Einführung in die Bioinformatik
Algorithmen zur Sequenzanalyse

!! Vorläufige Fassung, nur einzelne Abschnitte !!
!! Enthält wahrscheinlich noch viele Fehler !!
!! Wird regelmäßig erweitert und verbessert !!

Burkhard Morgenstern

11. Juli 2005

Inhaltsverzeichnis

Kapitel 1

Definitionen

Ein *Alphabet* $\mathcal{A}$ ist eine endliche Menge von *Symbolen* oder *Zeiche*. Bei der Analyse von Nukleinsäure- oder Proteinsequenzen ist dabei $\mathcal{A}$ meist die Menge der Nukleotide bzw. der Aminosäuren, d.h. man hat

$$\mathcal{A} = \{A, T, G, C\}$$

bzw.

$$\mathcal{A} = \{A, C, D, E, F, \ldots, W, Y\}.$$

Eine *Sequenz* X *über einem Alphabet* $\mathcal{A}$ ist eine Endliche Folge von Symbolen

$$X = X_1 X_2 \ldots X_n$$

mit $X_i \in \mathcal{A}$ für alle Indizes i. n ist dabei die *Länge* der Sequenz X. Ein *Präfix* einer Sequenz X ist der Teilabschnitt

$$X_1 X_2 \ldots X_i$$

von X von X_1 bis zu einem beliebigen Zeichen X_i. Hierzu gehören auch die Spezialfälle $i = 0$ und $i = n$. Die *leere Sequenz*, die aus überhaupt keinem Symbol besteht ($i = 0$) und die ursprüngliche Sequenz X ($i = n$) sind also auch Präfixe von X. Entsprechend ist ein *Suffix* einer Sequenz X definiert als ein Teilabschnitt

$$X_j X_{j+1} \ldots X_n$$

von X, der bei einem beliebigen Zeichen X_j beginnt und bis zum Ende von X geht. Die leere Sequenz und die ursprüngliche Sequenz X werden auch als Suffixe von X angesehen.

Kapitel 2

Paarweises Sequenzalignment

2.1 Was ist ein Alignment?

Praktisch alle Bioinformatik-Methoden zur Analyse von DNA-, RNA- oder Proteinsequenzen basieren direkt oder indirekt auf dem *Vergleich* von Sequenzen. Dabei werden zwei oder mehrere Sequenzen daraufhin untersucht, *ob* bzw. *wo* sie Ähnlichkeiten zueinander aufweisen, und wie *stark* diese Ähnlichkeiten sind. Vergleichende Sequenzanalyse ist ein zentrales Hilfsmittel zur Analyse von Verwandtschaftsverhältnissen und zur Konstruktion von Stammbäumen von Arten, aber auch zur Analyse von 3D-Struktur und Funktion von Proteinen oder zur Genomanalyse. Proteine mit ähnlicher *Primärsequenz*, d.h. ähnlicher Abfolge von Aminosäuren sind in der Regel *evolutionär* miteinander verwandt, d.h. sie gehen auf einen gemeinsamen Ursprung in der Stammesgeschichte zurück. Da sich Protein- und RNA-Strukturen im Lauf der Evolution langsamer ändern als die "dahinter" liegenden Primärsequenzen, kann man von Ähnlichkeiten in der Primärsequenz im Allgemeinen auf Ähnlichkeiten in der Sekundär- oder Tertiärstruktur schließen. Da die Funktion eines Proteins von seiner Struktur abhängt, kann man von Ähnlichkeiten auf der Sequenzebene oft auf die Funktion von Proteinen schließen.

Innerhalb einer Protein- oder einer Nukleinsäuresequenz sind in der Regel diejenigen Bereiche am stärksten konserviert, die für Struktur und Funktion besonders wichtig sind. In strukturell und funktionell wichtigen Bereichen von Makromolekülen können Mutationen nur begrenzt toleriert werden, da zu große Änderungen auf der Sequenzebene Struktur und Funktion des Moleküls beeinträchtigt und die Überlebenschancen des Organismums verkleinert würden. *Lokal* konservierte Bereichen in Sequenzen sind daher im Allgemeinen funktionell wichtig. In Genomen von Eukaryoten sind zum Beispiel Protein-kodierende Bereiche in der Regel stärker konserviert als nicht-kodierende Bereiche. Diese Tatsache kann zur Identifizierung von Genen in Genomsequenzen ausgenützt

werden.

Wenn Sequenzen miteinander verglichen werden sollen, ist zunächst nicht offensichtlich, *welche* Bereiche einer Sequenz überhaupt mit welchen Bereichen anderer Sequenzen zu vergleichen sind. Die erste Aufgabe beim Sequenzvergleich ist daher eine *Zuordnung* von Sequenzbereichen bzw. einzelnen Positionen der Sequenzen. Ein ähnliches Problem hat man zum Beispiel beim Vergleich von Computer-Dateien. Wenn man zwei Versionen einer Textdatei miteinander vergleichen will, kann man im Allgemeinen nicht die i-te Zeile der ersten Datei mit der i-ten Zeile der zweiten Datei vergleichen, denn bei der Bearbeitung der Datei könnten ja Zeilen gelöscht oder hinzugefügt worden sein. Bevor man sagen kann, welche Unterschiede zwischen den beiden Dateien bestehen, muss man also zunächst bestimmen, welche Zeilen einander entsprechen.

Ein *Alignment* ist eine Zuordnung von zwei oder mehreren Sequenzen, die man dadurch erhält, dass man die Sequenzen *übereinander* schreibt. Das Ziel ist dabei, die Sequenzen so übereinander anzuordnen, dass nach Möglichkeit diejenigen Positionen der Sequenzen, übereinander stehen, die "biologisch miteinander verwandt sind". Da man nicht erwarten kann, dass miteinander verwandte Positionen in den zu vergleichenden Sequenzen unbedingt an den selben Positionen stehen, müssen im Allgemeinen Teile der Sequenzen gegeneinander verschoben werden, bis die "passenden" Bereiche der Sequenzen übereinander stehen. Hierfür werden *Lücken* (englisch: "Gaps") in die Sequenzen eingefügt, siehe Abbildung 2.1. Die Gaps in einem Alignment werden mit dem Symbol "–" bezeichnet. Ein Alignment von zwei Sequenzen heißt *paarweises* Alignment, eines von mehreren Sequenzen *multiples Alignment*.

Um den Begriff des paarweisen Alignments formal zu definieren, betrachten wir ein *Alphabet* $\mathcal{A}$ und zwei *Eingabesequenzen*

$$X = X_1 X_2 \dots X_m$$

und

$$Y = Y_1 Y_2 \dots Y_n,$$

über diesem Alphabet, wobei m und n die Längen von X und Y sind. Wir haben also $X_i \in \mathcal{A}$ und $Y_j \in \mathcal{A}$ für alle Indizes i und j. Beim Alignment von Nukleinsäuren ist das Alphabet $\mathcal{A}$ dabei die Menge der vier Nukleotide, beim Proteinalignment ist $\mathcal{A}$ die Menge der 20 Aminosäuren. Ein paarweises Alignment A kann man dann als ein Paar von so genannten *verallgemeinerten Sequenzen*

$$X^* = X_1^* X_2^* \dots X_K^*$$

und

$$Y^* = Y_1^* Y_2^* \dots Y_K^*,$$

über einem *erweiterten* Alphabet

$$\mathcal{A}^* = \mathcal{A} \cup \text{"–"}$$

definieren, das folgende zwei Bedingung erfüllt:

```
ATCGGAGTACT ,  ACCGGTTAGT


ATCGGAGT-ACT
ACCG--GTTAGT
```

Abbildung 2.1: Paarweises Alignment (unten) zweier fiktiver DNA-Sequenzen (oben). Lücken (*Gaps*) sind in beide Sequenzen eingefügt worden, so dass "zusammengehörige" Bereiche der Sequenzen übereinander stehen. Bei biologischen Sequenzen will man in einem Alignment nach Möglichkeit diejenigen Bereiche der Sequenzen übereinander schreiben, die (vermutlich) biologisch miteinander verwandt sind.

1. Wenn man aus X^* und X^* die Gap-Zeichen "$-$" entfernt, erhält man die Eingabesequenzen X und Y
2. Für jeden Index i ist entweder $X_i^* \in \mathcal{A}$ oder $Y_i^* \in \mathcal{A}$

Mit anderen Worten: Man fügt in die Sequenzen X und Y Gap-Zeichen so ein, dass die resultierenden "verallgemeinerten Sequenzen" X^* und Y^* gleich lang sind und dass keine zwei Gap-Zeichen übereinanderstehen, wenn man X^* und Y^* übereinander schreibt. Dabei nennt man

$$A_i = \begin{pmatrix} X_i^* \\ Y_i^* \end{pmatrix}$$

die i-te *Spalte* des Alignments A. Eine Spalte nennt man ein *Match*, wenn zwei identische Zeichen aus $\mathcal{A}$ übereinanderstehen und ein *Mismatch*, wenn zwei verschiedene Zeichen aus $\mathcal{A}$ übereinanderstehen. Ein Alignment besteht daher aus Matchen, Mismatchen und Gaps.

In einem Alignment will man solche Bereiche der Sequenzen einander zuordnen, die "biologisch" miteinander verwandt sind. Auf die Fragen, was "biologisch verwandt" bedeuten soll, gibt es prinzipiell zwei verschiedene Antworten. Wenn man Nukleinsäure- oder Proteinsequenzen vom Standpunkt der *Evolution* betrachtet, bedeutet "verwandt" so viel wie "auf einen gemeinsamen Ursprung in der Evolution zurückgehend". Diese Art von Verwandtschaft nennt man auch *Homologie* – im Gegensatz zur *Paralogie*, wo Ähnlichkeiten in Organismen oder Sequenzen *unabhängig* voneinander entstanden sind. Vom Standpunkt der Evolution sind zwei Positionen in einer Sequenz also dann miteinander verwandt, wenn sie auf eine gemeinsame Position in einer gemeinsamen Vorläufersequenz zurückgehen.

