Einträgen von bekannten Kriminellen in der Datenbank. Aber wie kann man solche Basensequenzen effektiv vergleichen?

Biologen entdecken ein bisher unbekanntes Protein. Ist dies eine Mutation eines bekannten Proteins, gibt es sonst eine Ähnlichkeit, die auf ähnliche Funktionen hinweisen kann? Dafür müssen die Aminosequenzen miteinander verglichen werden.

Diese beiden Beispiele, aber auch Vaterschaftstests und Genvergleiche führen in der Bioinformatik alle auf das selbe Problem: Wir haben zwei oder mehrere Sequenzen, für die wir feststellen möchten, wie ähnlich sich diese sind. Dazu müssen wir zunächst festlegen, was im Bezug auf diese Sequenzen „Ähnlichkeit“ überhaupt bedeutet und dann eine optimale Anordnung finden. Was das genau bedeutet, wollen wir uns im Folgenden anschauen. In dieser Arbeit wird dabei nur der Fall von zwei zu vergleichenden Sequenzen betrachtet, für mehr Sequenzen benötigt man dann andere Verfahren zur effektiven Lösung. Desweiteren lösen wir uns von der biologischen Interpretation und betrachten die Problemstellung und die vorgestellten Verfahren möglichst allgemein.

Dabei werden wir zunächst das Problem exakt definieren, und dann die Lösungsidee von Needleman und Wunsch [15] (1969) erläutern, welche das Finden von möglichst gut zugeordneten Sequenzen auf ein Kürzeste-Wege-Problem zurückführt. Danach beschreiben wir die für das hier gewählte Verfahren grundlegende Idee der dynamischen Programmierung und, als eine Anwendung davon, den Dijkstra-Algorithmus zum Bestimmen von kürzesten Wegen. Als neuen Ansatz für dieses Problem wollen wir uns dann den mit dem Dijkstra verwandten A*-Algorithmus anschauen. Dieser benötigt allerdings eine Heuristik, das heißt wir müssen zusätzliche Informationen

über das Problem bereitstellen. Wir überlegen uns, wie wir diese aus der Konstruktion von Needleman und Wunsch erhalten können. Danach wird diese Idee dann praktisch getestet, indem wir den Algorithmus implementieren und auf Beispielsequenzen anwenden. Dabei treten dann noch ein paar weitere Fragestellungen auf, z.B. in Hinsicht auf die effiziente Minimum-Suche während des Algorithmus.

## 1.1 Literaturübersicht

Eine gute Übersicht über das Pairwise Sequence Alignment sowie den Lösungsansatz mit dynamischer Programmierung findet man in dem Paper [9]. Von der dynamischen Programmierung und ihrer Geschichte handelt der Artikel [7]. Zum Dijkstra-Algorithmus findet man das ursprüngliche Paper [6] sowie viele Lehrbücher, wie z.B. [14]. Der A*-Algorithmus wird neben dem originalen Paper [12] auch in vielen Büchern, z.B. [16], [13] oder [2] beschrieben.

Verwandte Probleme:

- Erweiterung auf mehr als zwei Sequenzen, das Multiple Sequence Alignment, erläutert u.a. in [10] oder [11]
- Lokales Alignment, lösbar mit dem Smith-Waterman-Algorithmus [18]: Suche aus beiden Sequenzen je eine Teilsequenz, die möglichst ähnlich sind.

In [3] wird eine Anwendung des A*-Algorithmus auf das Multiple Sequence Alignment beschrieben. Hier werden die vorliegenden paarweisen Alignments für die einzelnen Sequenzenpaare für die Heuristik verwendet.

Zur Programmierung in C++ bietet das Tutorial [19] eine gute Einführung.

# 2 Das Problem

**Definition 1.** Ein *Alphabet* $A$ sei eine endliche Menge $A = \{A_1, ..., A_n\}$ von Zeichen.
Eine *Sequenz* $S = (s_1, ..., s_k)$ sei eine endliche Abfolge von Elementen $s_i \in A$.
Die *Länge* der Sequenz sei $l(S) := \#S$.
Eine *Kostenfunktion* $d$ sei eine Abbildung $d : A \times A \to \mathbb{R}$

Beispielsweise besteht das Alphabet bei der DNA aus den Basen $A, C, T, G$ (=Adenin, Cytosin, Thymin, Guanin). Eine Sequenz wäre dann z.B. $C, T, G, G, A, C, T$ und eine triviale Kostenfunktion $d(a, b) = 1$, für $a \neq b$, $d(a, b) = 0$ für $a = b$

**Definition 2** (Paarweises Sequenz-Alignment)**.** Sei $A$ ein gegebenes Alphabet. Betrachte dann $A' = A \cup \{-\}$. $'-'$ steht für eine sogenannte Lücke in der Sequenz. Ein *paarweises Sequenz-Alignment* für zwei Sequenzen $S^1$, $S^2$ ist jetzt eine Matrix $P$ mit 2 Zeilen und $L$ Spalten, wobei gilt:

- $max(l(S^1), l(S^2)) \leq L \leq l(S^1) + l(S^2)$
- Alle Einträge in $P$ sind aus $A'$.
- In der 1. Zeile sind alle Elemente aus $S^1$ in der gleichen Reihenfolge, d.h. für $S^1_i, S^1_j$ aus $S^1$ mit $i < j$ gibt es $i' < j'$, so dass $P_{1i} = S^1_i$, $P_{1j} = S^1_j$. Genauso für die 2.Zeile.
- Es gibt kein $i \in [1, L]$, so dass $P_{1i} = P_{2i} = '-'$, also keine zwei Lücken untereinander.

Uns reicht es allerdings nicht, irgendein paarweises Sequenz-Alignment zu finden, dies wäre auch trivial möglich, in dem man die beiden Sequenzen untereinander schreibt und die kürzere mit Lücken auffüllt. Stattdessen wollen wir das beste paarweise Sequenz-Alignment finden. Was das bedeutet, definieren wir wie folgt:

**Definition 3** (Optimales paarweises Alignment)**.** Sei wieder ein Alphabet $A$ gegeben und $A'$ wie zuvor. Zudem gebe es jetzt eine Kostenfunktion $d : A' \times A' \to \mathbb{R}$, also insbesondere seien auch $d(a, -)$ und $d(-, a)$ für alle $a \in A$ definiert. Wir

betrachten die Summe der Kosten in allen Spalten, also aller entstehenden Paare

$$C(P) := \sum_{i=1}^{L} d(P_{1i}, P_{2i}).$$

Ein *optimales paarweises Alignment* ist das (nicht eindeutige) paarweise Alignment, dass $C$ minimiert.

Wir kennen jetzt also unsere Aufgabe, nämlich für beliebige zwei Sequenzen und eine beliebige Kostenfunktion ein optimales paarweises Alignment zu ermitteln. Im nächsten Abschnitt werden wir einen Lösungsansatz dafür kennen lernen. Zunächst aber noch ein kurzes Beispiel:

**Beispiel 1.** Seien das Alphabet $A = \{C, D, E, F\}$ und die Sequenzen

$$S^1 = (C, D, E, E, F)$$

und

$$S^2 = (C, E, F, D)$$

gegeben. Dann sind folgende Matrizen paarweise Alignments:

$$P^1 = \begin{pmatrix} C & D & E & E & F \\ C & E & F & D & - \end{pmatrix}$$

$$P^2 = \begin{pmatrix} C & D & E & E & F & - \\ C & - & - & E & F & D \end{pmatrix}$$

$$P^3 = \begin{pmatrix} C & D & E & E & F \\ C & - & E & F & D \end{pmatrix}$$

Für die Kostenfunktion $d(a, b) = \begin{cases} 0, & \text{falls } a = b \\ 1, & \text{falls } a \neq b,\ a, b \neq' - ' \\ 2, & \text{falls } a =' - ' \text{ oder } b =' - ' \end{cases}$
erhalten wir dann $C(P^1) = 0+1+1+1+2 = 6$, $C(P^2) = 0+2+2+0+0+2 = 6$, $C(P^3) = 0 + 2 + 0 + 1 + 1 = 4$; $P^3$ ist also am besten und wie man sich leicht

überlegt, auch insgesamt das optimale Alignment.
Falls uns Lücken lieber sind als falsche Zuordnungen, könnten wir in der Kostenfunktion die 1 und die 2 vertauschen. In dem Fall erhalten wir $C(P^1) = 7$, $C(P^2) = 3$, $C(P^3) = 5$ und $P^2$ wäre optimal.

Später brauchen wir noch folgende Definitionen:

**Definition 4** (Graph)**.** (nach [5]) Ein *Graph* ist ein Paar $G = (V, E)$ von endlichen Mengen, so dass $E \subseteq V \times V$, d.h. die Elemente von $E$ sind zweielementige Teilmengen von $V$. Die Elemente aus $V$ sind die *Knoten* des Graphen $G$, die Elemente aus $E$ sind seine *Kanten*. Ein *Weg* ist eine geordnete, endliche Teilmenge der Knoten $w = w_0, ..., w_k, k \in \mathbb{N}, w_i \in V \ \forall i \in [0, k]$.

**Definition 5** (Kürzeste-Wege-Problem)**.** (nach [4]) In einem *Kürzeste-Wege-Problem* haben wir einen gewichteten Graphen $G = (V, E)$ mit einer Gewichtsfunktion $c : E \to \mathbb{R}$. Das *Gewicht* eines Weges $w = (w_0, w_1, ..., w_k)$ ist die Summe der Gewichte der enthaltenen Kanten:

$$l(w) = \sum_{i=1}^{k} c(w_{i-1}, w_i).$$

Wir definieren das *Kürzester-Weg-Gewicht* von $u$ nach $v$ durch

$$\delta(u, v) = \begin{cases} \min\{l(w) : w \text{ ist Weg von } u \text{ nach } v\}, & \text{falls es einen Weg von } u \text{ nach } v \text{ gibt} \\ \infty, & \text{sonst} \end{cases}$$

# 3 Idee von Needleman und Wunsch

Saul B. Needleman und Christian D. Wunsch haben 1969 in [15] folgende Methode für das Bestimmen des optimalen paarweisen Alignments beschrieben:
Für die zwei Sequenzen $A = (A_1, ..., A_m)$ und $B = (B_1, ..., B_n)$ erstellt man eine zweidimensionale Matrix $M$ mit m Spalten und n Zeilen, wobei die Spalten genau den Elementen von $A$ und die Zeilen den Elementen von $B$ entsprechen. Der Matrixeintrag $M_{ij}$ entspricht dann einer Kombination von $A_j$ und $B_i$. (z.B. Tabelle 1)
Wir wandeln diese Matrix jetzt für unsere Zwecke in einen Graphen (z.B. Abb. 1) um. Dazu ergänzen wir eine 0. Spalte und 0. Zeile, die Matrix hat dann Größe $(n+1) \times (m+1)$. Dann richten wir für jeden Eintrag der Matrix $M_{ij}$ einen Knoten $(i, j)$ ein. Anschließend verbinden wir jeden Knoten durch in ihm startende Kanten mit seinem rechten, rechten unteren und unteren Nachbarn, falls dies möglich ist. Knoten $(i, j)$ wird also mit den Knoten $(i, j + 1)$, $(i + 1, j + 1)$ und $(i + 1, j)$ verbunden, falls diese existieren. Die Idee dahinter ist, dass wir beim Erstellen der Alignment-Matrix $P$ zu jedem Zeitpunkt (also wenn wir die ersten $k$ Spalten erstellt haben für $0 \leq k < L$) genau drei Möglichkeiten haben:

- Ordne die jeweils in den Sequenzen nächsten Einträge einander zu, d.h. in die $(k + 1)$-te Spalte von $P$ kommen zwei Nicht-Lücken. Dies entspricht der Kante nach rechts unten, also von $(i, j)$ nach $(i + 1, j + 1)$.

- Füge in der Sequenz $A$, also in der 1.Zeile von $P$ eine Lücke ein und ordne dieser den nächsten Eintrag aus Sequenz $B$ zu. Dies entspricht der Kante nach unten, also von $(i, j)$ nach $(i + 1, j)$.

- Füge in der Sequenz $B$, also in der 2. Zeile von $P$ eine Lücke ein und ordne dieser den nächsten Eintrag aus Sequenz $A$ zu. Dies entspricht der Kante nach rechts, also von $(i, j)$ nach $(i, j + 1)$.