Beim Vergleich von Protein- und RNA-Sequenzen, gibt es noch eine zweite Sichtweise: Wenn man sich für die *3D-Struktur* von Makromolekülen interessiert, wird man Positionen in Sequenzen dann als "verwandt" ansehen, wenn sie die selbe Stelle im dreidimensionalen Raum einnehmen. Die beiden Standpunkte sind nicht so unterschiedlich, wie man zunächst annehmen könnte. Da

wie oben bemerkt, die 3D-Strukturen von Proteinen im Verlauf der Evolution stärker konserviert bleiben als die dahinter liegende Primärstruktur, sind Aminosäuren, die an vergleichbaren Stellen im 3D-Raum stehen, im Allgemeinen auch evolutionär verwandt – und umgekehrt.

Aus der oben gegebenen Definition des Alignments folgt auch schon eine prinzipielle Einschränkung ihrer Anwendbarkeit. Ein Alignment kann biologisch verwandte Bereiche der Sequenzen nur dann richtig einander zuordnen, wenn diese Bereiche in den verglichenen Sequenzen *in der selben relativen Reihenfolge* auftreten. Ob dies der Fall ist, hängt von der Art der Mutationen ab, die im Lauf der Evolution eingetreten sind, seit sich die untersuchten Sequenzen aus einer gemeinsamen Vorläufersequenz entstanden sind. Sind lediglich *Insertionen, Deletionen* und *Substitutionen* in den Sequenzen vorgekommen, dann ist die relative Reihenfolge innerhalb der Sequenzen nicht verändert, und die sich entsprechenden Bereiche der Sequenzen können in einem Alignment einander zugeordnet werden. Sind allerdings *Inversionen*, *Translokationen* oder *Duplikationen* vorgekommen, kann man die Verwandschaftsverhältnise in einer Sequenzfamilie nicht mehr oder nur noch eingeschränkt durch ein Alignment darstellen. Immerhin ist es auch in diesem Fall noch möglich, einzelne Regionen der Sequenzen miteinander zu alignen. Statt eines einzelnen Alignments, das die Verwandschaftsverhältnisse in einer Sequenzfamilie beschreibt, hat man in diesem Fall mehrere Alignments, die jeweils verwandte Teilbereiche der Sequenzen abdecken.

Alignments sind die Grundlage der vergleichenden Analyse von biologischen Sequenzen. Die Entwicklung von Algorithmen und Software für das paarweise und multiple Alignment ist daher eine zentrale Aufgabe der Bioinformatik. Die elementare Aufgabe ist dabei das *paarweise* Sequenzalignment, d.h. das Alignment von zwei Sequenzen. Ein multiples Alignment – d.h. ein Alignment von drei oder mehr Sequenzen – enthält zwar wesentlich mehr Information als ein paarweises alignment, alle Methoden für das Multiple Alignment basieren aber letztlich auf Methoden für das paarweise Alignment. Die in diesem Kapitel vorgestellten Methoden für das paarweise Alignment bilden daher auch die Grundlage für die im nächsten Kapitel diskutierten Ansätze für den multiplen Sequenzvergleich.

2.2 Zielfunktionen für das paarweise Alignment

Das Ziel beim Sequenzalignment ist die Berechnung von Alignments, die *biologisch* Sinn machen, die also vom Standpunkt der Evolution oder der 3D-Struktur richtig sind. Ein Computer ist eine Rechenmaschine. Er versteht weder etwas von Evolution noch etwas von Protein- oder RNA-Strukturen. Wenn ein Rechner sinnvolle Alignments produzieren soll, braucht er daher als erstes ein einfaches mathematisch definiertes *Kriterium*, mit dem er beurteilen kann, wie gut oder schlech ein gegebenes Alignment ist. Man muss daher zunächst eine *Zielfunktion* definieren, mit der jedem *möglichen* Alignment A einer gegebenen Sequenzfamilie ein *Qualitätswert* oder *Score* $S(A)$ zugeordnet werden kann. Nur wenn solch eine Zielfunktion gegeben ist, kann eine Rechenmschine daran gehen, ein

möglichst "gutes" Alignment zu finden, das dann (hoffentlich) auch noch *biologisch* Sinn macht.

Die erste Aufgabe bei der Entwicklung von Algorithmen für das Sequenzalignment ist die Definition eines *Bewertungsschemas*, das jedem möglichen Alignment einen numerischen Wert zuordnet, und zwar so, dass so weit wie möglich *biologisch* sinnvolle Alignments auch *mathematisch* hohe Werte bekommen - und umgekehrt. Im Idealfall hätte man gerne eine Zielfunktion, die für jeden beliebigen Satz von Eingabesequenzen dem *biologisch* besten Alignment den *mathematisch* höchsten Score zuordnen würde. So eine Zielfunktion existiert natürlich nicht, denn biologische Verwandtschaft läßt sich nicht durch ein einfaches mathematisches Verfahren berechnen. Die Aufgabe ist, ein Bewertungsschema für Alignments zu finden, das in *möglichst vielen Situationen* den biologisch sinnvollen Alignments die besten – oder jedenfalls hohe – mathematische Scores zuordnet.

Ist ein solches Bewertungsschema – d.h. eine sinnvolle Zielfunktion – gegeben, kann man *Optimierungsverfahren* entwickeln, die möglichst effizient ein *optimales* Alignment berechnen. Dieses mathematische Problem wird in diesem und im folgenden Kapitel behandelt. Es ist jedoch klar, dass jedes *automatische* Verfahren zur Berechnung von Alignments nur dann sinnvolle Ergebnisse liefern kann, wenn die zu Grunde liegende Zielfunktion sinnvoll ist, d.h. wenn biologisch sinnvolle Alignments auch numerisch hohe Scores bekommen. Die Definition einer geeigneten Zielfunktion ist daher die wichtigste Frage bei der Entwicklung von Alignmentmethoden.

Die meisten Zielfunktionen für das Sequenzalignment basieren darauf, dass man erstens jedes Paar von miteinander alinierten Basen bzw. Aminosäuren bewertet. Zweitens bewertet man jedes *Gap* im Alignment. Aus diesen Werten berechnet man dann den *Score* $S(A)$ des Alignments A. Für jeweils zwei Zeichen a und b aus dem Alphabet $\mathcal{A}$ benötigt man daher zunächst einen Ähnlichkeitswert

$$s(a, b).$$

Beim Alignment von DNA- und RNA-Sequenzen fragt man dabei nur, ob a und b gleich oder verschieden sind. Man setzt dann

$$s(a,b) = \begin{cases} C_1 & \text{falls } a = b \\ C_2 & \text{falls } a \neq b, \end{cases} \tag{2.1}$$

wobei C_1 und C_2 Konstanten sind mit $C_1 > 0$ und $C_2 < 0$.

Beim Proteinalignment ist die Sache etwas komplizierter. Die 20 Aminosäuren haben chemische und physikalische Eigenschaften, die ihre Funktion bestimmen, und diese Eigenschaften weisen unterschiedliche *Grade* von Ähnlichkeit untereinander auf. Daher wäre es zu einfach, nur zwischen *identisch* und *nicht identisch* zu unterscheiden. Beim Proteinalignment verwendet man ein Bewertungsschema, dass Paaren von ähnlichen Aminosäuren hohe (positive) Werte zuordnet, während unähnlichen Paaren von Aminosäuren niedrige bzw. negative Werte zugeordnet werden. Die Ähnlichkeitswerte für alle möglichen Paare (a, b) von Zeichen kann man in einem rechteckigen Schema anordenen. Man spricht

daher von einer *Ähnlichkeitsmatrix* bzw. *Substitutionsmatrix*. Wie solche Substitutionsmatrizen konstruiert werden können, wird in Kapitel **??** ausführlich besprochen. In diesem Kapitel gehen wir davon aus, dass wir eine Funktion s haben, die die Ähnlichkeit von jeweils zwei Basen bzw. Aminosäuren bewertet. Wie diese Funktion aussieht, und wie sie definiert worden ist, ist für die Beschreibung von Alignment-Algorithmen nicht wichtig.

Schließlich wird einem *Gap* im Alignment ein Wert zugeordnet, der von seiner *Länge* l abhängt. Man hat also eine Funktion

$$\gamma : N \rightarrow R,$$

so dass jedes Gap der Länge l mit $\gamma(l)$ bewertet wird. Die Werte von γ sind grundsätzlich *negativ*; man nennt den einem Gap zugeordneten Wert die *Gap-Penalty*. Bei Standard-Methoden für das globale Sequenzalignment wird der Scores eines Alignments dann definiert als die *Summe der Ähnlichkeitswerte der alinierten Paare plus die Summe der Gap-Penalties.*

Der Vollständigkeit wegen soll erwähnt werden, dass es auch andere Zielfunktionen für das paarweise Sequenzalignment gibt. Eine Variante der oben erklärten Zielfunktion für das *lokale* Alignenment wird in Abschnitt **??** definiert. Eine grundsätzlich andere Zielfunktion für das paarweise und multiple Alignment wird in Abschnitt **??** eingeführt.

2.3 Der Needleman-Wunsch-Algorithmus

Auf der Grundlage der im letzten Abschnitt definierten *Zielfunktion* für das paarweise Alignment, ist jetzt die nächste Aufgabe, ein *optimales* Alignment A für zwei gegebene Sequenzen zu berechnen, d.h. ein Alignment mit maximalem Score $S(A)$. Die einfachste Methode, ein optimales Alignment zu berechnen, besteht darin, alle *möglichen* Alignments der Eingabesequenzen zu betrachten, ihre Scores zu berechnen und schließlich dasjenige mit dem höchsten Score auszugeben. Die Zahl der möglichen Alignments von zwei Sequenzen ist jedoch so groß, dass dieser *naive* Algorithmus für realistische Datensätze nicht in Frage kommt.