Jetzt müssen wir noch überlegen, welche Gewichte die Kanten erhalten sollen. Da kommt wieder die Kostenfunktion ins Spiel.

*Bemerkung* 1. Zur Vereinfachung der Darstellung nehmen wir ab hier an, dass die

Kosten für eine Lücke immer gleich sind, also dass gilt:

$$d(a,-) = d(-,a) = d_{Gap}, \forall a \in A, d_{Gap} \in \mathbb{R} \text{ fest}$$

Diese Annahme entspricht durchaus der Realität, ohne sie würde das Verfahren trotzdem funktionieren, wir würden aber später eventuell eine schlechtere Heuristik erhalten.

Da die horizontalen und vertikalen Kanten jeweils dem Einfügen einer Lücke entsprechen, setzen wir deren Gewichte auf $d_{Gap}$. Für die diagonalen Kanten betrachten wir die Paare, die wir durch sie erhalten, also die Kante, die in $(i,j)$ endet, bekommt als Gewicht die Kosten $d(A_j, B_i)$. Dadurch erreichen wir, dass das Bestimmen des optimalen Alignments äquivalent zum Bestimmen des kürzesten Weges von $(0,0)$ nach $(n,m)$ ist, denn für das Alignment ist die Zielfunktion ja gerade die Summe der Kosten und beim kürzesten Weg die Summe der Gewichte auf den benutzten Kanten. Der Start in $(0,0)$ (und nicht in $(B_1, A_1)$) ist notwendig, damit wir auch in der 1.Zeile bereits die Möglichkeit haben, Lücken einzufügen. Wenn wir dann den kürzesten Weg haben, ist die Konstruktion des Alignments einfach, da die gewählten Kanten nach obiger Konstruktion festlegen, ob und wo wir Lücken einfügen.

Wie finden wir jetzt den kürzesten Weg? Seien dafür $D(i,j)$ die Kosten für den kürzesten Weg von $(0,0)$ zu dem Knoten $(i,j)$. Nach unserer Konstruktion gehen von jedem Knoten nicht nur (maximal) drei Kanten ab, sondern es kommen auch (maximal) drei Kanten dort an, nämlich von links, links oben und oben. Wir haben also drei Möglichkeiten einen Knoten $(i,j)$ zu erreichen. Somit ergibt sich mit den oben beschriebenen Kosten folgenden rekursive Formulierung:

$$\begin{aligned} D(i,j) = \min\{&D(i-1,j-1) + d(B_{i-1}, A_{j-1}),\\ &D(i-1,j) + d_{Gap},\\ &D(i,j-1) + d_{Gap}\} \end{aligned}$$

| | 0 | C | D | E | E | F |
|---|---|---|---|---|---|---|
| 0 | $(0,0)$ | $(0,1)$ | | $\cdots$ | | $(0,5)$ |
| C | $(1,0)$ | $(1,1)$ | | | | $(1,5)$ |
| E | $\vdots$ | | | $\ddots$ | | |
| F | $\vdots$ | | | | $\ddots$ | |
| D | $(4,0)$ | $(4,1)$ | | $\cdots$ | | $(4,5)$ |

Tabelle 1: Needleman-Wunsch-Matrix

wobei die „Randbedingungen“

$$\begin{aligned} D(0,0) &= 0 \\ D(i,0) &= D(i-1,0) + d_{Gap} \\ D(0,j) &= D(0,j-1) + d_{Gap} \end{aligned}$$

lauten. Somit könnten wir jetzt den kürzesten Weg nach $(n,m)$ berechnen. Da wir aber für jeden Wert $D(i^*,j^*)$ rekursiv alle anderen Werte $D(i,j)$ mit $i \leq i^*$ und $j \leq j^*$ berechnen müssen, ist dieses Verfahren sehr aufwändig und dadurch besonders bei großen Instanzen ungeeignet.
Der Ausweg lautet hier „Dynamische Programmierung“ und wird im nächsten Abschnitt vorgestellt. Dazu führen wir auch den Dijkstra-Algorithmus ein, der eben auf diese Weise kürzeste Wege findet. Danach kehren wir wieder zu unserem Problem zurück und schauen uns kurz an, wie Needleman und Wunsch in ihrem Paper das Prinzip der dynamischen Programmierung angewandt haben.

**Beispiel 2.** Situation wie in Beispiel 1: $A = (C,D,E,E,F)$, $B = (C,E,F,D)$,

$$d(a,b) = \begin{cases} 0, & \text{falls } a = b \\ 1, & \text{falls } a \neq b,\ a,b \neq' -' \\ 2, & \text{falls } a =' -' \text{ oder } b =' -' \end{cases}$$

gibt die Matrix in Tabelle 1 und damit den Graphen in Abbildung 1. Die kürzesten Wege zu den jeweiligen Knoten ergeben sich dann wie in Tabelle 2.

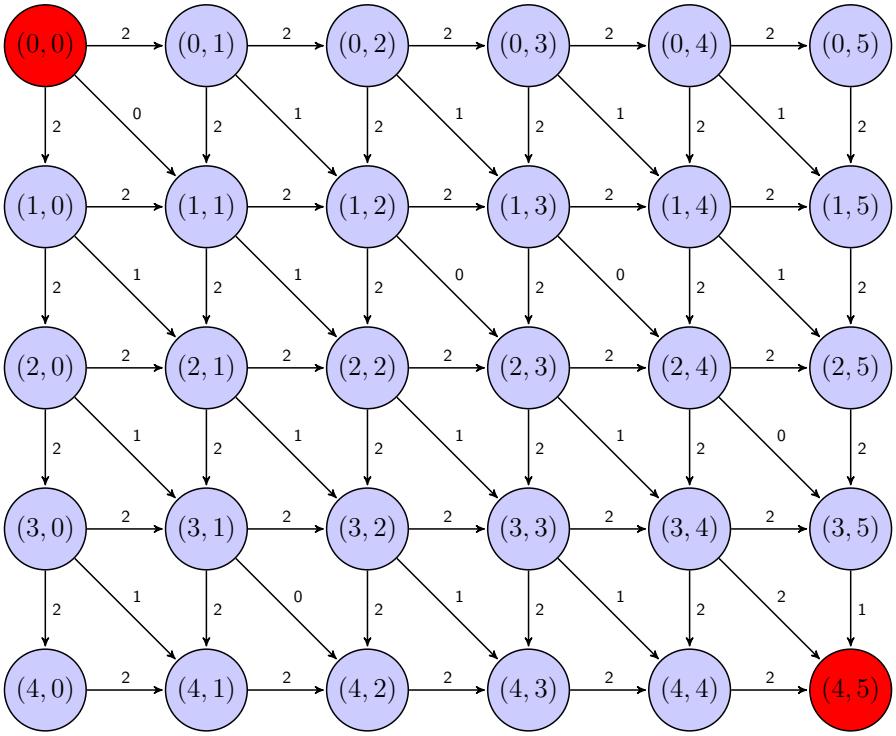



Abbildung 1: Needleman-Wunsch-Graph

| | 0 | C | D | E | E | F |
|---|---|---|---|---|---|---|
| 0 | **0** | 2 | 4 | 6 | 8 | 10 |
| C | 2 | **0** | **2** | 4 | 6 | 8 |
| E | 4 | 2 | 1 | **2** | 4 | 6 |
| F | 6 | 4 | 3 | 2 | **3** | 4 |
| D | 8 | 6 | 4 | 4 | 3 | **4** |

Tabelle 2: Kürzeste Wege zu allen Knoten

Wir können also den rechten unteren Knoten $(4, 5)$ mit Kosten 4 erreichen und zwar über die Knoten, die fett hervorgehoben sind. Daraus erhalten wir jetzt unser optimales paarweises Alignment, indem wir den Weg abarbeiten. Hier als erste Spalte $(C, C)$, dann $(D, -)$ (wegen der horizontalen Kante, die einer Lücke entspricht), weiter dann $(E, E)$,$(E, F)$ und $(F, D)$. Dadurch erhalten wir genau die Matrix

$$P^3 = \begin{pmatrix} C & D & E & E & F \\ C & - & E & F & D \end{pmatrix},$$

von der wir schon in Beispiel 1 angenommen hatten. dass sie optimal sei.

# 4 Dynamische Programmierung

Unter *dynamischer Programmierung* fasst man eine bestimmte Art von Algorithmen zum Lösen von Optimierungsproblemen zusammen. Man kann sie benutzen, wenn das Problem aus vielen kleineren identischen Teilproblemen besteht. Außerdem muss das *Optimalitätsprinzip von Bellmann* gelten:

**Definition 6** (Optimalitätsprinzip von Bellmann)**.** (übersetzt aus [1]) Eine optimale Verfahrensweise hat die Eigenschaft, dass unabhängig vom Anfangszustand und der ersten Entscheidung, die verbleibenden Entscheidungen eine optimale Verfahrensweise unter Beachtung des Zustandes nach der ersten Entscheidung darstellen.

Kurz gesagt: Die optimale Lösung für ein Problem setzt sich aus den optimalen Lösungen kleinerer Probleme zusammen.

*Bemerkung* 2. Bei dem paarweisen Sequenz-Alignment ist das Optimalitätsprinzip im folgenden Sinne erfüllt. Wenn wir bereits die ersten $k$ Spalten unserer Lösungsmatrix $P$ kennen, bzw. diese fest sind, müssen wir für das Erreichen des optimalen Alignments die verbliebenen $L - k$ Spalten nach dem gleichen Prinzip berechnen. Das funktioniert, weil die Zielfunktion eine Summe über die Spalten ist.

Der Trick der dynamischen Programmierung ist jetzt, als erstes die kleinsten Teilprobleme optimal zu lösen, diese Ergebnisse zu speichern und dann für die Berechnung der nächstgrößeren Probleme zu benutzen (auch *bottom-up* genannt). Dadurch wird die wiederholte Berechnung des gleichen Teilproblems vermieden und der Algorithmus hat eine wesentlich kürzere Laufzeit als die entsprechende rekursive Formulierung. Nachdem alle benötigten Teilprobleme berechnet wurden, kann dann die Gesamtlösung schnell zusammengesetzt werden.

Dynamische Programmierung kann in verschiedenen Bereichen der Optimierung eingesetzt werden, wo sonst ein rekursiver Algorithmus verwendet werden würde

Das Verfahren läuft ungefähr so ab:

1. Analysiere die Struktur der Optimallösung

2. Formuliere eine rekursive Darstellung der Lösung
3. Berechne die Lösung iterativ
4. Konstruiere die Lösung *aus den Teillösungen* der vorigen Schritte

**Beispiel 3** (Fibonacci)**.** Wir wollen die Fibonacci-Zahlen berechnen. Diese sind wie folgt rekursiv definiert: $F_1 = F_2 = 1$ und $F_{n+2} = F_n + F_{n+1}$ für $n \geq 1$. Man greift also in jedem Schritt auf zwei vorherige Lösungen zurück, die wieder je zwei vorherige Lösungen brauchen usw. Wenn man sich jetzt streng an die rekursive Formulierung hält, braucht man also ca. $2^{n-1}$ Berechnungen. Wenn man z.B. $F_4$ bestimmt, greift man auf $F_3$ und $F_2$ zurück. Für $F_2$ wiederum benötigt man $F_1$ und $F_0$ und für $F_3$ muss $F_2$ und $F_1$ berechnet werden. Unsere rekursive Formulierung muss jetzt für $F_2$ wiederum $F_1$ und $F_0$ bestimmen. Wie man sieht, wurde $F_0$ zweimal, $F_1$ dreimal und $F_2$ zweimal berechnet.

Jetzt wenden wir Dynamische Programmierung an. Die Fibonacci-Zahlen sind genauso definiert, aber wir berechnen jetzt zur Bestimmung von $F_n$ nacheinander die $F_i, i = 1, ..., n$, beginnend mit $F_1$. Für $F_4$ berechnen wir also $F_1$, $F_2$, $F_3$ und dann $F_4$. Da die benötigten zwei Vorgänger jeweils bekannt sind, sind keine weiteren Berechnungen notwendig. Da keine Werte mehrfach bestimmt werden, haben wir nun also lineare Laufzeit und bei geschicktem Überschreiben kommt man sogar mit konstantem Speicherplatz aus.