2.3.1 Voraussetzungen

Der grundlegende Optimierungsalgorithmus für das Alignment von zwei Sequenzen ist 1970 von zwei Biologen, S. Needleman und C. Wunsch, vorgeschlagen worden [2]. Der Algorithmus verwendet das Verfahren des *Dynamischen Programmierens*, das im Anhang erklärt ist. Eingabedaten des Algorithmus sind zwei Sequenzen

$$X = X_1 X_2 \ldots X_m$$

und

$$Y = Y_1 Y_2 \ldots Y_n,$$

Ausserdem sei die Ähnlichkeitsfunktion s und die *Gap-Penalty* γ vorgegeben. In diesem Kapitel gehen wir zunächst davon aus, dass γ eine *lineare* Funktion ist, dass es also eine Konstante g gibt mit

$$\gamma(l) = g \cdot l.$$

Ein Gap der Länge l zählt dann genau so viel wie l Gaps der Länge 1, nämlich

$$\gamma(l) = l \cdot \gamma(1) = l \cdot g$$

Etwas allgemeinere Gap-Penalties werden später behandelt. Die Voraussetzung der linearen Gap-Penalties ist entscheidend für die Version des Algorithmus, die in diesem Abschnitt erklärt wird. Bei linearen Gap-Penalties kann man bei der Berechnung des Scores eines Alignments so tun, als würde jedes Gap aus einzelnen Gaps der Länge 1 bestehen, d.h. statt ein Gap der Länge l als Ganzes mit $\gamma(l) = l \cdot g$ zu bewerten, kann man auch jedes der l Gap-Zeichen mit g bewerten. Damit kann man den Score $S(A)$ eines Alignments A *spaltenweise* berechnen, d.h. es gilt

$$S(A) = \sum_{i=1}^{K} S(A_i), \tag{2.2}$$

wobei $S(A_i)$ der Score der i-ten Spalte des Alignments ist, der definiert ist als

$$S(A_i) = \begin{cases} s(X_i^*, Y_i^*) & \text{falls } X_i^* \in \mathcal{A} \text{ und } Y_i^* \in \mathcal{A} \text{ ist, d.h. falls} \\ & \text{in } A_i \text{ zwei Zeichen aliniert sind.} \\ g & \text{sonst, d.h. falls in } A_i \text{ ein Gap steht} \end{cases} \tag{2.3}$$

Wenn man die Definition der Ähnlichkeitsfunktion durch

$$s(-, a) = s(a, -) = g$$

für alle $a \in \mathcal{A}$ erweitert, kann man Gleichung (2.3) auch schreiben als

$$S(A_i) = s(X_i^*, Y_i^*) \tag{2.4}$$

Alle Überlegungen in diesem Abschnitt basieren auf der Voraussetzung, dass wir lineare Gap-Penalties verwenden.

2.3.2 Optimales Alignment durch *Dynamisches Programmieren*

Wie beim Dynamischen Programmieren üblich, wird das Ausgangs-Problem – ein optimales Alignment der Sequenzen X und Y zu finden – in kleinere *Teil*probleme zerlegt, die dann schrittweise behandelt werden. Für Zahlen $0 \leq i \leq m$ und $0 \leq j \leq n$ betrachten wir dabei das Teilproblem, ein optimales Alignment der *Präfixe* von X und Y bis i bzw. j zu finden. Wir betrachten also die Sequenzen

$$X_1 X_2 \ldots X_i$$

und

$$Y_1 Y_2 \dots Y_j.$$

Beachte, dass hierbei auch Präfixe der Länge 0 betrachtet werden, also Sequenzen, die aus keinem einzigen Zeichen bestehen (falls $i = 0$ bzw. $j = 0$ ist). Entsprechend dem Vorgehen beim Dynamischen Programmieren lösen wir nacheinander die folgenden vier Fragen:

1. Bestimme die Struktur einer (Teil-)Lösung, d.h. der Lösung eines (Teil-) Problems.
2. *Definiere* den Score einer Teil-Lösung rekursiv.
3. *Berechne* den Score für die Teillösungen schrittweise.
4. Berechne die *Lösung* des ursprünglichen Problems

Die Anwendung dieses Prinzips auf das Problem des optimalen paarweisen Alignments besteht aus folgenden Schritten:

1. Struktur einer optimalen (Teil-)Lösung. Unter der oben gemachten Voraussetzung gilt: Wenn man ein Alignment

$$A = \begin{array}{cccccccc} A & T & T & G & G & T & C & A & - \\ A & T & - & - & C & T & A & A & T \end{array}$$

zweier Sequenzen X und Y nach einer beliebigen Spalte "durchschneidet", so dass man zwei Teil-Alignments

$$A^{(1)} = \begin{array}{cccccc} A & T & T & G & G & T \\ A & T & - & - & C & T \end{array}$$

und

$$A^{(2)} = \begin{array}{ccc} C & A & - \\ A & A & T \end{array}$$

erhält, dann gilt für die Scores dieser Alignment

$$S(A) = S\left(A^{(1)}\right) + S\left(A^{(2)}\right).$$

Dies folgt einfach daraus, dass bei linearen Gap-Penalties der Score eines Alignments die Summe der Spalten-Scores ist. Weiter folgt: Wenn A ein *optimales* Alignment der Eingabesequenzen X und Y ist, also ein Alignment mit *maximalem* Score, dann ist $A^{(1)}$ ein optimales Alignment der in $A^{(1)}$ alinierten *Präfixe* von X und Y, und $A^{(2)}$ ist ein optimales Alignment der entsprechenden *Suffixe*. Auch diese Tatsache ist leicht einzusehen: Gäbe es ein besseres Alignment $\tilde{A}^{(1)}$ der betreffenden Präfixe, dann könnte man in A einfach den "Anfang" $A^{(1)}$ durch $\tilde{A}^{(1)}$ ersetzen und bekäme ein Alignment von X und Y, das einen höheren Score als A hätte. Das geht aber nicht, da A nach Voraussetzung bereits optimal war. Daher kann es kein Alignment der Präfixe geben, das einen höheren Score als $A^{(1)}$ hat. Im nächsten Abschnitt wird diese Tatsache ausgenützt, um

den Score eines optimalen Alignments von Präfixen von X und Y zu berechnen, indem man die letzte Spalte des Alignments "abtrennt" und die Scores für die letzte Spalte und das "Rest-Alignment" getrennt berechnet.

2. Rekursive Gleichung für den Score von Teil-Alignments: Wir betrachten jetzt ein optimales Alignment der Präfixe

$$X_1 X_2 \dots X_i$$

und

$$Y_1 Y_2 \dots Y_j,$$

den Score dieses Alignments nennen wir $S(i,j)$. Wenn man die letzte Spalte eines Alignments $A^{(i,j)}$ der Präfixe betrachtet, gibt es drei Möglichkeiten:

(a) Die beiden letzten Zeichen der Präfixe – also X_i und Y_i – sind miteinander aliniert:

$$A^{(i,j)} = \begin{pmatrix} \cdots & X_i \\ \cdots & Y_j \end{pmatrix}$$

(b) X_i steht über einem Gap-Zeichen:

$$A^{(i,j)} = \begin{pmatrix} \cdots & X_i \\ \cdots & - \end{pmatrix}$$

oder

(c) Y_j steht unter einem Gap-Zeichen:

$$A^{(i,j)} = \begin{pmatrix} \cdots & - \\ \cdots & Y_j \end{pmatrix}$$

Um den Score $S(i,j)$ des optimalen Alignments der Präfixe bis i und j zu berechnen berechnen wir für *jede* der drei Möglichkeiten (a) bis (c) den Score des besten Alignments unter der jeweiligen Bedingung. Wir berechnen also den Score des besten Alignments der Präfixe (a) unter der Bedingung, dass X_i und Y_i alignt sind, (b) unter der Bedingung, dass X_i über einem Gap steht und (c) unter der Bedingung dass Y_j unter einem Gap steht. Diese Werte nennen wir $S^{(a)}(i,j), S^{(b)}(i,j)$ bzw. $S^{(c)}(i,j)$. Der Score $S(i,j)$ des *insgesamt* besten Alignments der Präfixe muss dann das *Maximum* dieser drei Werte sein:

$$S(i,j) = \max\left\{S^{(a)}(i,j), S^{(b)}(i,j), S^{(c)}(i,j)\right\} \tag{2.5}$$

Um die Scores $S^{(a)}(i,j), S^{(b)}(i,j)$ und $S^{(c)}(i,j)$ zu berechnen, teilen wir die entsprechenden optimalen Alignments in Teil-Alignments auf, indem wir die letzte Spalte "abtrennen". Entsprechend den Überlegungen in 1. ist der Score dann gegeben durch den Spaltenscore der letzten Spalte plus den Score des *optimalen* Alignments des Rests der Sequenzen. Damit bekommt man:

(a) Falls X_i und Y_j alignt sind, gilt

$$S^{(a)}(i,j) = S(i-1,j-1) + s(X_i, Y_j), \tag{2.6}$$

denn in diesem Fall ist $s(X_i, Y_j)$ der Spalten-Score der letzten Spalte, und der Rest des Alignments muss ein optimales Alignment der Präfixe bis $i-1$ bzw. $j-1$ sein.

(b) Wenn X_i über einem Gap steht, ist In diesem Fall gilt

$$S^{(b)}(i,j) = S(i-1,j) + g, \tag{2.7}$$

denn dann ist g der Spalten-Score der letzten Spalte, und der Rest des Alignments muss ein optimales Alignment der Präfixe bis $i-1$ bzw. j sein. Beachte, dass Sequenz X hier bis zum Zeiche X_{i-1} läuft, denn das Zeichen X_i steht ja in der letzten Spalte. Sequenz Y läuft dagegen bis Zeichen Y_j, denn in der letzten Spalte steht ein Gap in Sequenz Y, das Zeichen Y_j muss also im "Rest-Alignment" vorkommen.

1. Analog sieht man, dass

$$S^{(c)}(i,j) = S(i,j-1) + g, \tag{2.8}$$

gilt.