# 5 Dijkstra-Algorithmus

Ein bekannter Algorithmus mit dynamischer Programmierung ist der *Dijkstra-Algorithmus*[6] für das Kürzeste-Wege-Problem mit nicht-negativen Kantengewichten. Dabei beginnt man im Startknoten und sucht von dort den Knoten, der am schnellsten zu erreichen ist (schnell bedeutet: die Summe der Kantengewichte auf dem Weg dorthin ist am geringsten). Dann sucht man den Knoten. der am zweitschnellsten vom Startknoten aus zu erreichen ist und führt das so lange durch bis man alle Knoten erreicht hat. Nach Konstruktion hat man diese dann auf dem jeweils kürzesten Weg erreicht und diesen somit gefunden. Für jeden Knoten, den wir zwischendurch erreichen, speichern wir die Distanz zum Startknoten. Dadurch müssen wir den kürzesten Weg dorthin nicht mehrfach bestimmen, sondern können auf den bekannten Wert zurückgreifen.

**Algorithm 1** Dijkstra-Algorithmus

1: Initialisierung: Jeder Knoten $v$ bekommt die Bewertung $d(v) = \infty$, nur der Startknoten $s$ die Bewertung $d(s) = 0$.
2: **while** Es gibt Knoten, die noch nicht bearbeitet wurden **do**
3: Wähle daraus den Knoten $v$ mit der minimalen Bewertung aus
4: $v$ als bearbeitet markieren
5: **for** alle nicht bearbeiteten Nachbarknoten $w_i$ **do**
6: Berechne die Summe aus Bewertung von $v$ und Kantengewicht zwischen $v$ und $w_i$.
7: **end for**
8: **if** Diese Summe ist kleiner als die Bewertung von $w_i$ **then**
9: Setze Summe als neue Bewertung von $w_i$
10: Speichere $v$ als Vorgänger von $w_i$
11: **end if**
12: **end while**

Auf diese Weise erreicht man irgendwann den Zielknoten auf dem kürzesten Weg und kann diesen Weg rekonstruieren, indem man die jeweils gespeicherten Vorgänger benutzt. Das Verfahren funktioniert, weil das Optimalitätsprinzip von Bellman hier erfüllt ist: Die kürzesten Strecken zwischen Knoten in einem Pfad bilden zusammen die kürzeste Strecke auf dem Pfad. Dazu folgendes Lemma (aus [14]):

**Lemma 1.** *Sei $d(v)$ die vom Algorithmus bestimmte Bewertung von $v$ und $s \in V$ ein beliebiger Startknoten. Dann gilt beim Abbruch des Algorithmus für alle $v \in V$ $d(v) = \delta(s, v)$. ($\delta(s, v)$: kürzester Weg von $s$ nach $v$, wie in Def.5)*

*Beweis.* Sei $S_i$ die Menge der bearbeiteten Knoten mit $i$ Elementen. Ein Weg $w = (w_1, ..., w_{n+1})$ heiße *Grenzweg*, falls $w_1, ..., w_n \in S_i$ und $w_{n+1} \in V \setminus S$. Dann behaupten wir:

1. Für alle $u \in S_i$ ist $d(u) = \delta(s, u)$ und für den kürzesten Weg $w$ von $s$ nach $u$ gilt $w \subseteq S_i$

2. Für alle $u \in V \setminus S_i$ ist $d(u) = \min\{l(w) : w \text{ ist Grenzweg von } s \text{ nach } u\}$ , wenn kein solcher Weg existiert, dann ist $d(u) = \infty$.

Wir zeigen das per Induktion über $i$:
*Induktionsanfang*: $i = 1$
$d(s) = 0$, also ist $S_1 = \{s\}$ und 1. gilt. Nach der Bearbeitung des ersten Knotens gilt nach Konstruktion für alle zu $s$ benachbarten Knoten $u$, dass $d(u) = l((s, u))$ und für alle anderen $l(u) = \infty$. Da alle Grenzwege die Form $(s, u)$ haben, folgt Behauptung 2.
*Induktionsvoraussetzung*: 1. und 2. gelten für $i$.
*Induktionsschritt*: $i \to i + 1$
Sei $S_{i+1} = S_i \cup \{v\}$. Der Knoten $v$, der dazugenommen wird, ist der mit der kleinsten Bewertung. Da es immer mindestens einen Knoten gibt, der schon erreicht aber noch nicht bearbeitet wurde, gilt $d(v) < \infty$. Wir nehmen an, dass es einen kürzesten Weg $w$ von $s$ nach $v$ gibt, der kürzer als $d(v)$ ist, also $\delta(s, v) < d(v)$. Nach IV 2. hat bei $S_i$ der kürzeste Grenzweg von $s$ nach $v$ die Länge $d(v)$. Somit kann $w$ kein Grenzweg in $S_i$ sein. Sei also $u$ das erste Element von $w$, das nicht in $S_i$ liegt und definiere $w' \subset w$ als $w' = (s, ..., u)$. $w'$ ist somit ein Grenzweg von $s$ nach $u$ und wiederum wegen IV 2. gilt $l(w') \geq d(u)$. Da $v$ das unbearbeitete Element mit der geringsten Bewertung ist und $u \notin S_{i+1}$, muss gelten $d(u) > d(v)$. Da wir nur positive Gewichte zulassen, können wir folgern: $l(w) \geq l(w') \geq d(u) \geq d(v)$, was einen Widerspruch zur Annahme $l(w) < d(v)$ darstellt. Es gibt also keinen Weg nach $v$, der kürzer ist als $d(v)$. Da wegen IV 2. ein Grenzweg der Länge $d(v)$ existiert, haben wir 1. gezeigt.

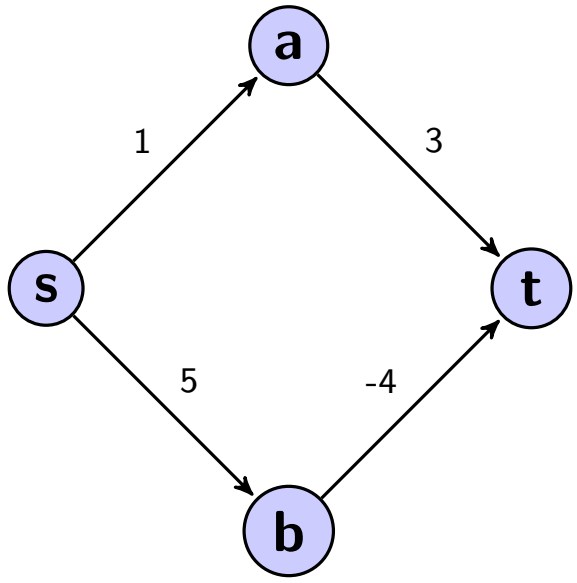



Abbildung 2: Dijkstra mit negativen Gewichten

Sei jetzt $u \in V \setminus S_{k+1}$. $d(u)$ und $d(v)$ seien die Bewertungen vor der Bearbeitung im $(i+1)$-ten Schritt. Wir suchen jetzt die Länge $l(w)$ des kürzesten Grenzwegs in $S_{i+1}$ von $s$ nach $u$. Diese ist das Minimum aus der Länge des kürzesten Grenzweges in $S_i$ von $s$ nach $u$ (anschaulich: wir lassen $v$ aus und gehen direkt nach $u$) und $\delta(s,v) + c(v,u)$ (anschaulich: wir gehen über $v$ nach $u$). Nach IV hat der kürzeste Grenzweg in $S_i$ die Länge $d(u)$ und es gilt $d(v) = \delta(s,v)$. Also erhalten wir:

$$l(w) = \min\{d(u), d(v) + c(v,u)\}.$$

Nach der Konstruktion des Algorithmus wird $d(u)$ aber gerade auf diesen Wert gesetzt, also $d(u) = l(w)$ und 2. ist erfüllt. Insbesondere wird vor Abbruch $t$ in $S_i$ aufgenommen und 1. zeigt somit das Lemma. □

*Bemerkung* 3. Im Beweis nutzen wir, dass die Kantengewichte positiv sind. für negative Kantengewichte muss der Dijkstra-Algorithmus also nicht mehr funktionieren. Tatsächlich gibt es solche Fälle, wie Abbildung 2 zeigt.

Hier wird zuerst $a$ ausgewält und dann schon direkt $t$, weil $1 + 3 < 5$ ist. Das Problem ist, dass wir durch das Auswählen der Knoten nach Bewertung den

Knoten $b$ hier gar nicht untersuchen. Der kürzeste Weg würde aber über $b$ verlaufen, also funktioniert der Algorithmus hier nicht.

## 5.1 Laufzeit und Speicherplatz

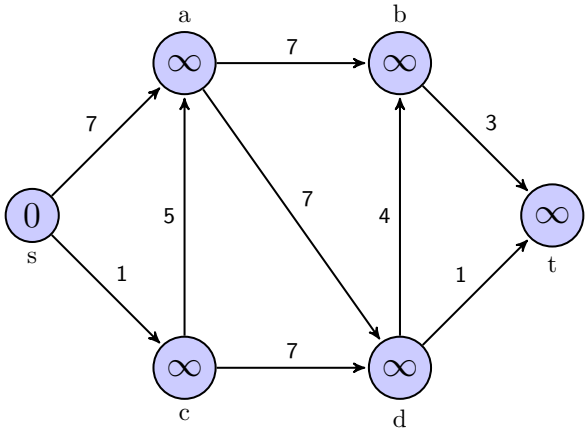
Die Knoten und Kanten können in Matrizen gespeichert werden, die Vorgänger als Zeiger und die Bewertungen in Feldern. Die Menge $Q$ der unbearbeiteten Knoten wird optimal in einer Prioritätswarteschlange („priority queue“) gespeichert, die nach der Bewertung sortiert ist und als Fibonacci Heap gespeichert wird. Dann gilt Folgendes (siehe [14]):

**Satz 1.** *Der Dijkstra-Algorithmus kann so implementiert werden, dass die Laufzeit* $\mathcal{O}(n \log n + m)$ *beträgt.*

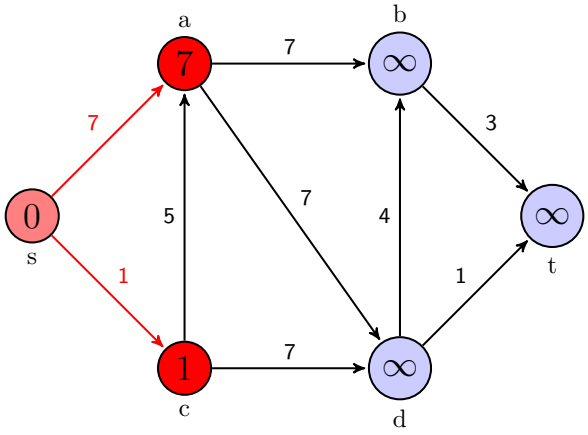
*Beweis.* Zunächst müssen die $n$ Knoten in die Warteschlange eingefügt werden, das sind $n$ Operationen. Es werden im Worst Case alle Knoten untersucht, die Schleife wird also $n$-mal durchlaufen, wobei jedes Mal geprüft wird, ob $Q$ leer ist, und ein Element aus $Q$ entfernt wird. Die Summe aus Bewertung und Kantengewicht muss im Worst Case für alle $m$ Kanten des Graphen berechnet werden, mit der bisherigen Bewertung verglichen werden und dadurch die Reihenfolge in der Warteschlange geändert werden. Fast alle Operationen brauchen $\mathcal{O}(1)$ Zeit, nur das Suchen des kleinsten Elements dauert bei der Fibonnaci Heap $\mathcal{O}(\log n)$. Somit erhalten wir für die Gesamtlaufzeit $\mathcal{O}(n \log n + m)$. □

## 5.2 Beispiel

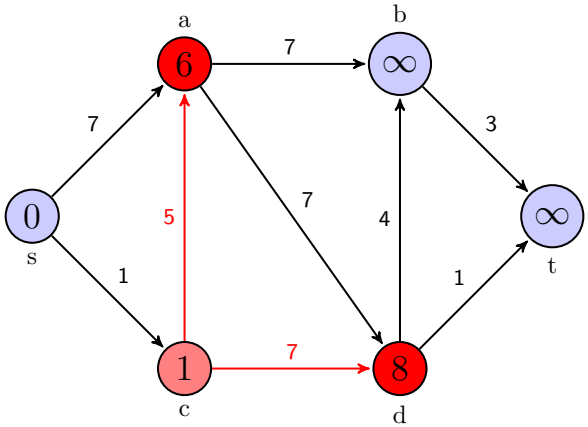
Betrachten wir folgenden Graphen, wo wir den kürzesten Weg von $s$ nach $t$ suchen. Dabei stellt der Wert im Knoten die jeweils aktuelle Bewertung, also die bisher kleinste Entfernung zu dem Knoten, dar.