Insgesamt erhalten wir aus den Gleichungen (2.5) bis (2.8) eine Rekursionsformel für den Score $S(i,j)$ des optimalen Alignments der Präfixe bis i und j:

$$S(i,j) = \max \begin{cases} S(i-1,j-1) & + & s(X_i, Y_j) \\ S(i-1,j) & + & g \\ S(i,j-1) & + & g \end{cases} \tag{2.9}$$

3. Effiziente Berechnung der Scores der Teil-Alignments Mit Gleichung (2.9) kann der Score $S(i,j)$ des optimalen Alignments der Präfixe bis i und j berechnet werden, wenn $i > 0$ und $j > 0$ ist und falls die entsprechenden Scores $S(i-1,j-1), S(i-1,j)$ und $S(i,j-1)$ bekannt sind. Für $i = 0$ kann $S(i,j) = S(0,j)$ direkt bestimmt werden. In diesem Fall hat das Präfix von X die Länge 0, und es gibt nur ein einziges mögliches Alignment der beiden Präfixe (das dann natürlich das optimale Alignment ist); dieses Alignment besteht aus einem einzigen Gap der Länge j:

$$\begin{pmatrix} - & - & \cdots & - \\ Y_1 & Y_2 & \cdots & Y_j \end{pmatrix}$$

sein Score ist

$$S(0,j) = \gamma(j) = j \cdot g, \tag{2.10}$$

entsprechend gilt

$$S(i,0) = \gamma(i) = i \cdot g. \tag{2.11}$$

```
Algorithmus RE_ALI_SCORE(X, Y, i, j) {
    if (i == 0 or j == 0) {
        Wende (2.10) bzw. (2.11) an, um S(i, j) zu berechnen ;
    }
    else {        /* d.h. i > 0 und j > 0 */
        S(i − 1, j − 1) = REC_ALI_SCORE(X, Y, i − 1, j − 1);
        S(i, j − 1) = REC_ALI_SCORE(X, Y, i − 1, j);
        S(i − 1, j) = REC_ALI_SCORE(X, Y, i, j − 1);
        Wende Gleichung (2.9) an, um S(i, j) zu berechnen ;
    }
    RETURN S(i, j) ;
}
```

Abbildung 2.2: Rekursive Berechnung des Scores eines Optimalen Alignments von zwei Sequenzen X und Y durch direkte Anwendung von Gleichung (2.9). Der Algorithmus ist extrem ineffizient, weil der selbe Wert $S(i, j)$ wiederholt berechnet wird.

Die Längen der Eingabesequenzen X und Y sind m und n, der Score des optimalen Alignments A von X und Y ist also gegeben als

$$S(A) = S(m, n).$$

Diesen Score könnte man durch *rekursive* Anwendung von Gleichung (2.9) berechnen; der entsprechende rekursive Algorithmus ist in Abbildung 2.2 skizziert. Dieser Algorithmus wäre alledings extrem ineffizient, da er sie selben Werte $S(i, j)$ mehrfach berechnen würde. Um $S(A) = S(m, n)$ zu berechnen würde der Algorithmus zum Beispiel zunächst sich selbst aufrufen, um die Werte $S(m-1, n-1)$, $S(m, n-1)$ und $S(m-1, n)$ berechnen, die in Gleichung benötigt werden. Um den Wert $S(m, n-1)$ zu berechen, wird dann nochmals $S(m-1, n-1)$ berechnet, u.s.w.

Das Prinzip des *Dynamischen Programmierens* beruht darauf, diese Zwischenlösungen jeweils nur *einmal* zu berechnen und für spätere Anwendungen zu speichern. Der Algorithmus von Needleman und Wunsch geht daher folgendermassen vor: Zunächst werden die Werte $S(i, 0)$ und $S(0, j)$ für alle $0 \leq i \leq m$ und $0 \leq j \leq n$ nach Gleichungen (2.10) und (2.11) berechnet. Dann werden für wachsende i und j die Werte $S(i, j)$ berechnet, bis schließlich $S(m, n)$ berechnet ist. Um den Wert $S(i, j)$ zu berechnen, müssen die drei möglichen "Vorgängerwerte" $S(i-1, j-1)$, $S(i-1, j)$ und $S(i, j-1)$ bekannt sein, siehe Figur 2.3. Man kann daher alle Werte $S(i, j)$ berechnen, indem man von links oben nach rechts unten in die in Abbildung 2.3 gezeigte *Vergleichsmatrix* geht und an jeder Stelle den Wert $S(0, j)$ aus den entsprechenden Vorgängerwerten berechnet. Der zuletzt berechnete Wert $S(m, n)$ ist dann der Score eines optimalen Alignments von X und Y.

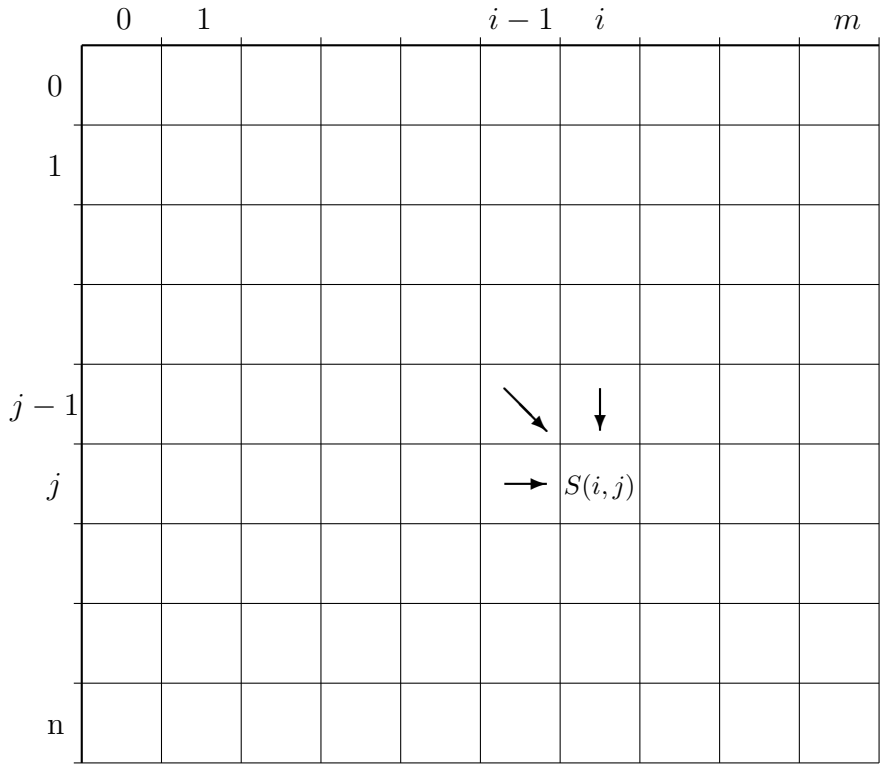


Abbildung 2.3: Rekursive Berechnung des Scores $S(i,j)$ eines optimalen Alignments der *Präfixe* der Sequenzen X und Y bis zu den Positionen i bzw. j. Der Wert $S(i,j)$ kann berechnet werden, wenn drei mögliche "Vorgängerwerte" $S(i-1,j-1)$, $S(i-1,j)$ und $S(i,j-1)$ bekannt sind. Die Werte $S(i,j)$ können daher schrittweise berechnet werden, indem man die zu den Sequenzen X und Y gehörende *Vergleichsmatrix* von links oben nach rechts unten ausfüllt.

Algorithmus NEEDLEMAN_WUNSCH(X, Y) **{**

```
/* Initialisieren der Werte von S(i,j) für i = 0 und j = 0 */
for( i = 0 ; i ≤ m ; i++ ) {
    Wende (2.10) an, um S(i,0) zu berechnen ;
    P(i,0) = 2 ;
}
for( j = 0 ; i ≤ n ; i++ ) {
    Wende (2.11) an, um S(0,j) zu berechnen ;
    P(i,0) = 3 ;
}

/* Rekursive Berechnung von S(i,j) */
for( i = 1 ; i ≤ m ; i++ )
for( j = 1 ; i ≤ n ; i++ ) {
    Wende (2.9) an, um S(i,j) zu berechnen ;
    Setze P(i,j) auf 1, 2, oder 3 je nach dem,
    ob in (2.9) das Maximum in Zeile 1, 2 oder 3
    angenommen wird ;
}

/* Trace-Back */
(i,j) = (m,n) ;
while( (i,j) ≠ (0,0)) {
    if(P(i,j) == 1 ){
        X_i aligned with Y_j;
        (i,j) = (i-1,j-1);
    }
    if(P(i,j) == 2 ){
        X_i aliniert mit Gap ;
        (i,j) = (i-1,j);
    }
    if(P(i,j) == 3 ) {
        Y_i aliniert mit Gap ;
        (i,j) = (i,j-1);
    }
}
}
```

Abbildung 2.4: Der *Needleman-Wunsch-Algorithmus* für die effiziente Berechnung eines Optimalen Alignments von zwei Sequenzen X und Y der Länge m bzw. n durch *Dynamisches Programmieren*. Zunächst werden die *Scores* $S(i, j)$ von optimalen Alignmments von *Präfixen* von X und Y schrittweise berechnet, Im letzten Schritt wird das Alignment von X und Y mit *Trace-Back* berechnet. Die Variable $P(i, j)$ gibt an, wo Gleichung (2.9) ihr Maximum annimmt – m.a.W. ob das optimale Alignment der Präfixe bis i und j mit einem Alignment von X_i und Y_j, mit einem Gap in Sequenz X oder mit einem Gap in Sequenz Y endet.

4. Berechnung des Alignments von X und Y Im Allgemeinen will man natürlich nicht nur den *Score* eines optimalen Alignments der Eingabesequenzen berechnen, sondern das optimale Alignment selbst. Dies erfordert nur wenig zusätzliche Arbeit. Für jedes Paar (i, j) von Indizes muss man dafür die Information haben, *wo* Gleichung (2.9) ihr Maxium annimmt, d.h. welcher der drei möglichen Werte der größte ist. Man definiert dann für jedes Paar (i, j) eine Variable $Ptr(i, j)$, die Werte 1, 2 oder 3 annehmen kann, je nachdem wo das Maximum in (2.9) angenommen wird. In anderen Worten: $Ptr(i, j)$ sagt, ob das optimale Alignment der Präfixe bis X_i und Y_j mit einem Alignment von X_i und Y_j endet, mit einem Gap in Sequenz X oder mit einem Gap in Sequenz Y.