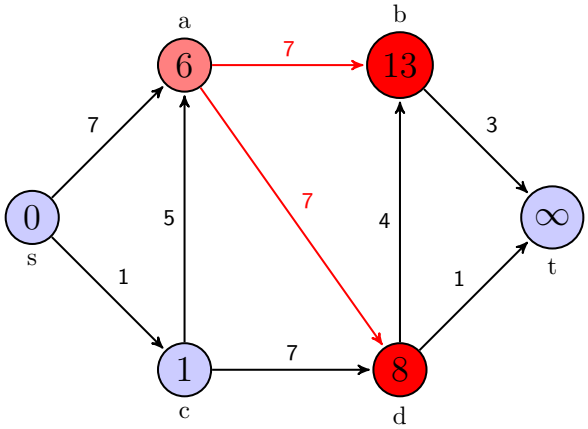

Wir beginnen bei $s$ und untersuchen als erstes die Nachbarknoten $a$ und $c$.



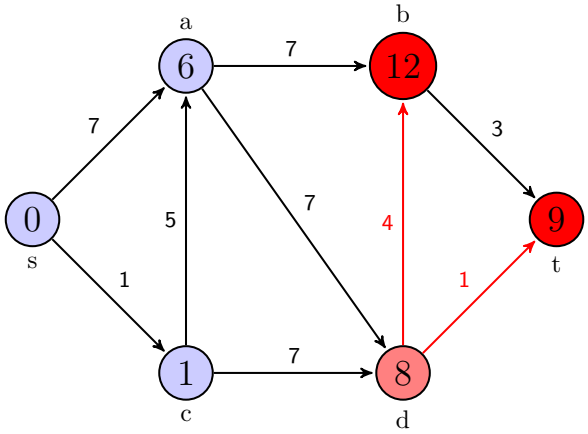

Wir haben jetzt also die Bewertungen $d(a) = 7, d(c) = 1$. Wir wählen die kleinste Bewertung und untersuchen Knoten $c$.



Jetzt haben wir einen kürzeren Weg zu $a$ gefunden, denn die Summe aus $d(c) = 1$ und dem Kantengewicht von $(c, a)$ ist $6 < 7$. Also setzen wir $d(a) = 6$. Außerdem ist $d(d) = 1 + 7 = 8$. Wir untersuchen also als nächstes $a$.




Für $b$ erhalten wir jetzt eine Bewertung von $d(b) = 6 + 7 = 13$. für $d$ würden wir auch eine Bewertung von $6 + 7 = 13$ erhalten, aber da die bisherige Bewertung $8 < 13$ ist, bleibt $d(d) = 8$. Dies ist auch die geringste Bewertung von allen nicht vollständig untersuchten Knoten, also untersuchen wir jetzt d.




Für $b$ erhalten wir jetzt als neue Bewertung $d(b) = 8 + 4 = 12 < 13$. Vor allem erreichen wir aber den Zielknoten $t$ mit der Bewertung $d(t) = 8 + 1 = 9$ und sind damit fertig.

## 5.3 Beziehung zu Needleman/Wunsch

Den Dijkstra-Algorithmus kann man jetzt auch auf unser Kürzeste-Wege-Problem beim Sequenz-Alignment anwenden, *vorausgesetzt alle Gewichte sind nichtnegativ.* Wir kennen jetzt also einen Weg, das optimale paarweise Alignment zu finden. Für die Laufzeit erhalten wir $\mathcal{O}(nm \log nm)$, wenn $n$ und $m$ wieder die Längen der Sequenzen sind, denn es gibt dann nach Konstruktion $(n+1) * (m+1) = \mathcal{O}(nm)$ Knoten und weniger als $3(n+1) * (m+1) = \mathcal{O}(nm)$ Kanten.
Needleman und Wunsch gehen etwas anders vor: Neben der Tatsache, dass sie hier maximieren statt minimieren wollen, weil sie keine Kostenfunktion sondern eine positive Bewertungsfunktion benutzen und sie rechts unten statt links oben anfangen, was nur ein Unterschied in der Notation ist, gibt es auch einen wichtigen Unterschied. Sie gehen zeilenweise die Knoten durch und ermitteln jeweils den kürzesten Weg zu ihnen. Dabei wird kein Knoten mehrmals betrachtet. Dies funktioniert, da die Knoten nach der Konstruktion topologisch sortiert sind.

**Definition 7.** [4] Eine *topologische Sortierung* eines gerichteten azyklischen Graphen $G = (V, E)$ ist eine lineare Anordnung aller seiner Knoten mit der Eigenschaft, dass $u$ in der Anordnung vor $v$ liegt, falls es $(u, v) \in E$ gibt.

**Lemma 2.** *Der Graph, der von Needleman/Wunsch konstruiert wurde, ist topologisch sortiert.*

*Beweis.* Die lineare Anordnung ist gegeben durch
$((0,0), ..., (0,m), (1,0), ..., (1,m), ..., (n,0), ..., (n,m)$, denn aufgrund der Konstruktion gilt für alle Kanten $((i,j),(i',j'))$ $i \leq i'$ und $j \leq j'$ (sowie $(i,j) \neq (i',j')$). Somit ist $(i,j)$ in der Anordnung stets vor $(i',j')$. □

Damit kann man folgern:

**Korollar 1.** *Der Needleman/Wunsch-Algorithmus findet den kürzesten Weg in* $\mathcal{O}(nm)$

*Beweis.* Durch das Durchlaufen des Graphen gemäß der topologischen Sortierung kann jeder untersuchte Knoten nur von zuvor untersuchten Knoten erreicht werden. Dadurch kennt man nach der ersten Untersuchung des Knoten den kürzesten Weg

dorthin bereits sicher. Somit erhält man auch den kürzesten Weg nach $(n, m)$, nachdem man diesen Knoten als letztes untersucht hat.
Zur Laufzeit: In jedem Knoten muss man die Gewichte zu drei Nachbarknoten miteinander vergleichen. Dies geht in konstanter Zeit. Da es $\mathcal{O}(nm)$ Knoten gibt, wird der gesamte Graph in $\mathcal{O}(nm)$ untersucht. □

Der Dijkstra-Algorithmus braucht länger, weil er noch zusätzlich eine Minimum-Suche durchführt.

Auf diese Beobachtung kommen wir später zurück, wenn wir die praktischen Ergebnisse diskutieren. Jetzt wollen wir im nächsten Schritt die Performance des Dijkstra-Algorithmus verbessern, indem wir ihn mit einer Heuristik zu einem *A*-Algorithmus* machen.

# 6 A*-Algorithmus

Wir wollen uns jetzte einen weiteren Algorithmus für das Kürzeste-Wege-Problem anschauen. Dafür benutzen wir einen *heuristischen Ansatz*, das bedeutet, dass wir spezielle Information über das Problem nutzen, um die Effizienz des Algorithmus zu erhöhen.

**Beispiel 4** (Routenplanung)**.** Das typische Beispiel für dieses Verfahren ist kürzeste Wege z.B. in einem Straßennetz zwischen geographisch existierenden Punkte, also z.B. Städten zu suchen. Die Abschätzung der Wegstrecke kann hierbei über die „Luftlinie“, also die euklidische Entfernung zwischen den Punkten erfolgen. Es kann kein Weg kürzer sein, als dieser direkte, insofern bildet diese Entfernung eine untere Schranke. Das wird in der Routenplanung dazu genutzt, dass der optimale Weg nicht gleichmäßig in alle Richtungen gesucht wird, sondern Zwischenpunkte, die näher am Ziel sind. besser bewertet werden.

In diesem Abschnitt folgen wir weitesgehend dem Artikel [12], in dem der A*-Algorithmus eingeführt wurde.

Ein wesentlicher Unterschied in der Problemstellung ist, dass wir hier nicht mehr die kürzesten Wege zu allen Knoten bestimmen wollen, sondern nur zu einer Menge $T$ von *Zielknoten*.

## 6.1 Definitionen

Wir nennen einen Knoten $v \in V$ *Nachfolger* von $u \in V$, wenn es eine Kante von $u$ nach $v$ gibt.

Der *Nachfolge-Operator* $\Gamma$ ist auf der Menge der Knoten $V$ definiert und gibt zu jedem Knoten $v_i$ die Menge von Paaren $\{(v_j, c_{ij})\}$, wobei die $v_j$ die Nachfolger von $v_i$ sind und die $c_{ij}$ die jeweiligen Gewichte der Kanten von $v_i$ nach $v_j$.

Wenn $\Gamma$ auf einen Knoten angewendet wird, sagen wir, dass der Knoten *erweitert* wird.

Der *Subgraph* $G_v$ von irgendeinem Knoten $v \in V$ ist der Teilgraph, der aus $v$ durch mehrfache Anwendung des Nachfolge-Operators $\Gamma$ entsteht, also alle Knoten enthält, die von $v$ aus erreichbar sind.

Für jeden Knoten $v \in V$ ist ein Element $t \in T$ ein *bevorzugter* Zielknoten, falls die Kosten des optimalen Wegs von $v$ nach $t$ nicht größer sind als die Kosten irgendeines anderen Weges von $v$ zu irgendeinem Zielknoten.

Wir nennen einen Algorithmus *zulässig*, falls es sicher ist, dass man einen kürzesten Weg von $s$ zu einem bevorzugten Zielknoten von $s$ auf jedem Graph mit positiven Kantengewichten findet.

## 6.2 Idee

Die Grundidee ist die gleiche wie beim Dijkstra-Algorithmus: Wie beginnen den Algorithmus bei einem *Startknoten* $s$ und erzeugen einen Teil des Subgraphen $G_s$ durch Anwendung des Nachfolge-Operators. Jedes Mal, wenn ein Knoten erweitert wird, speichern wir für jeden Nachfolger $v$ sowohl die Kosten um auf dem bisher kürzesten gefundenen Weg zu $v$ zu kommen als auch einen Zeiger zum Vorgänger von $v$ auf diesem Weg. Wenn der Algorithmus dann bei einem Zielknoten $t$ terminiert, werden keine weiteren Knoten erweitert. Dann kann man den kürzesten Weg von $s$ nach $t$ durch Folgen der Zeiger leicht ermitteln.

Es kann jetzt verschiedene zulässige Algorithmen geben, die sich in der Reihenfolge der Erweiterung der Knoten und/oder der Anzahl der erweiterten Knoten unterscheiden. Im Folgenden werden wir einen davon beschreiben und zeigen, dass er zulässig ist. Desweiteren werden wir unter gewissen Annahmen zeigen, dass dieser Algorithmus die Informationen über den Graph optimal nutzt, d.h., dass er die kleinstmögliche Anzahl von Knoten erweitert, um den kürzesten Weg zu finden.

Um möglichst wenige Knoten zu erweitern, muss der Algorithmus in jedem Schritt den Knoten, der als nächstes erweitert wird, mit so viel Information, wie möglich, auswählen. Knoten, die offensichtlich nicht auf dem kürzesten Weg liegen können, sollten nicht erweitert werden. Andererseits dürfen Knoten, die auf einem kürzesten Weg liegen könnten, nicht ausgelassen werden, um die Zulässigkeit nicht zu gefährden.

## 6.3 Der Algorithmus

Wir wollen eine *Auswertungsfunktion* $\hat{f}(v)$ für jeden Knoten $v$ bestimmen. Diese soll so definiert sein, dass der Knoten mit dem kleinsten Wert der Funktion als

nächstes erweitert werden soll. Daraus ergibt sich der Algorithmus 1, in dem $s$ der Startknoten und $T$ die Menge der Zielknoten ist.