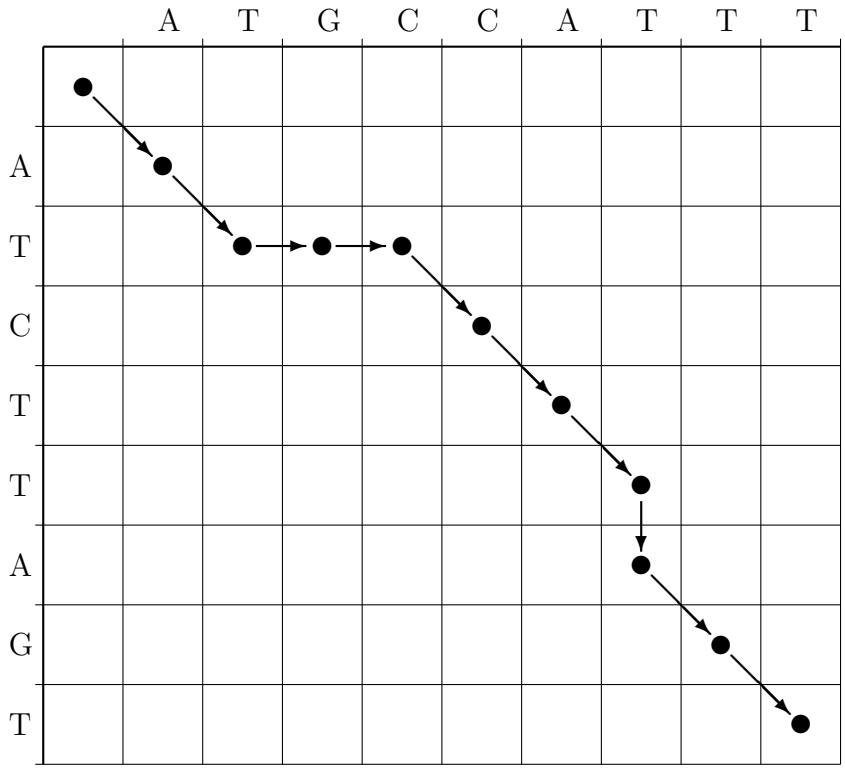
$$
\begin{aligned}
Ptr(i,j) = 1 \quad &\Rightarrow \quad \text{Optimales Alignment der Präfixe endet mit} \begin{pmatrix} X_i \\ Y_j \end{pmatrix} \\
Ptr(i,j) = 2 \quad &\Rightarrow \quad \text{Optimales Alignment der Präfixe endet mit} \begin{pmatrix} X_i \\ - \end{pmatrix} \\
Ptr(i,j) = 3 \quad &\Rightarrow \quad \text{Optimales Alignment der Präfixe endet mit} \begin{pmatrix} - \\ Y_j \end{pmatrix}
\end{aligned}
$$

2.3.3 Alignments als Pfade durch die *Verglelichsmatrix*

Wie in Abbildung 2.5 dargestellt, kann ein Alignment zweier Sequenzen X und Y durch einen *Pfad* durch die zu den Sequenezen gehörige *Vergleichsmatrix* dargestellt werden. Das Alignment in Abbildung 2.5 entspricht z.B. dem darüber skizzierten Pfad. Der zu einem Alignment gehörige Pfad läuft von der linken oberen Ecke in die rechte untere Ecke der Matrix. An jeder Stelle kann der Pfad um eine Position entweder senkrecht nach unten oder waagrecht nach rechts oder diagonal nach recht unten laufen. Jeder solche Pfad entspricht umgekehrt einem Alignment der Sequenzen X und Y. Der genaue Zusammenhang zwischen Alignments und Pfaden in der Vergleichsmatrix ist folgendermassen:

Jede Zelle in der Vergleichsmatrix, durch die der Pfad läuft, – mit Ausnahme der ersten Zelle $(0, 0)$ – entspricht dabei einer Spalte im Alignment. Wenn der Pfad von links oben in eine Zelle (i, j) läuft, bedeutet das, dass die Positionen X_i und Y_j im entsprechenden miteinander aliniert sind. Läuft der Pfad von links in die Zelle (i, j), dann steht X_i einem Gapzeichen gegenüber, läuft er von oben in die Zelle (i, j), dann steht Y_j einem Gapzeichen gegenüber.

Der zum Alignment A gehörige Score $S(A)$ kann direkt aus dem entsprechenden Pfad abgelesen werden: Jeder senkrechte oder waagerechte Schritt entspricht einem Gapzeichen. Sein Beitrag zum Score des Alignemnts ist daher die Penalty g für ein einzelnes Gapzeichen. Der Beitrag eines diagonalen Schritts in eine Zelle (i, j) zum Gesamtscore ist $s(X_i, X_j)$, da X_i und Y_j aliniert werden. Das Problem, ein optimales Alignment der Sequenzen X und Y zu finden ist also gleichbedeutend mit dem Problem, einen optimalen *Pfad* durch die dazugehörige Vergleichsmatrix zu finden. Den Score $S(i, j)$ kann man in diesem Sinne interpretieren als den Score des besten Pfades von der Zelle $(0, 0)$ bis zur Zelle (i, j). Beim *Trace-Back* wird der optimale Pfad rekonstruiert, indem man an jeder Zelle feststellt, woher der optimale Pfad, der zu dieser Zelle führt,



A T G C C A T - T T
A T - - C T T A G T

Abbildung 2.5: Ein Alignment zweier Sequenzen X und Y kann durch einen *Pfad* durch die zu den Sequenezen gehörige *Vergleichsmatrix* dargestellt werden. Der Pfad läuft von der linken oberen Ecke in die rechte untere Ecke. An jeder Stelle kann der Pfad um eine Position entweder senkrecht nach unten oder waagrecht nach rechts oder diagonal nach recht unten laufen. Jeder solche Pfad entspricht umgekehrt einem Alignment der Sequenzen X und Y.

kommt. Die Variable $Ptr(i, j)$ enthält die Information darüber, ob der optimale Pfad bis (i, j) von links, von oben oder von links oben in die Zelle (i, j) läuft. Im Folgenden werden daher Alignments und Pfade identifiziert, und ob wir von Alignments oder von Pfaden sprechen, hängt ganz davon ab, was in der jeweiligen Situation anschaulicher ist.

2.3.4 Die Komplexität des Needleman-Wunsch-Algorithmus

Um die Komplexität des oben besprochenen Algorithmus zum Alignment von zwei Sequenzen X und Y zu berechnen, betrachten wir die drei Hauptbestandteile des Algorithmus (vgl. Abbildung 2.4). Wir betrachten zunächst die *Zeit*-Komplexität.

- Das *Initialisieren* der Werte $S(i, 0)$ und $S(0, j)$ benötigt

  $$O(n + m)$$

  Zeit, denn für jeden der Werte $S(i, 0)$ und $S(0, j)$ ist eine konstante Zahl von Rechenoperationen notwendig.

- Die Berechnung der Werte $S(i, j)$ benötigt

  $$O(n \cdot m)$$

  Zeit, denn die Vergleichsmatrix hat $O(n \cdot m)$ Zellen (genauer: $(m+1) \cdot (n+1)$ Zellen), und an jeder Zelle muss wieder eine konstante Zahl von Operationen durchgeführt werden (nämlich der Vergleich der drei "Vorgängerwerte").

- Das Trace-Back benötigt wieder

  $$O(m + n)$$

  Zeit. Bei jeder Spalte im optimalen Alignment muss rekonstruiert werden, ob in ihr die betreffenden Positionen von X und Y aliniert sind, oder ob ein Gapzeichen in X bzw. in Y steht. Mit anderen Worten, man muss feststellen, woher der optimale Pfad bis (i, j) kommt. Für jede Zelle im Pfad braucht man hierfür eine konstante Zahl von Operationen. Die Rechenzeit für das Trace-Back ist also proportional zur Länge des optimalen Pfades, die höchstens $n + m$ sein kann.

Insgesamt läuft der Needleman-Wunsch-Algorithmus also in

$$O(n \cdot m) \tag{2.12}$$

Zeit. Um die Speicherplatzkomplexität des Algorithmus zu berechnen, muss man zunächst Speicherplatz für die beiden Eingabesequenzen veranschlagen. Hierfür ist

$$O(n + m)$$

Speicherplatz notwendig. Die Berechnung der Werte $S(i, j)$ benötigt

$$O(n)$$

Speicherplatz, denn der Algorithmus geht "spaltenweise" durch die Vergleichsmatrix. Um die Werte in Spalte i zu berechnen, müssen die entsprechenden Werte in Spalte $i-1$ bekannt sein. Wenn alle Werte in Spalte i berechnet sind, werden die Werte in Spalte $i-1$ nicht mehr benötigt und können gelöscht werden. Um den *Score* $S(m, n)$ eines optimalen Alignments von X und Y zu berechnen ist also insgesamt

$$O(n+m)$$

Speicherplatz notwendig. Für das "Trace-Back" werden allerdings zusätzlich die Werte $Ptr(i, j)$ benötigt. Da im ersten Teil des Algorithmus nicht bekannt ist, durch welchen Teil der Vergleichsmatrix der optimale Pfad (bzw. das optimale Alignment) verlaufen wird, müssen die Werte $Ptr(i, j)$ für *alle* Zellen (i, j) gespeichert werden. Wenn also nicht nur der Score des optimalen Alignments, sondern das optimale Alignment selbst ausgegeben werden soll, benötigt der Needleman-Wunsch-Algorithmus

$$O(m \cdot n)$$

Speicherplatz. In vielen Fällen sind die Längen der alinierten Sequenzen nicht allzu unterschiedlich. Dann ist es nicht notwendig, n und m bei der Komplexität getrennt zu betrachten, und man setzt

$$L = \max\{m, n\}.$$

Mit dieser Bezeichnung ist die Zeitkomplexität des Needleman-Wunsch-Algorithmus

$$O(L^2).$$

Die Speicherplatz-Komplexität ist

$$O(L)$$

für die Berechnung des *Scores* des optimalen Alignments und

$$O(L^2)$$

für die Berechnung des optimalen Alignments selbst. Man sagt daher, dass die (Zeit- und Speicherplatz-)Komplexität des Algorithmus *quadratisch* ist (bzw. die Speicherplatz-Komplexität linear, falls nur der optimale Score gesucht ist).

2.4 Varianten des Needleman-Wunsch-Algorithmus

2.4.1 Alignment mit allgemeinen Gap-Penalties

Der im letzten Kapitel erklärte Algorithmus für das optimale Alignment von zwei Sequenzen basiert darauf, dass die Scores von optimalen Alignments von

Präfixen von X und Y rekursiv berechnet werden. Die verwendete Rekursionsgleichung (2.9) hat ausgenutzt, dass man die letzte Spalte in den Präfix-Alignments "abspalten" konnte und den Score des Alignments als die Summe der resultierenden Teil-Alignments berechnen konnte. Die wesentliche Voraussetzung hierfür war, dass dass eine *lineare* Gap-Penalty verwendet wurde. Nur unter dieser Voraussetzung kann man ein Alignment A in Teil-Alignments $A^{(1)}$ und $A^{(2)}$ aufspalten und den Score von A als Summe der Scores der Teil-Alignments berechnen.

Vom Standpunkt der Biologie sind nicht-lineare Gap-Penalties sinnvoller als lineare. Insertionen und Deletionen betreffen oft nicht nur einzelne Basen bzw. Aminosäuren, sondern ganze Segmente der Sequenzen. Daher macht es biologisch wenig Sinn, ein Gap der Länge l genau so zu behandeln wie l Gaps der Länge 1. Wenn eine beliebige Gap-Funktion $\gamma(l)$ verwendet wird, ist der Score des Alignments im Allgemeinen nur für diejenigen Spalten additiv, in denen Positionen aus X und Y aliniert werden. Gaps müssen als ganzes behandelt werden. Wenn z.B. ein Alignment

$$A = \begin{array}{cccccccc} A & G & T & A & G & T & A & A & - \\ A & C & - & - & - & T & G & A & T \end{array}$$

innerhalb eines Gaps in Teil-Alignments

$$A^{(1)} = \begin{array}{cccc} A & G & T & A \\ A & C & - & - \end{array}$$

und

$$A^{(2)} = \begin{array}{ccccc} G & T & A & A & - \\ - & T & G & A & T \end{array}$$

zerlegt, dann ist im Allgemeinen

$$S(A) \neq S\left(A^{(1)}\right) + S\left(A^{(2)}\right).$$

Neben Spalten, in denen Positionen aliniert sind, kann ein Alignment A dagegen so gespalten werden, dass die Summe der Scores der Teil-Alignments gleich dem Score von A ist. Für A wie oben gilt z.B. für die Teil-Alignments

$$A^{(1)} = \begin{array}{cc} A & G \\ A & C \end{array}$$

und

$$A^{(2)} = \begin{array}{ccccccc} T & A & G & T & A & A & - \\ - & - & - & T & G & A & T \end{array}$$

$$S(A) = S\left(A^{(1)}\right) + S\left(A^{(2)}\right).$$

Um den Score $S(i,j)$ eines optimalen Alignments der Präfixe bis i und j zu berechnen, betrachten wir wieder die Situationen (a) bis (c) und die Scores $S^{(a)}(i,j)$, $S^{(b)}(i,j)$ und $S^{(a)}(i,j)$ wie auf Seite 15.

(a) Falls X_i und Y_j aligned sind, gilt wie bisher

$$S^{(a)}(i,j) = S(i-1,j-1) + s(X_i, Y_j), \tag{2.13}$$

denn in diesem Fall können wir die letzte Spalte des Alignments abspalten und die Scores für die letzte Spalte und das "Rest-Alignment" addieren.

(b) Wenn X_i über einem Gap steht, kann man nicht mehr einfach die letzte Spalte abtrennen, denn falls das Gap länger als 1 ist, würde man dann das Alignment innerhalb eines Gaps durchtrennen. Da wir nicht wissen, wie lange das Gap in Sequenz Y ist, müssen wir Gaps aller *möglichen* Längen l betrachten. Für festes l betrachten wir den Score des besten Alignments, das in Sequenz Y mit einem Gap der Länge l endet. Wir können das Alignment *vor* diesem Gap "auftrennen" und sein Score ist gegeben als Score des Gaps der Länge l plus dem Score des "Rest-Alignments", also als

$$S(i-l,j) + \gamma(l). \tag{2.14}$$

Um den Score $S^{(b)}(i,j)$ des besten Alignments zu berechnen, das mit einem Gap in Sequenz Y endet, müssen wir das *Maximum* von (2.14) über alle Gap-Längen l nehmen. Wir erhalten damit

$$S^{(b)}(i,j) = \max_{1 \le l \le i} S(i-l,j) + \gamma(l). \tag{2.15}$$

(c) Analog bekommt man

$$S^{(c)}(i,j) = \max_{1 \le l \le j} S(i,j-l) + \gamma(l). \tag{2.16}$$

Zusammenfassend erhält man den Score $S(i,j)$ als

$$S(i,j) = \max \begin{cases} S(i-1,j-1) & + & s(X_i, Y_j) \\ \max_{1 \le l \le i} S(i-l,j) & + & \gamma(l) \\ \max_{1 \le l \le j} S(i,j-l) & + & \gamma(l) \end{cases} \tag{2.17}$$

Ein optimales Alignment der Sequenzen X und Y kann man mit dem in Abbildung 2.4 beschrieben Needleman-Wunsch-Algorithmus berechnen, wenn man Rekursionsgleichung (2.9) durch Gleichung (2.17) ersetzt. Die Variable $Ptr(i,j)$ muss dann gegebenenfalls die Information enthalten, bei welcher Gap-Länge das Maximum in (2.17) angenommen wird.

Die Zeitkomplexität des Needleman-Wunsch-Algorithmus mit allgemeinen Gap-Penalties ist höher als mit linearen Gap-Kosten. Wie im vorherigen Abschnitt muss $S(i,j)$ für alle Zellen (i,j) der Vergleichsmatrix berechnet werden. Auswertung der Gleichung 2.17 ist jetzt aber nicht mehr in konstanter Zeit möglich, da alle Gap-Längen l berücksichtigt werden müssen. l kann Werte bis zu m bzw. n annehmen. Die Berechnung des Maximums in (2.17) benötigt daher für eine einzelne Zelle (i,j) der Vergleichsmatrix $O(\max\{m,n\})$ Schritte.

Da dieses Maximum für *jede* Zelle der Matrix berechnet werden muss, ist die Zeitkomplexität des Algorithmus

$$O(m \cdot n \cdot \max\{m, n\})$$

bzw. wenn man wieder $L = \max\{m, n\}$ setzt,

$$O(L^3). \tag{2.18}$$

Die Speicherplatzkomplexität ist wie bisher

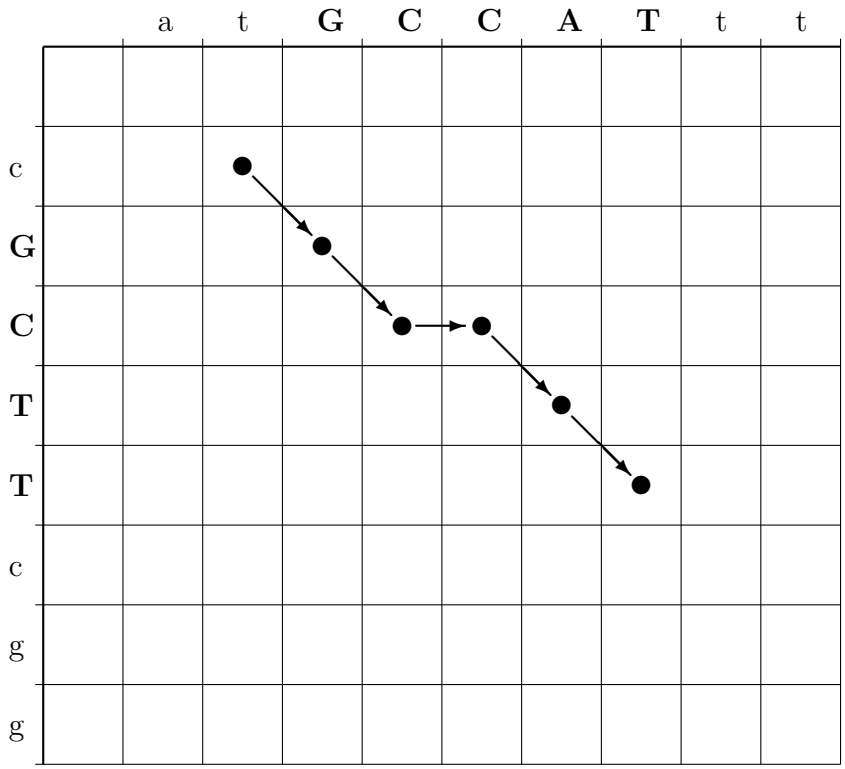
$$O(L^2), \tag{2.19}$$

nicht wesentlich mehr Werte gespeichert werden müssen.

2.4.2 Lokales Alignment (Smith und Waterman)

Bisher haben wir Algorithmen betrachtet, die ein *globales* Alignment zweier Eingabesequenzen X und Y berechnen, d.h. ein Alignment, dass die Eingabesequenzen über ihre ganze Länge aliniert. Oft hat man allerdings die Situation, dass Sequenzen nur *lokal* miteinander verwandt sind, z.B. wenn zwei Proteinsequenzen eine gemeinsame Domäne aufweisen, aber außerhalb dieser Domäne keine Ähnlichkeit zueinander aufweisen. Man benötigt daher Alignment-Methoden, die solche lokalen Ähnlichkeiten entdecken und alinieren, ohne die Sequenzen als ganzes zu alinieren. Lokale Alignments sind insbesondere bei der Datenbanksuche wichtig: Wenn man wissen will, welche Sequenzen in einer Datenbank mit einer gegebenen "Anfragesequenz" X verwandt sind, will man sich im Allgemeinen nicht auf globale Ähnlichkeiten beschränken, sondern man ist auch an Datenbank-Sequenzen interessiert, die lediglich lokale Ähnlichkeit zu X zeigen, da lokale Homologien oft schon Hinweise auf gemeinsame Struktur oder Funktion sein können.

Der in diesem Abschnitt beschriebene Ansatz für das lokale Alignment geht auf T. Smith und M. Waterman [3] zurück und ist als *Smith-Waterman-Algorithmus* bekannt. Der Einfachkeit halber betrachten wir nur lokale Alignments mit *linearen* Gap-Penalties. Formal kann man das Problem des lokalen Alignments folgendermaßen formulieren: Gesucht sind (1) *Teilsequenzen* $X' = X_{i_b} \dots X_{i_e}$ und $Y' = X_{j_b} \dots X_{j_e}$ von X bzw. Y, so dass der Needleman-Wunsch-Score des optimalen Alignments von X' und Y' maximal ist und (2) das optimale Alignment (im Sinne von Needleman und Wunsch) von X' und Y'. Dieses Alignment nennt man das *optimale lokale Alignment von X und Y*. Wenn wir Alignments als Pfade durch die Vergleichsmatrix betrachten, kann man das Problem auch so formulieren: Gesucht ist ein optimaler Pfad durch die Vergleichsmatrix, der an einer beliebigen Stelle $(i_b - 1, j_b - 1)$ beginnen und an einer beliebigen Stelle (i_e, j_e) enden kann. Beachte, dass der Pfad bereits in der Zelle $(i_b - 1, j_b - 1)$ beginnt, da ein Alignment der Länge K einem Pfad der Länge $K + 1$ entspricht (änlich, wie der Pfad zu einem globalen Alignment nicht bei $(1, 1)$, sondern bei $(0, 0)$ begint).