**Algorithm 2** A*-Algorithmus

1: Markiere $s$ als „offen“ und berechne $\hat{f}(s)$
2: Wähle den offenen Knoten $v$, dessen Wert von $\hat{f}(v)$ am kleinsten ist. Falls es kein eindeutiges Minimum gibt, wähle bevorzugt ein $t \in T$, sonst zufällig
3: **if** $v \in T$ **then**
4: Markiere $v$ als geschlossen.
5: Beende den Algorithmus.
6: **else** Markiere $v$ als geschlossen
7: Wende Nachfolgeoperator $\Gamma$ auf $v$ an
8: Berechne für jeden Nachfolger von $v$ den Wert von $\hat{f}$
9: Markiere jeden nicht geschlossenen Nachfolger als offen
10: **if** Für einen geschlossenen Nachfolger $w$ ist $\hat{f}(w)$ jetzt kleiner als bei der Schließung von $w$ **then**
11: Markiere $w$ als offen
12: **end if**
13: Gehe zu Schritt 2.
14: **end if**

Als nächstes überlegen wir uns, wie wir die Auswertungsfunktion $\hat{f}$ wählen, damit der A*-Algorithmus zulässig ist.

## 6.4 Auswertungsfunktion

Seien $f(v)$ die tatsächlichen Kosten eines kürzesten Weges *durch* $v$ von $s$ zu einem bevorzugten Zielknoten von $t$. Seien $h(v)$ die Kosten des kürzesten Wegs von $v$ zu einem bevorzugten Zielknoten von $v$, d.h.

$$h(v) = \min_{t \in T} \delta(v, t)$$

Dabei ist $\delta(v, t)$ wieder das Kürzeste-Wege-Gewicht von $v$ nach $t$.

$f(s) = h(s)$ sind die Kosten eines kürzesten Weges von $s$ zu einem bevorzugten Zielknoten von $s$. Außerdem gilt $f(v) = f(s)$ für jeden Knoten $v$ auf einem kürzesten Weg und $f(u) > f(s)$ für jeden Knoten $u$, der nicht auf einem kürzesten Weg liegt. Da wir $f(v)$ bei Ausführung des Algorithmus noch nicht kennen, wählen

wir einen Schätzer für die Funktion als unsere Auswertungsfunktion $\hat{f}(v)$. (Das erklärt auch die Bezeichnung mit dem Hut.)

Wir schreiben jetzt

$$f(v) = g(v) + h(v),$$

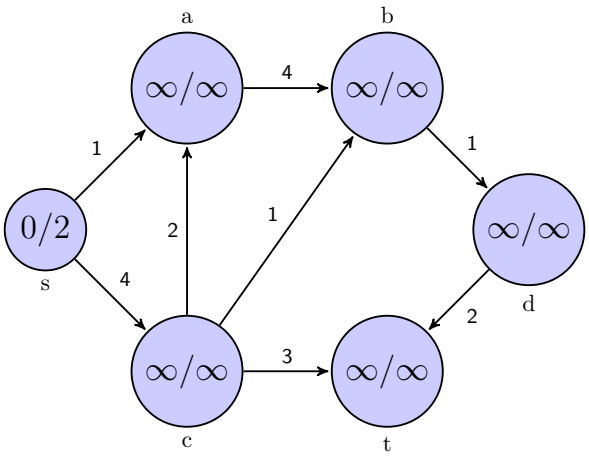
wobei $g(v)$ die tatsächlichen Kosten eines Kürzesten Weges von $s$ nach $v$ sind und $h(v)$ wie oben definiert. Um $f$ zu schätzen, brauchen wir also Schätzer für $g$ und $h$. Sei also $\hat{g}(v)$ ein Schätzer für $g(v)$. Naheliegend ist es, für $\hat{g}(v)$ die Kosten des bisher kürzesten gefundenen Weges von $s$ nach $v$ zu wählen. Dann gilt $\hat{g}(v) \geq g(v)$.

Jetzt brauchen wir noch einen Schätzer $\hat{h}(v)$ für $h(v)$. Dazu greifen wir auf die Information über das Problem zurück und versuchen so, eine Abschätzung für die Kosten des Weges von $v$ zum Ziel zu finden. Wie einleitend erwähnt, wäre das bei Orten die Luftlinie. Diese ist die kürzestmögliche Länge von irgendeiner Straße zwischen den zwei Orten. Deshalb ist sie eine untere Schranke für $h(v)$. Es sind aber auch andere Abschätzungen denkbar. Wie genau $\hat{h}$ in unserem Problem aussehen könnte, betrachten wir später, zunächst sei es irgendeine untere Schranke für $h$.

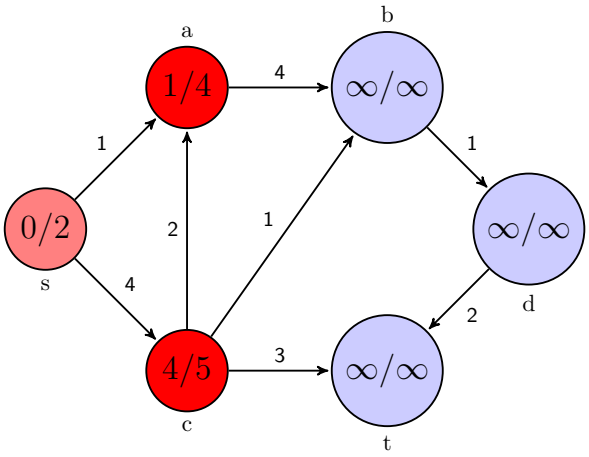
*Bemerkung* 4. Die Funktion $\hat{g}$ entspricht genau der Bewertungsfunktion $d$ beim Dijkstra-Algorithmus. Dort passiert im Wesentlichen das Gleiche wie hier, allerdings mit $\hat{h} \equiv 0$. Außerdem ist es dort nicht möglich, Knoten mehrmals zu untersuchen.

## 6.5 Beispiel

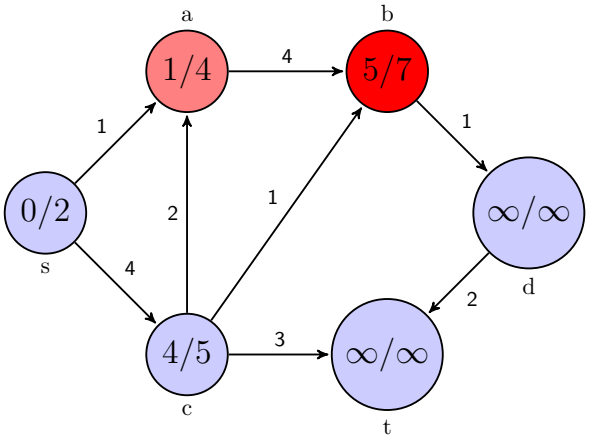
Auch hierfür wollen wir uns wieder ein kleines Beispiel anschauen. Sei dafür folgender Graph gegeben, in dem wir den kürzesten Weg von $s$ nach $t$ suchen. Die Zahlen an den Kanten bezeichnen wieder die Gewichte, in den Knoten steht die erste Zahl für die Länge des bisher kürzesten Wegs dahin (also wie beim Dijkstra) und die zweite Zahl den Wert der Bewertungsfunktion. Weiterhin sei die Information gegeben, dass die minimale Kantenlänge 1 beträgt. Deshalb wählen wir als Abschätzung für den verbleibenden Weg die Anzahl dessen Kanten.



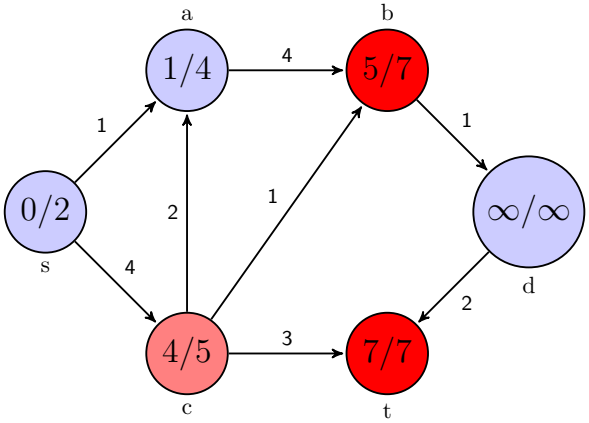

Im 1.Schritt erweitern wir natürlich $s$ und erhalten neue Wege nach $a$ und $c$. Als Bewertung für $a$ haben wir dann die Länge des Wegs dorthin, 1, plus die minimale Kantenzahl zum Ziel, 3, also insgesamt 4. Genauso erhalten wir für $c$ den Wert $4 + 1 = 5$. Bemerke, dass wir für die Abschätzung zwar den direkten Weg nach $t$ betrachten, der tatsächliche kürzeste Weg aber auch anders sein könnte.




Da die Bewertung für $a$ am niedrigsten ist, erweitern wir jetzt diesen Knoten und erhalten für $b$ einen kürzesten Weg der Länge 5 und eine Bewertung von 7.




Nun ist der Knoten $c$ an der Reihe und bei seiner Erweiterung erreichen wir das Ziel $t$ mit einem Weg der Länge 7. Da es keinen restlichen Weg gibt, ist die Heuristik 0 und die Bewertung ebenfalls 7.




Der A*-Algorithmus ist jetzt fertig, denn es gibt keine kleinere Bewertung als 7 und somit kann es auch keinen kürzeren Weg geben. Der Dijkstra müsste jetzt noch die Knoten $b$ und $d$ erweitern, um dies herauszufinden.

## 6.6 Zulässigkeit

Wir haben jetzt die Auswertungsfunktion

$$\hat{f}(v) = \hat{g}(v) + \hat{h}(v)$$

mit $\hat{g}(v)$ und $\hat{h}(v)$ wie eben beschrieben.

**Lemma 3.** *Für jeden nicht geschlossenen Knoten $v$ und für jeden kürzesten Weg $P$ von $s$ nach $v$ gibt es einen offenen Knoten $u$ auf $P$ mit $\hat{g}(u) = g(u)$*

*Beweis.* siehe [12] □

Das benutzen wir für folgende Aussage:

**Lemma 4.** *Gelte $\hat{h}(v) \leq h(v)$ für alle $v$, sei A* noch nicht beendet. Dann gibt es für jeden Kürzesten Weg $P$ von $s$ zu einem bevorzugten Zielknoten von $s$ einen offenen Knoten $u$ auf $P$ mit $\hat{f}(u) \leq f(s)$*

*Beweis.* Nach dem Lemma 3 gibt es einen offenen Knoten $u$ auf $P$ mit $\hat{g}(u) = g(u)$. Also gilt

$$\hat{f}(u) = \hat{g}(u) + \hat{h}(u) = g(u) + \hat{h}(u) \leq g(u) + h(u) = f(u)$$

Da $P$ ein Kürzester Weg ist, gilt $f(u) = f(s)$ für alle $u \in P$. □

Jetzt können wir die Zulässigkeit zeigen:

**Satz 2.** *Falls $\hat{h}(v) \leq h(v)$, ist A* zulässig*

*Beweis.* (aus [12]) Wir schauen uns die drei Fälle an, wo A* nicht zulässig ist, also wo A* nicht damit endet, einen kürzesten Weg zu einem bevorzugten Zielknoten von $s$ zu finden und wollen diese zu einem Widerspruch führen. Dies sind: Ende an einem Nicht-Zielknoten, kein Ende des Algorithmus, Ende an einem Zielknoten ohne minimale Kosten.

*Fall 1*: Ende an Nicht-Zielknoten
Dieser Fall widerspricht der Beendigungsbedingung in Zeile 3-5 des Algorithmus, kann also ausgeschlossen werden.