G C C A T
G C - T T

Abbildung 2.6: *Lokales* Alignment zweier Sequenzen X und Y. Das lokale Alignment entspricht einem (globalen) Alignment von zwei Teil-Sequenzen, hier dargestellt durch Großbuchstaben. Bereiche außerhalb des lokalen Alignments (in Kleinbuchstaben) werden nicht aliniert. Ein lokales Alignment entspricht einem Pfad durch die Vergleichsmatrix, der an einer beliebigen Position beginnen und an einer beliebigen Position enden kann.

Ein optimales lokales Alignment von X und Y kann mit einer einfachen Modifikation des Needleman-Wunsch-Algorithmus berechnet werden. Die Größe, die an jeder Stelle (i, j) der Vergleichsmatrix berechnet wird, ist der Score eines optimalen lokalen Alignments von X und Y, das an beliebigen Positionen X_{i_b} und Y_{j_b} beginnt und an den Positionen X_i und Y_j endet. Anders ausgedrückt: Gesucht ist ein optimaler Pfad durch die Vergleichsmatrix, der an einer beliebigen Zelle $(i_e - 1, j_e - 1)$ beginnt und in der Zelle (i, j) endet. Den Score dieses optimalen Pfades bezeichen wir wieder mit $S(i, j)$. Beachte, dass auch Alignments der Länge 0 bzw. Pfade der Länge 1 zugelassen sind. Um $S(i, j)$ zu berechnen betrachtet man wie beim globalen Alignment die drei möglichen "Vorgängerwerte" $S(i-1, j-1)$, $S(i-1, j)$ und $S(i, j-1)$. Beim lokalen Alignment gibt es jedoch noch eine vierte Möglichkeit, nämlich den Fall, dass der optimale Pfad bis (i, j) die Länge 1 hat, d.h. bei (i, j) beginnt, d.h. das optimale lokale Alignment kann die Länge 0 haben. Das ist der Fall, wenn alle drei obigen Werte kleiner als 0 sind. In diesem Fall kann man sich beim lokalen Alignment dafür entscheiden, ein "neues" Alignment anzufangen, dass dann die Länge 0 hat. In diesem Fall hat man daher $S(i, j) = 0$. Damit ist die Rekursionsgleichung für $S(i, j)$ gegeben durch

$$S(i,j) = \max \begin{cases} S(i-1, j-1) & + & s(X_i, Y_j) \\ S(i-1, j) & + & g \\ S(i, j-1) & + & g \\ 0, & & \end{cases} \tag{2.20}$$

und die Variable $Ptr(i, j)$ nimmt Werte zwischen 1 und 4 an, je nachdem, wo in (2.20) das Maximum angenommen wird.

Analog zum globalen Alignment werden die Werte $S(i, j)$ berechnet, indem man spaltenweise von links oben nach rechts unten durch die Vergleichsmatrix geht. Beim Initialisieren setzt man $S(i, 0) = 0$ und $S(0, j) = 0$ für $0 \leq i \leq m$ bzw. $0 \leq j \leq n$. Schließlich ist beim *Trace-Back* noch zu berücksichtigen, dass der optimale Pfad i. A. nicht bis (m, n) geht, sondern bis zu derjenigen Zelle (i_e, j_e) der Vergleichsmatrix, für die der Score $S(i, j)$ maximal ist. Bei der Berechnung der Werte $S(i, j)$ muss also zusätzlich die Zelle $(i_{\max}, j_{\max})$ aktualisiert werden, bei der $S(i, j)$ maximal ist. Das Trace-Back beginnt dann bei Zelle $(i_e, j_e) = (i_{\max}, j_{\max})$ und endet bei der ersten Zelle, für die $Ptr(i, j) = 4$ ist.

Kapitel 3

Multiples Sequenzalignment

Im vorherigen Kapitel hatten wir den *paarweisen* Vergleich von Protein- bzw. Nukleinsäuresequenzen betrachtet, d.h. den Vergleich von jeweils zwei Sequenzen. Ein Vergleich von *mehreren* Protein- oder Nukleinsäuresequenzen liefert sehr viel mehr Information als ein Vergleich von nur zwei Sequenzen. Wenn man zum Beispiel feststellt, dass ein Motiv in einer ganzen *Familie* von Proteinen konserviert ist, wird man eher davon ausgehen, dass dieses Motiv biologisch wichtig ist, als wenn man das Motiv nur in zwei Sequenzen findet. Daher interessiert man sich für Algorithmen, die ein Alignment von mehreren Sequenzen berechnen können.

3.1 Definition eines Multiplen Alignments

Die in Kapitel 2 eingeführten Begriffe und grundlegenden Algorithmen für das paarweise Sequenzalignment können direkt auf das multiple Alignment übertragen werden. Eingabedaten sind dabei N Sequenzen $X^{(1)}, \ldots, X^{(N)}$ über einem Alphabet $\mathcal{A}$, wobei L_i die Länge $i-$ten Sequenz $S^{(i)}$ ist:

$$\begin{aligned} X^{(1)} &= X_1^{(1)} X_2^{(1)} \ldots X_{L_1}^{(1)} \\ &\vdots \\ X^{(N)} &= X_1^{(N)} X_2^{(N)} \ldots X_{L_N}^{(N)} \end{aligned}$$

Analog zum paarweisen Alignment erhält man ein multiples Alignment, indem man *Gapzeichen* "$-$" in die Eingabesequenzen einfügt, so dass die entstehenden N *erweiterten* Sequenzen die gleiche Länge K haben. L bezeichnet die Länge der längsten Sequenz, d.h. $L = \max\{L_1, \ldots, L_N\}$. Die Länge des Alignments ist dabei eine Zahl K mit $L \leq K \leq \sum_i L_i$. In Worten: Das Alignment ist mindestens so lang wie die längste Sequenz (falls in diese Sequenz kein einziges Gapzeichen eingesetz wird) und höchstens so lang wie die Gesamtlänge aller

Sequenzen (falls keine zwei Zeichen der Sequenzen miteinander alignt werden, d.h. falls jedes Zeichen nur mit Gapzeichen alignt wird.

Formal kann man ein multiples Alignment als eine $N \times K$-Matrix

$$m = \begin{pmatrix} m_{1,1} & m_{1,2} & \dots & m_{1,K} \\ \vdots & \vdots & & \vdots \\ m_{N,1} & m_{N,2} & \dots & m_{N,K} \end{pmatrix} \tag{3.1}$$

mit Elementen

$$m_{i,j} \in \mathcal{A}^* = \mathcal{A} \cup \{-\}$$

definieren, die folgende zwei Eigenschaften hat:

1. Für jedes $i \in \{1, \dots, N\}$ gilt: Wenn man aus der ie i-ten Zeile von m die Gap-Zeichen "$-$" entfernt, erhält man die i-te Sequenz $X^{(i)}$.
2. Keine Spalte von m enthält nur Gap-Zeichen, d.h. jede Spalte enthält mindestens ein Zeichen aus dem Alphabet $\mathcal{A}$.

Die i-te Spalte des Alignments m wird dabei mit

$$m_i = \begin{pmatrix} m_{1,i} \\ \vdots \\ m_{N,i} \end{pmatrix} \tag{3.2}$$

bezeichnet.

3.2 Zielfunktionen für das Multiple Alignment

Als nächstes muss die Frage geklärt werden, wie eine sinnvolle *Zielfunktion* für das Multiple Alignment definiert werden kann, d.h. wie man die *Qualität* eines gegebenen Alignments bewerten kann. Hierfür kann der im letzten Kapitel diskutierte Ansatz für das Multiple Alignment verallgemeinert werden: Man berechnet die Summe aller Ähnlichkeitswerte $s(a, b)$ der alignten Aminosäuren bzw. Basen a und b und zieht davon eine *Penalty* für jedes Gap ab. Im einfachsten Fall betrachtet man dabei die Summe der Ähnlichkeitsweirte *aller* Paare von Basen bzw. Aminosäuren, die in einer Spalte des Alignments übereinander stehen. Dieses Bewertungsschema wird als *Sum-of-Pairs*-Zielfunktion bezeichnet.

Der Einfachkeit halber betrachten wir beim Multiplen Alignment nur *lineare* Gappenalties: Für jedes Gap der Länge k wird also eine Gappenalty von $k \cdot g$ berechnet, wobei g eine Konstante ist. Damit wird ein Gap der Länge k genau so bewertet wie k einzelne Gaps der Länge 1. Wie im letzten Kapitel kann man also so tun, als bestünde ein Gap der Länge k aus k Gaps der Länge 1, und statt Gaps als ganzes zu "bestrafen", kann man für jedes einzelne Gapzeichen eine Penalty von g berechnen. Damit kann man die Spalten m_i des Alignments unabhängig von den anderen Spalten mit einem Score $S(m_i)$ bewerten. Das

Gapzeichen kann man dabei als zusätzliches Zeichen im Alphabet $\mathcal{A}$ ansehen, wobei

$$s(a, -) = s(-, a) = -g \tag{3.3}$$

für jedes Zeichen $a \in \mathcal{A}$ gilt. Da in einem multiplen Alignment auch mehrere Gapzeichen in einer Spalte stehen können, definiert man außerdem

$$s(-, -) = 0 \tag{3.4}$$

Damit ist der Score $S(m_i)$ einer Spalte m_i im Alignment gegeben als

$$S(m_i) = \sum_{k<l} s(m_{k,i}, m_{l,i}), \tag{3.5}$$

und der Score $S(m)$ des gesamten Alignments m ist die Summe dieser *Spalten-Scores*:

$$S(m) = \sum_{1 \le i \le K} S(m_i). \tag{3.6}$$

Ein multiples Alignment A der Sequenzen $X^{(1)}, \ldots, X^{(N)}$ impliziert für jedes Paar von Sequenzen $(X^{(k)}, \ldots, X^{(l)})$ ein paarweises Alignment dass sozusagen in A "enthalten" ist. Man erhält dieses paarweise Alignment dadurch, dass man in A einfach alle Spalten bis auf die k-te und die l-te Spalte ignoriert und dann alle Gapzeichen ignoriert, die in übereinander stehen. Das so entstandene paarweise Alignment der Sequenzen $X^{(k)}$ und $X^{(l)}$ bezeichnet man als die *Projektion* von A auf die Sequenzen $X^{(k)}$ und $X^{(l)}$, man schreibt $A^{k,l}$. Für das multiple Alignment

$$A = \begin{pmatrix} A & T & T & G & T & A \\ A & T & T & A & T & - \\ A & T & - & A & T & - \\ T & T & - & G & A & G \\ - & G & - & G & T & A \end{pmatrix}$$

ist die Projektion $A^{3,5}$ auf die Sequenzen $X^{(3)}$ und $X^{(5)}$ also zum Beispiel gegeben als

$$A^{3,5} = \begin{pmatrix} A & T & A & T & - \\ - & G & G & T & A \end{pmatrix}$$

Damit kann man den Score $S(A)$ eines Multiplen Alignments auch als die Summe der Scores aller möglichen Projektionen $A^{k,l}$ berechnen:

$$S(A) = \sum_{k<l} S(A^{k,l}). \tag{3.7}$$

In anderen Worten: Wenn der Score eines Multiplen Alignments dadurch definiert ist, dass man die Ähnlichkeitswerte $s(a, b)$ aller Paare von Zeichen $a, b \in \mathcal{A}^*$ summiert, dann ist es egal, ob man erst die Summe für jede Spalte bildet und dann die so berechneten Spalten-Scores summiert wie in (3.6), oder ob man zuerst die Summe der Ähnlichkeitswerte für jedes Paar von Sequenzen $X^{(k)}$ und $X^{(l)}$ berechnet, und dann die Summe über alle Paare von Sequenzen bildet wie in (3.7).