*Fall 2*: kein Ende
Sei $t$ ein bevorzugter Zielknoten von $s$, der vom Start in endlich vielen Schritten mit minimalen Kosten $f(s)$ erreicht werden kann. Die Gewichte auf jedem Knoten sind positiv, es gibt also ein $\epsilon > 0$, das kleiner als jedes Gewicht ist. Für jeden Knoten $v$, der mehr als $M := f(s)/\epsilon$ Schritte von $s$ entfernt ist, gilt $\hat{f}(v) \geq \hat{g}(v) \geq g(v) >$

$M\epsilon = f(s)$. So ein Knoten $v$ wird nie erweitert, denn nach dem Lemma 4 gibt es einen offenen Knoten $u$ auf eine Kürzesten Weg, so dass $\hat{f}(u) \leq f(s) < \hat{f}(v)$. Deshalb wird A* in Schritt 2 $u$ statt $v$ auswählen. Wir wissen also, dass alle Knoten, die erweitert werden, in $M$, also endlich vielen Schritten erreicht werden können, es kann also nur endlich viele geben. Ein nicht vorhandenes Ende von A* kann also nur noch durch ein wiederholtes Öffnen der gleichen Knoten entstehen. Sei $X(M)$ die Menge der Knoten, die in $M$ Schritten von $s$ erreichbar sind, sei $x(M) = \#(X(M))$. Jeder Knoten $v$ in $X(M)$ kann nur endlich oft geöffnet werden, sagen wir $y(v, M)$ mal, denn es gibt nur eine endlich Anzahl Wege von $s$ nach $v$, die nur durch Knoten aus $X(M)$ gehen. Sei dann

$$y(M) = \max_{v \in X(M)} y(v, M)$$

die maximale Anzahl von Öffnungen irgendeines Knotens. Daraus folgt, dass nach maximal $x(M)y(M)$ Erweiterungen alle Knoten in $X(M)$ für immer geschlossen sind. Da außerhalb von $X(M)$ keine Knoten erweitert werden, muss A* ein Ende haben.

*Fall 3*: Ende ohne minimale Kosten
Angenommen, A* würde an einem Zielknoten $t$ enden, wo $\hat{f}(t) = \hat{g}(t) > f(s)$ gelten würde. Nach dem Korollar würde dann direkt vor dem Ende ein offener Knoten $u$ auf einem Kürzesten Weg mit $\hat{f}(u) \leq f(s) < \hat{f}(t)$ existieren. Deshalb wäre dann $u$ statt $t$ für die Erweiterung ausgewählt worden, was der Annahme widerspricht. □

Wir haben jetzt also schon ein gutes Ergebnis, nämlich, dass der A*-Algorithmus funktioniert, wenn wir $\hat{h}$ so wählen, wie wir es uns überlegt hatten, als untere Schranke für $h$.

*Bemerkung* 5. Hart et al. [12] zeigen zudem noch die Optimalität des A*-Algorithmus, in dem Sinne, dass jeder andere zulässige Algorithmus, der die gleichen Informationen über das Problem erhält, mindestens die Knoten untersuchen muss, die auch A* erweitert.

*Bemerkung* 6. Der A*-Algorithmus bietet gegenüber dem Dijkstra-Algorithmus

also zwei Vorteile:

1. Durch die Abschätzung der weiteren Kosten bis zum Zielknoten werden u.U. weniger Knoten untersucht und dadurch reduziert sich die Laufzeit. Wie groß dieser Vorteil ist, hängt sehr davon ab, wie gut die benutzte Heuristik $\hat{h}$ ist und dies wiederum davon, wie viel Information über den Graphen und die Problemstellung bekannt ist.

2. Im Gegensatz zum Dijkstra-Algorithmus sind hier auch negative Gewichte möglich, denn in dem Beweis der Zulässigkeit haben wir die Positivität nur dafür benutzt, um zu zeigen, dass nur endlich viele Knoten erweitert werden. Da wir aber für Graphen nur endliche Knotenmengen zulassen, ist dies immer noch erfüllt. Anschaulich erklärt sich dies dadurch, dass insbesondere auch

   $$\hat{h}(v) \leq h(v) < 0$$

   möglich ist, das heißt wenn die verbleibende Distanz negativ ist, muss die Heuristik so gewählt sein, dass die Ungleichung weiterhin gilt, also auch entsprechend negativ sein. In dem Beispiel aus Bemerkung 3 müsste $\hat{h}(b) \leq -4$ gelten und somit $\hat{f}(b) \leq 1$. Dadurch ist sichergestellt, dass $b$ noch erweitert wird, bevor der Algorithmus terminiert. Wenn es negative Kreise gibt, gibt es keine Funktion $\hat{h}(v) \leq h(v)\ \forall v \in V$, da $h(v)$ dann den Wert $-\infty$ annimmt. Insofern ist er A*-Algorithmus für diesen Fall nicht anwendbar, was sinnvoll ist, weil es dann auch keine Lösung gibt.

# 7 Wahl der Heuristik

Kehren wir jetzt zum paarweisen Alignment und dem Ansatz von Needleman und Wunsch zurück. Wie wir gerade gesehen haben, brauchen wir für den A*-Algorithmus eine Funktion $\hat{h}$ mit $\hat{h}(v) \leq h(v)\ \forall v \in V$. Das heißt, wir müssen eine Abschätzung der Kosten bis zum Zielknoten $(n, m)$ finden.

Neben den Kantengewichten haben wir noch die Information über den „rechteckigen“ Aufbau des Graphen. Diesen nutzen wir jetzt mit folgenden Überlegungen aus ($A$ sei wieder das zugrundeliegende Alphabet):

- An der Stelle $(i, j)$ brauchen wir noch mindestens $(n - i)$ Kanten, um in die letzte ($n$-te) Zeile zu kommen.
- An der Stelle $(i, j)$ brauchen wir noch mindestens $(m - j)$ Kanten, um in die letzte ($m$-te) Spalte zu kommen.
- Daraus folgt: An der Stelle $(i, j)$ brauchen wir noch mindestens $\max\{n - i, m - j\}$ Kanten, um nach $(m, n)$ zu kommen.
- An der Stelle $(i, j)$ können wir noch maximal $\min\{n - i, m - j\}$ diagonale Kante benutzen , um nach $(n, m)$ zu kommen, da jede diagonale Kante den Zeilenindex und den Spaltenindex um genau 1 erhöht.
- Sei jetzt $d_{min} := \min_{a,b \in A \times A}\ d(a, b)$ das kleinstmögliche Gewicht auf einer Diagonalen.
- **Fall 1**: $2 * d_{Gap} \geq d_{min}$, d.h. der Weg über die Diagonale kann besser sein als der über zwei Lücken-Kanten.
  - Die Abschätzung für den kürzesten Weg benutzt dann möglichst viele diagonale Kanten, also $\min\{n - i, m - j\}$ Stück.
  - Es bleiben also noch $\max\{n - i, m - j\} - \min\{n - i, m - j\} = |(n - i) - (m - j)|$ horizontale oder vertikale Kanten, für die wir Lücken einfügen müssen.
  - Die Heuristik ergibt sich dann als

$$\hat{h}(i, j) = \min\{n - i, m - j\} * d_{min} + |(n - i) - (m - j)| * d_{Gap}$$

- **Fall 2**: $2 * d_{Gap} < d_{min}$, d.h. der Weg über 2 Lücken-Kanten kann günstiger sein als die Diagonalkante.
  - Der Weg wird dann nur über die Gap-Kanten abgeschätzt
  - Dieser besteht dann aus $n-i$ Schritten nach unten und $m-j$ Schritten nach rechts.
  - Wir erhalten dann also

$$\hat{h}(i,j) = ((m-i) + (n-j)) * d_{Gap}$$

Nach dieser Konstruktion wissen wir jetzt:

**Satz 3.** *Die Heuristik*

$$\hat{h}(i,j) = \begin{cases} \min\{n-i, m-j\} * d_{min} + |(n-i) - (m-j)| * d_{Gap}, & \textit{falls } 2 * d_{Gap} \geq d_{min} \\ ((m-i) + (n-j)) * d_{Gap}, & \textit{sonst} \end{cases}$$

*erfüllt die Bedingung* $\hat{h}(i,j) \leq h(i,j)\ \forall (i,j) \in [0,n] \times [0,m]$

*Bemerkung* 7. Im Normalfall wird in den Anwendungen nur Fall 1 eintreten, denn eine Übereinstimmung der zugeordneten Zeichen in dem Alignment sollte immer günstiger sein als das Einfügen einer Lücke.

Mit dieser Heuristik können wir den A*-Algorithmus jetzt auf das paarweise Alignment anwenden und untersuchen, wie groß die Verbesserung zum Dijkstra-Algorithmus ist. Dazu werden wir im praktischen Teil dieser Arbeit beide Algorithmen auf Beispiel-Sequenzen anwenden und die Laufzeiten vergleichen.

Die Güte der Heuristik, also die Abweichung der tatsächlichen $h(v)$ von den geschätzten $\hat{h}(v)$, und damit deren Effektivität hängt natürlich stark von den untersuchten Instanzen ab. Wenn beispielsweise alle diagonalen Kanten das gleiche Gewicht hätten, würde der Schätzer dem wahren Wert entsprechen. Wenn das Gewicht einer diagonalen Kante deutlich kleiner ist als das der anderen, ist eine Schätzung vermutlich viel zu niedrig.

Um diesen Effekt etwas abzufangen, könnte man die Definition von $d_{min}$ noch etwas abändern und diesen Wert von $(i,j)$ abhängig machen:

1. Man betrachtet nicht alle möglichen $(a, b) \in A^2$, sondern nur diejenigen Paare, die rechts unten von dem gerade untersuchten Knoten in der Matrix wirklich vorkommen. Die Abschätzung bleibt offensichtlich gültig, denn der wirkliche kürzeste Weg hat ja auch nur diese Paare zur Verfügung. Bei einigen Instanzen könnte dadurch die Heuristik verbessert werden. Allerdings müssten wir dafür zunächst alle Knoten einmal betrachten, damit bekannt ist, welche Paare wirklich auftreten. Dies widerspricht gerade der Idee, nicht alle Knoten zu betrachten.

2. Man ermittelt am Anfang für alle Zeilen und alle Spalten das Minimum der Kosten und nennt diese $d^k_{min}$ für alle Zeilen $k \in [1, n]$ und $d^l_{min}$ für alle Spalten $l \in [1, m]$. Dann kann man folgende Heuristik aufstellen:

$$\hat{h}(i,j) = \begin{cases} \sum_{k=i+1}^{n} d^k_{min} + |(n-i)-(m-j)| * d_{Gap}, & \text{falls } n-i \leq m-j \\ \sum_{l=j+1}^{m} d^l_{min} + |(n-i)-(m-j)| * d_{Gap}, & \text{sonst} \end{cases}$$

   **Lemma 5.** *Dies ist eine zulässige Heuristik, falls* $2 * d_{Gap} \geq d_{min}$.

   *Beweis.* Zunächst ist es wegen der Bedingung $2 * d_{Gap} \geq d_{min}$ optimal, möglichst viele Diagonalen zu benutzen. Falls $n-i \leq m-j$ gilt, benutzt der optimale Weg also $n-1$ Diagonalen, wobei jeweils eine aus den Zeilen $(i+1)$ bis $n$ kommt. D.h., der kürzeste Weg hat mindestens die Minimalkosten aus jeder dieser Zeilen. Dazu kommen noch die Lücken-Kosten, die die Differenz zwischen $(n-i)$ und $(m-j)$ auffüllen. Für $n-i > m-j$ gilt dasselbe für die Spalten, es muss aus jeder von $(j+1)$ bis $m$ genau eine Diagonalkante benutzt werden. □

   Auch hier ist bei entsprechenden Instanzen wieder eine Verbesserung der Heuristik möglich. Allerdings ist der Nachteil, dass wieder alle Knoten betrachtet werden müssen, und am Anfang viele Minima berechnet werden müssen. Eine bessere Laufzeit ist also auch hier nicht garantiert.

3. Man kann 1. und 2. natürlich auch kombinieren, also die spalten- bzw. zeilenweise Minima der Restmatrix berechnen, die Problematik bleibt die gleiche.

# 8 Implementierung

Nun wollen wir den A*-Algorithmus praktisch testen und ihn mit dem Dijkstra-Algorithmus vergleichen. Dafür implementieren wir die Algorithmen in der Programmiersprache C++ und messen dann für Beispielinstanzen die Laufzeit.

## 8.1 Dijkstra

Für das Programm brauchen wir zunächst eine Möglichkeit dafür, einen Graphen zu speichern. Ein Framework hierfür wurde mir von Prof. Westphal zur Verfügung gestellt und dann von mir für diesen Zweck angepasst. Es besteht aus Klassen für die Knoten, die Kanten sowie den Graphen.

| **Klasse** | **Informationen** | **Funktionen** |
|---|---|---|
| Node | Name<br>inzidente Kanten<br>inzidente Knoten<br>Knotenbewertung<br>Vorgänger | Erzeugung des Knoten<br>Zufügen von inz. Kanten<br>Zufügen von inz. Knoten |
| Edge | Name<br>Gewicht<br>Startknoten<br>Zielknoten | Erzeugung der Kante |
| Graph | Name<br>enthaltene Knoten<br>enthaltene Kanten | Erzeugung des Graphen<br>Entfernung des Graphen<br>Knoten hinzufügen<br>Kante hinzufügen |

Tabelle 3: Benutzte Klassen

Im Hauptteil des Programms werden dann zunächst der Graph, die Knoten und die Kanten erzeugt und einander zugeordnet. Hier müssen wir den Code an den gewünschten Graphen anpassen und die Zuordnung der Kanten und Knoten sowie die Gewichte dementsprechend eintragen.

Danach kann der eigentliche Dijkstra-Algorithmus beginnen. Dabei legen wir zuerst den Startknoten fest und initialisieren eine leere Knotenmenge $Q$. In einer Schleife werden dann alle Knoten des Graphen in $Q$ eingefügt. Gleichzeitig werden ihre Bewertungen auf $\infty$ gesetzt. Dann wird die Bewertung des Startknoten auf

0 gesetzt. Nachdem eine weitere Knotenmenge $S$ eingerichtet wurde, beginnt die Ausführung in einer Schleife, die solange läuft, bis $Q$ leer ist.

Darin wird zunächst das Minimum der Knoten im Bezug auf ihre Bewertungen gesucht. Das geschieht in einer trivialen Art, indem alle Knoten durchlaufen werden, sich immer das aktuelle Minimum gemerkt und falls es einen kleineren Wert gibt, aktualisiert wird. Der Knoten $u$ mit dem Minimum wird dann aus $Q$ entfernt und in $S$ eingefügt und untersucht. Dazu werden in einer Schleife zu $u$ inzidenten Knoten $v$ betrachtet und die dazugehörigen Kanten ermittelt. Dann wird entsprechend der Idee des Algorithmus die Knotenbewertung von $v$ aktualisiert, falls dieser über $u$ schneller erreicht wird als zuvor. In dem Fall wird $u$ als der Vorgänger von $v$ gespeichert.

Nachdem dieses in der großen Schleife für alle Knoten passiert ist, können wir das Ergebnis anzeigen lassen. Dazu lassen wir uns in einer Schleife für alle Knoten in $S$ den Vorgänger ausgeben.

## 8.2 A*

Zunächst fügen wir in die Knotenklasse eine zusätzliche Float-Variable ein, die die aktuelle heuristische Bewertung darstellt. Die anderen Klassen bleiben unverändert.

In der Hauptfunktion muss bei der Konstruktion des Graphen jedem Knoten auch seine heuristische Bewertung vorgegeben werden. Der Zielknoten sollte dabei sinnvollerweise den Wert 0 erhalte. In der Minimumsuche wird die Heuristik berücksichtigt, indem statt den Knotenbewertungen die Summen aus Knotenbewertung und heuristischer Bewertung verglichen werden. Ansonsten wird das Minimum genauso gesucht. Auch der weitere Algorithmus bedarf keiner Veränderung, da bei der Erweiterung der Knoten die Heuristik keine Rolle spielt. Es wird lediglich eine Abbruchbedingung eingefügt für den Fall, dass der ausgewählte Knoten bereits der Zielknoten ist.

## 8.3 Anwendung auf Pairwise Sequence Alignment

Um das Programm jetzt für unsere Untersuchungen zum Sequenz-Alignment zu benutzen sind einige Änderungen notwendig. In der Knoten-Klasse werden statt

des Namens jetzt zwei ganzzahlige $x$- und $y$-Koordinaten gespeichert. Dies bietet sich an, weil wir die Knoten auf dieser Grundlage benutzen wollen. Genauso wird der Vorgänger im kürzesten Weg jetzt über seine Koordinaten gespeichert. Außerdem ergänzen wir einen Vektor mit den Knoten, von denen eine Kante zum jeweiligen Knoten führt. In der Kanten-Klasse muss der Konstruktor so angepasst werden, dass auch dieser neue Vektor sinnvoll gefüllt wird.

Vor der Hauptfunktion wird jetzt die Kostenfunktion implementiert. Dort können die gewünschten Kosten für alle möglichen Paare von Zeichen eingegeben werden.

In der Hauptfunktion selbst wird jetzt der Graph nicht mehr direkt formuliert, sondern aus den Sequenzen konstruiert. Dazu werden diese zunächst eingegeben, dann deren Größe bestimmt und ein entsprechend großes Array erzeugt, dass der Needleman/Wunsch-Matrix entspricht. Außerdem müssen noch die Lücken-Kosten angegeben werden. Dann wird in zwei Schleifen das Array mit den Knoten gefüllt und die Kanten gemäß der Idee des Needleman/Wunsch-Graphen und unter Berücksichtigung der Kostenfunktion erzeugt.

Anschließend kann der gewünschte Algorithmus darauf angewendet werden. Für die Auswertung werden Zähler für die Anzahl der untersuchten Knoten bzw. Kanten sowie eine Zeitmessung zugefügt, deren Werte werden am Ende ausgegeben. Als Zielknoten, also als Abbruchspunkt benutzt man den letzten Eintrag des Arrays. Abschließend wird über die Werte der Vorgänger-Variablen der tatsächlich kürzeste Weg konstruiert und danach das entsprechende Alignment ausgegeben.

Beim A*-Algorithmus muss natürlich zusätzlich noch die Heuristik erzeugt werden. Dazu muss zunächst neben der Kostenfunktion auch die minimale Kantenbewertung eingegeben werden. Damit kann dann bei der Erzeugung des Graphen die heuristische Bewertung einfach anhand der gewünschten Funktion $\hat{h}$ in Abhängigkeit von den Koordinaten des Knotens berechnet werden und dann mit in dem Array gespeichert werden.

## 8.4 Effizientere Minimum-Suche

Um die Suche des Minimums effizienter zu gestalten, wollen wir folgende Idee nutzen. Statt jedes Mal neu nach dem kleinsten Element zu suchen, wollen wir ein Feld benutzen, in dem die Elemente gemäß ihrer aktuellen Bewertung sortiert

sind. Dann ist das Minimum immer der erste (bzw. der erste noch nicht benutzte) Eintrag in diesem Feld und kann einfach benutzt werden. Dabei nutzen wir aus, dass zu Beginn des Algorithmus die Knotenbewertungen für alle Knoten, außer den Startknoten $\infty$ sind, d.h. am Anfang ist noch keine Sortierung notwendig. Im Verlauf müssen dann jeweils einzelne Knoten neu einsortiert werden, höchstens so oft wie die Anzahl der Kanten.

In der Implementierung führen wir dazu neue Felder mit den Knoten und ihren jeweiligen Bewertungen ein, die immer mitgeführt und aktualisiert werden, wenn sich die Bewertung ändert. Bei der Auswahl des nächsten zu erweiternden Knoten, wird jetzt nicht mehr das Minimum gesucht, sondern der jeweils $z$-te Eintrag des Feldes, wenn wir im $z$-ten Durchlauf der Schleife sind, also den $z$-ten Knoten auswählen. Das funktioniert, weil das Feld stets sortiert gehalten wird, dadurch steht dann immer das Minimum an der gewünschten Stelle. Diese Sortierung erreichen wir, indem wir in dem Fall, dass sich die Bewertung eines Knotens ändert (also wenn ein kürzerer Weg gefunden wird), auch das Feld entsprechend aktualisieren. Das geschieht, indem man den Wert des veränderten Knotens mit seinen Vorgängern vergleicht und dann an der richtigen Stelle einsortiert, dementsprechend werden alle Knoten die dazwischen liegen, in dem Feld um eine Stelle nach hinten verschoben. Nach der Bearbeitung eines Knoten ignorieren wir diesen, indem wir das Feld nur noch ab der darauf folgenden Stelle betrachten.

Beim A*-Algorithmus ist zu beachten, dass in der Bewertung auch die heuristische Bewertung enthalten sein muss. Diese ändert sich zwar nicht, hat aber einen Einfluss auf die Reihenfolge im Heap.

## 8.5 Topologische Sortierung

Als Vergleich zu unseren Algorithmen implementieren wir auch den Ansatz, die Knoten einfach der Reihe nach „abzuarbeiten“ und somit die topologische Sortierung zu nutzen (siehe Abschnitt 5.3).

Dabei durchlaufen wir das Knoten-Array in zwei Schleifen (zeilenweise und spaltenweise) und verzichten dafür auf die Suche nach dem Minimum. Die Knoten werden dann wie bisher behandelt und die Bewertungen entsprechend aktualisiert.

# 9 Ergebnisse

In diesem Kapitel wollen wir jetzt die zuvor entwickelten Algorithmen testen und vergleichen. Wir haben:

- Dijkstra-Algorithmus (D)
- A*-Algorithmus (A*)
- Dijkstra-Algorithmus mit verbesserter Minimum-Suche (DMIN)
- A*-Algorithmus mit verbesserter Minimum-Suche (A*MIN)
- Topologische Sortierung (TOP)

## 9.1 Technische Voraussetzungen

Der für die Ausführung der Programme benutzte Computer hat folgende Systemeigenschaften:

| | |
|---|---|
| Prozessor | Pentium Dual-Core CPU T4200 @ 2.00 GHz |
| Arbeitsspeicher | 3,00 GB |
| Betriebssystem | Windows Vista (32 Bit) |

Tabelle 4: Systemeigenschaften

Zum Kompilieren des C++-Codes wurde die GNU Compiler Collection(GCC) von MinGW benutzt.

## 9.2 Beobachtung

Bei den Instanzen können wir die Sequenzen, insbesondere deren Länge, sowie die Kostenfunktion variieren. Wir werden zunächst mit folgender festen Kostenfunktion die Laufzeiten und die Anzahl der betrachteten Kanten der Algorithmen

vergleichen.

$$\begin{cases} d(x,y) = 1 & \text{falls } x = y \\ d(A,B) = d(B,A) = 2, & \\ d(A,C) = d(C,A) = 6, & \\ d(B,C) = d(C,B) = 7, & \\ d_{Gap} = 10 & \end{cases}$$

In der Tabelle 5 sind die durchschnittlichen Werte für verschiedene, zufällige Sequenzen mit Längen $n$ und $m$ angegeben.

| n | m | | D | A* | DMIN | A*MIN | TOP |
|---|---|---|---|---|---|---|---|
| 10 | 10 | Kanten | 222 | 99 | 234 | 105 | 320 |
| | | Knoten | 79 | 34 | 83 | 36 | 121 |
| | | Laufzeit | 0 | 0 | 0 | 0 | 0 |
| 40 | 60 | Kanten | 5206 | 2121 | 5221 | 2139 | 7300 |
| | | Knoten | 1767 | 710 | 1772 | 716 | 2501 |
| | | Laufzeit | 0.218 | 0.11 | 0.062 | 0.032 | 0 |
| 100 | 110 | Kanten | 13829 | 3598 | 13823 | 3646 | 33210 |
| | | Knoten | 4620 | 1201 | 4618 | 1217 | 11211 |
| | | Laufzeit | 2.231 | 0.733 | 0.359 | 0.14 | 0.016 |
| 200 | 200 | Kanten | 53427 | 17074 | 53408 | 17182 | 120400 |
| | | Knoten | 17825 | 5693 | 17819 | 5729 | 40401 |
| | | Laufzeit | 32.588 | 12.854 | 5.959 | 2.324 | 0.078 |

Tabelle 5: Ergebnisse für feste Kostenfunktion

Um einen möglichen Einfluss der Kostenfunktion zu erkennen, halten wir in der nächsten Versuchreihe die Länge der Sequenzen fest bei $(100, 110)$ und variieren die Kostenfunktion. Wir testen die Modelle aus Tabelle 6.

Die Ergebnisse hierfür stehen in Tabelle 7.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $d(x,y),\ x=y$ | 1 | 1 | 0 | 1 | 1 |
| $d(A,B)$ | 2 | 1.2 | 0.5 | 10 | 3 |
| $d(A,C)$ | 6 | 1.4 | 0.5 | 20 | 3 |
| $d(B,C)$ | 7 | 1.1 | 0.5 | 30 | 4 |
| $d_{Gap}$ | 10 | 1.5 | 0.7 | 50 | 1.2 |

Tabelle 6: Kostenmodelle

| Modell | | D | A* | DMIN | A*MIN | TOP |
|---|---|---|---|---|---|---|
| 1 | Kanten | 13829 | 3598 | 13823 | 3646 | 33210 |
| | Knoten | 4620 | 1201 | 4618 | 1217 | 11211 |
| | Laufzeit | 2.231 | 0.733 | 0.359 | 0.14 | 0.016 |
| 2 | Kanten | 28227 | 2860 | 28214 | 2899 | 33210 |
| | Knoten | 9446 | 955 | 9434 | 968 | 11211 |
| | Laufzeit | 3.244 | 0.608 | 0.628 | 0.114 | 0.031 |
| 3 | Kanten | 10658 | 4210 | 13683 | 4268 | 33210 |
| | Knoten | 3559 | 1405 | 4570 | 1424 | 11211 |
| | Laufzeit | 1.778 | 0.858 | 0.483 | 0.164 | 0.031 |
| 4 | Kanten | 9550 | 3199 | 9655 | 3259 | 33210 |
| | Knoten | 3191 | 1068 | 3226 | 1088 | 11211 |
| | Laufzeit | 1.699 | 0.687 | 0.281 | 0.141 | 0.032 |
| 5 | Kanten | 33210 | 7180 | 33210 | 7314 | 33210 |
| | Knoten | 11211 | 2359 | 11211 | 2422 | 11211 |
| | Laufzeit | 3.37 | 1.388 | 0.767 | 0.299 | 0.031 |

Tabelle 7: Ergebnisse für verschiedene Kostenfuktionen

## 9.3 Analyse

Was können wir jetzt aus diesen Zahlen ablesen? Betrachten wir zunächst Dijkstra- und A*-Algorithmus. Wir sehen, dass beim A* weniger Knoten untersucht werden als beim Dijkstra, der Faktor schwankt zwischen 2 und 4, im Kostenmodell 2 untersucht A* sogar nur ca. 10% der Knoten vom Dijkstra. Dadurch verringert sich auch die Laufzeit um einen ähnlich hohen Faktor. Durch die Einführung der Heuristik können wir also schon einiges an Rechenzeit einsparen, wie viel, hängt sehr von der Kostenfunktion ab. Bei Kostenfunktion 2 ist der Effekt besonders groß, weil zum einen die Abschätzung recht gut ist, da die Abweichung von den minimalen Kosten maximal 0.4 ist. Zum anderen sind die Lücken-Kosten relativ gering, so dass der Dijkstra besonders viele Knoten untersuchen muss. Weiterhin kann man beobachten, dass die Größe der Sequenzen keinen großen Einfluss auf den Effekt des A*-Algorithmus hat. Für noch größere Sequenzen ist also kein höherer Nutzen der Heurisik zu erwarten.

Schauen wir uns jetzt an, wie sich die effizientere Minimum-Suche auswirkt. Zunächst stellen wir fest, dass die Anzahl der untersuchten Knoten ungefähr gleich groß bleibt. Dies konnte man erwarten, weil sich lediglich die Strategie ändert, wie man das Minimum findet, aber die ausgewählten Knoten bleiben in etwa die selben. Abweichungen lassen sich dadurch erklären, dass das Minimum nicht eindeutig ist. Trotzdem verkürzt sich die Laufzeit, dies zeigt, dass die neue Suchmethode tatsächlich effizienter ist. Der Verbesserungsfaktor schwankt dabei zwischen 3 und 6 und ist beim Dijkstra- und A*-Algorithmus ungefähr gleich groß. Ein größerer Effekt bei größeren Instanzen ist auch hier nicht zu erkennen. Auch scheint bei diesem Vergleich das Kostenmodell kaum einen Einfluss zu haben.

Wenn man sich dann den kumulierten Effekt von A*-Algorithmus und effizienter Minimum-Suche anschaut, sieht man dass wir uns bei den größeren Instanzen mindestens um den Faktor 10 bei der Laufzeit verbessern, im Kostenmodell 2 sogar fast um den Faktor 30.

Um zu wissen wie gut wir wirklich sind, müssen wir uns aber noch dem Vergleich mit der topologischen Sortierung unterziehen. Wir sehen, dass diese noch einmal besser ist, obwohl dabei deutlich mehr Knoten (nämlich alle) untersucht werden. Der Vorteil kommt also ausschließlich daher, dass keine Suche des Mini-

mums vonnöten ist. Bei einem für unser Verfahren günstigen Kostenmodell kommen wir immerhin auf den Faktor 4 daran, allerdings sehen wir auch, dass bei größeren Instanzen das Ausnutzen der topologischen Sortierung doch deutlich (hier Faktor 30) besser ist.

## 9.4 Fazit

Wir haben es geschafft, den ursprünglichen Dijstra-Algorithmus so zu verbessern, dass die Laufzeit deutlich kürzer geworden ist. Insbesondere hat der A*-Algorithmus mit der von uns entwickelten Heuristik einen merklichen positiven Einfluss auf die Performance des Vefahrens. Für alle untersuchten Konstellationen konnte eine Verbesserung beobachtet werden. Dass diese maßgeblich von der Kostenfunktion abhängt, war zu erwarten. Während der Untersuchungen konnten wir feststellen, dass ein beträchtlicher Teil der Rechenzeit dafür benötigt wird, bei jeder Auswahl des Knotens das Minimum zu bestimmen. Daraufhin haben wir uns eine effizientere Methode zu dessen Ermittlung überlegt, die sich praktisch ebenfalls bewährt hat und die Laufzeit noch einmal verkürzen konnte.

Trotzdem können wir noch nicht ganz zufrieden sein, denn das Verfahren, die Knoten der (topologisch sortierten) Reihe nach abzuarbeiten, ist immer noch besser. Zwar kommen wir teilweise schon recht nahe, aber für größere Instanzen, und diese werden vermutlich in der Anwendung benötigt, sind wir mit unserem A*-Algorthmus noch deutlich unterlegen. Diese Problematik hatten wir schon in Abschnitt 5.3 angedeutet.

Was könnte man also noch tun? Zum einen wäre es möglich, noch eine bessere Heuristik zu finden. Wie schon in Kapitel 7 dikustiert, bräuchte man dafür aber wahrscheinlich eine weitere aufwändige Minimum-Suche. Außerdem wäre der mögliche Erfolg davon auch wieder sehr von der jeweiligen Instanz abhängig. Da der Nachteil vom unseren Verfahren nicht in der Anzahl der untersuchten Knoten, sondern im Suchen des Minimums liegt, bietet es sich an, dort nach Verbesserungsmöglichkeiten zu suchen. Mit der Datenstruktur Heap (siehe z.B. [16]) ist es möglich, das Feld mit den Knoten effizient zu sortieren, und dann das Minimum in konstanter Zeit zu erhalten. Das Sortieren und Aufrechterhalten der Sortierung benötigt eine Laufzeit von $\mathcal{O}(\log k)$, wenn $k$ die Größe des Heaps ist, in unserem

Fall also $\mathcal{O}(\log mn)$. Da wir im schlimmsten Fall für jede Kante den Heap aktualisiern müssen, weil sich Kantenbewertungen ändern, hätten wir eine Gesamtlaufzeit von $\mathcal{O}(mn \log(mn))$, die immer noch größer ist als die $\mathcal{O}(mn)$, die wir bei der topologischen Sortierung ermittelt hatten. Dieser Nachteil könnte dadurch ausgeglichen werden, dass entsprechend weniger Knoten untersucht werden, also der Verbesserungsfaktor im Bereich $\mathcal{O}(\log(mn)$ liegt. Zumindest diese Größenordnung haben wir in unseren Versuchen auch erreicht, problematisch ist bloß, dass sich mit größerer Sequenzlänge keine Verbesserung eingestellt hat. Dies wäre aber nötig, um den Nachteil der Minimum-Suche stets ausgleichen zu können.
Wir haben also durchaus noch Hoffnung, dass man mit dem von uns eingeschlagenen Weg das Finden des paarweisen Sequenz-Alignment verbessern kann, haben es allerdings noch nicht geschafft und mussten feststellen, dass das eigentlich simplere Verfahren schon sehr gute Ergebnisse liefert und es deshalb fraglich ist, ob wirklich noch eine Verbesserung möglich ist.

# 10 Anhang

# Literatur

[1] Bellman, Richard Ernest: Dynamic Programming. Reprint. Mineola, New York: Courier Dover Publications, 2003

[2] Bungartz, Hans-J. ; Zimmer, Stefan ; Buchholz, Martin ; Pflüger, Dirk: Modellbildung und Simulation: Eine anwendungsorientierte Einführung. Berlin, Heidelberg: Springer 2009

[3] Cazenave, Tristan: Approximate Multiple Sequence Alignment with A-star, Université Paris 8

[4] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L.: Introduction To Algorithms. 2. Aufl.. Cambridge: MIT Press, 2001.

[5] Diestel, Reinhard: Graph Theory. 3rd ed.. Berlin, Heidelberg: Springer, 2005.

[6] Dijkstra, Edsger W.: A note on two problems in connexion with graphs, Numerische Mathematik 1, 1959

[7] Dreyfus, Stuart: Richard Bellman on the Birth of Dynamic Programming. Operations Research, Vol.50, No.1, Jan-Feb 2002, pp. 48-51

[8] Fauster, Janosch: Ein metaheuristischer Ansatz für das Multiple Sequence Alignment Problem. TU Wien, 2004

[9] Giegerich, Robert ; Wheeler, David: Pairwise Sequence Alignment. 1996

[10] Gusfield, Dan: Algorithms on Strings, Trees and Sequences. New York: Cambridge University Press, 1997

[11] Hansen, Andrea: Bioinformatik: Ein Leitfaden für Naturwissenschaftler. Basel, Boston, Berlin: Birkhäuser, 2004

[12] Hart, Peter E. ; Nilsson, Nils J. ; Raphael , Bertram: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions of systems science and cybernetics, Vol. SSC-4, No.2, 1968

[13] Hülsmann, Michael H.: Entwurf und Implementierung eines Routing-Algorithmus... 1.Aufl. 2008. Norderstedt: Grin Verlag, 2008

[14] Krumke, Sven Oliver ; Noltemeier, Hartmut: Graphentheoretische Konzepte Und Algorithmen. 2. akt. Aufl. 2009. Berlin: Springer DE, 2009.

[15] Needleman, Saul B. ; Wunsch, Christian D.: A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. J. Mol. Biol. (1970) 48,443-453

[16] Pomberger, Gustav ; Dobler, Heinz: Algorithmen und Datenstrukturen. 1. Aufl.. München: Pearson Deutschland GmbH, 2008

[17] Rosenberg, Michael S.: Sequence Alignment : Methods, Models, Concepts, and Strategies. London: University of California Press, 2009.

[18] Smith, T.F. ; Waterman, M.S. : Identification of Common Molecular Subsequences. J. Mol. Biol (1981) 147,195-197

[19] Soulié, Juan: C++ Language Tutorial, cplusplus.com

[20] Steger, Gerhard: Bioinformatik : Methoden Zur Vorhersage Von Rna- Und Proteinstrukturen. 2003. Aufl.. Berlin: Springer DE, 2003.

## 10.1 Inhalt der CD

Auf der beigefügten CD sind folgende Inhalte:

| | |
|---|---|
| Bachelor.pdf | Die Arbeit als pdf-Datei |
| Dijkstra.cpp | Der Dijkstra-Algorithmus |
| Astar.cpp | Der A*-Algorithmus |
| PSADijkstra.cpp | Dijkstra für Pairwise Sequence Alignment |
| PSAAstar.cpp | A* für Pairwise Sequence Alignment |
| Dmin.cpp | Dijkstra für PSA mit effizienter Minimum-Suche |
| Amin.cpp | A* für PSA mit effizienter Minimum-Suche |
| Topsort.cpp | Pairwise Sequence Alignment mit topologischer Sort. |

Die C++-Programme sind jeweils mit Beispielinstanzen ausgestattet, die beliebig verändert werden können. Die Ausgabe der Alignments ist leider nicht ganz zuverlässig, die kürzesten Wege stimmen aber.

## 10.2 Selbstständigkeitserklärung

Hiermit versichere ich, Philipp Seemann (Matrikelnummer: 20979060), dass ich diese Bachelorarbeit mit dem Thema „Kürzeste-Wege-Algorithmen für Sequence Alignment“ selbsständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

---

Göttingen, 1. Februar 2013