Anhang A

Algorithmen

Das vorliegende Buch ist eine Einführung in die Algorithmen zur Analyse von Protein- und Nukleinsäuresequenzen. Für Nicht-Informatiker soll in diesem Abschnitt eine kurze Einführung in die Theorie der Algorithmen gegeben werden. Dabei wird zunächst anhand eines einfachen Beispiels geklärt, was ein *Algorithmus* ist, und was man unter der *Komplexität* eines Algorithmus versteht. Dann werden vier wichtige Typen von *Optimierungsalgorithmen* vorgestellt, die in der Bioinformatik von fundamentaler Bedeutung sind. Für eine gut lesbare systematische Einführung in die Theorie der Algorithmen wird auf das Buch von Cormen *et al.* verwiesen [1], aus dem wir auch einige Beispiele entnommen haben.

A.1 Algorithmen und Komplexität

Ein *Algorithmus* ist eine genau definierte Folge von Rechenschritten, mit der ein mathematisches Problem gelöst werden kann. Man kann einen Algorithmus daher als eine Art mathematisches "Kochrezept" auffassen: Man nehme einen vorgegebenen Satz von *Eingabedaten*, führe damit nacheinander diese und jene Berechnungen aus, mache mit dem Zwischenergebnis folgendes u.s.w., und zum Schluß gebe man folgende *Ausgabedaten* aus. Die durch einen Algorithmus vorgegebenen Rechenschritte können entweder von einem Menschen oder von einer Maschine ausgeführt werden. Algorithmen sind entwickelt worden längst bevor es Computer gab; z.B. der Algorithmus von Euklid zur Berechnung des größten gemeinsamen Teilers zweier ganzer Zahlen (3. Jahrhundert v. Chr.). Einige einfache Algorithmen kennt jeder aus der Schule, auch wenn er diese Rechenvorschriften nicht unter der Bezeichnung "Algorithmus" kennt. Z.B. ist das Verfahre zur schriftlichen Addition von mehrstelligen Zahlen ein einfacher Algorithmus. Er kann von einem Menschen ausgeführt werden, der einstellige Zahlen addieren kann und ein gewisses *Speichermedium* (z.B. Papier und Bleistift) zur Verfügung hat, um bei der Ausführung des Algorithmus entstehende Zwischenergebnisse abzuspeichern.

Ein Computer ist eine Maschine, die nicht selbst denken kann. Die Rechenvorschriften eines Algorithmus müssen ihm daher nach formal genau definierten Regeln mitgeteilt werden. Damit ein Algorithmus von einem Rechner ausgeführt werden kann, muß er daher in einer *Programmiersprache*, z.B. in $C, Perl$ oder $Java$ *implementiert* werden, d.h. die Rechenvorschriften des Algorithmus müssen in der entsprechenden Sprache geschrieben werden. Die Implementierung eines Algorithmus, also das entsprechende *Software-Programm* sollte daher von dem Algorithmus, selbst, also dem 'dahinter' liegenden mathematischen Konzept, unterschieden werden. Oft wird jedoch das Wort "Algorithmus" verwendet, wenn in Wirklichkeit "Software-Programm", gemeint ist. Das liegt wahrscheinlich daran, dass "Algorithmus" wissenschaftlicher klingt als "Software-Programm".

Bei einem Algorithmus interessiert man sich vor allem für zwei Eigenschaften, nämlich (a) für seine *Korrektheit* und (b) für seine *Komplexität*. Mit 'Korrektheit' meint man, dass der Algorithmus eine korrekte Lösung für die jeweilige Aufgabenstellung findet, und zwar für *alle* zulässigen Eingabedaten. Im Allgemeinen ist nicht offensichtlich, ob ein Algorithmus korrekt ist, hier muß also ein mathematischer *Beweis* geführt werden. Viele der in diesem Buch vorgestellten *Optimierungsalgorithmen* sind *nicht* korrekt, da sie für das jeweilige Optimierungsproblem nicht notwendig eine *optimale* Lösung finden, sondern im Allgemeinen nur eine *suboptimale*. Unter der *Komplexität* eines Algorithmus versteht man seinen Verbrauch an *Ressourcen* in Abhängigkeit von der Größe der Eingabedaten. Hierbei interessiert man sich (i) für die Rechenzeit, die er verbraucht und (ii) für seinen Bedarf an Speicherplatz. Man spricht dabei von Zeit- und Speicherplatzkomplexität. (Statt "Speicherplatzkomplexität" sagt man auch Raumkomplexität.)

Zur übersichtlichen und (mehr oder weniger) exakten Darstellung eines Algorithmus verwendet man gewöhnlich so genannten *Pseudocode*, der sich an Synthax einer Programmiersprache anlehnt. Gegenüber der wirklichen Implementierung des Algorithmus ist die Darstellung in Pseudocode jedoch stark vereinfacht, indem technische Einzelheiten weggelassen werden und Teilabschnitte des Programms abgekürzt in Worten zusammengefasst werden. Man kann des Pseudocode zu einem Algorithmus als eine grobe Anleitung zu seiner Implementierung ansehen, für die dann "nur" noch die fehlenden Einzelheiten nachgetragen werden müssen. Zur Beschreibung von Algorithmen mit Pseudocode gibt es keine festen Regeln. Sowohl die Programmiersprache, an die sich die Beschreibung anlehnt, als auch der Grad der Detaillierung sind beliebig. In diesem Skript lehnen wir uns an die Programmiersprachen C und $Perl$ an.

A.1.1 Beispiel: Sortieren einer Liste durch INSERTION_SORT

Bei der Entwicklung von Computer-Programmen und anderswo hat man häufig das Problem, eine Liste von Objekten gemäß einem gegebenen Kriterium zu *sortieren*. Daher ist man an effizienten Algorithmen interessiert, die dieses Problem lösen. Der Algorithmus INSERTION_SORT ist eine einfache – wenn auch keine sehr effiziente – Methode, eine Liste zu sortieren. Wir beschreiben diesen Algo-

```
Algorithmus INSERTION_SORT(L = L[1] ... L[n]) {
    M = {} ;          /* d.h. M ist eine Liste der Länge 0 */
    n = Länge von L ;
    n' = 0 ;          /* Länge von M */
    for (i = 1 ; i ≤ n ; i++) {
        /* Finde passende Position p in M, an der
        aktueller Wert L[i] einsortiert werden soll */
        p = 1 ;
        while (L[i] < M[p] and p ≤ n')
            p++ ;
        /* Verschiebe M von Position p an um 1 nach rechts
        und Sortiere L[i] an Position p in M ein */
        for (j = n' + 1 ; j > p ; j--)
            M[j] = M[j - 1] ;
        M[p] = L[i];
        n'++;
    }
    RETURN M ;
}
```

Abbildung A.1: Ausführliche Pseudocode-Beschreibung des Algorithmus **INSERTION_SORT** zum Sortieren einer Liste L.

rithmus zunächst in Worten und stellen uns vor, wir wollen z.B. einen Stapel von Spielkarten sortieren. Unsere Eingabedaten sind also eine Liste von nichtsortierten Objekten (Spielkarten in unserem Fall), die Ausgabe des Algorithmus ist eine sortierte Liste der selben Objekte.

Bei INSERTION_SORT hat man zwei Listen L und M von Objekten. Zu Beginn besteht L aus der eingegebenen Liste, d.h. L ist eine Liste von nichtsortierten Objekten. Die Liste M ist zu Anfang leer, d.h. sie enthält kein einziges Element. Der Algorithmus nimmt ein Objekt nach dem anderen aus der Liste L und setzt es in die Liste M ein. Das macht er so, dass das Objekt, das jeweils neu in M eingesetzt wird, an der richtigen Stelle eingesetz wird – d.h. so, dass M zu jedem Zeitpunkt sortiert bleibt. Zum Schluß ist die Liste L leer, und M enthält alle Objekte der Eingabe-Liste in sortierter Reihenfolge. Damit ist unser Problem also gelöst.

Abbildung A.1 zeigt eine detaillierte Beschreibung dieses Algorithmus in Pesudocode. Fast alle notwendigen Schritte sind beschrieben. Für eine Implementierung des Algorithmus in C müßten nur noch wenige Zeilen hinzugefügt werden, z.B. für das Einlesen der Eingabedaten und für die Ausgabe des Ergebnisses. Abbildung A.2 zeigt eine knappere Pseudocode-Beschreibung desselben Algorithmus. Hier sind die meisten Einzelheiten nicht wie in einem Computer-Programm formal notiert, sondern statt dessen verbal umschrieben.

Algorithmus INSERTION_SORT($L = L[1] \dots L[n]$) **{**
  $M = \{\}$;
  $n =$ Länge von L;
  for ($i = 1$; $i \leq n$; $i++$) **{**
    – Gehe von links nach rechts durch M, um passende Position p zu finden, an der aktueller Wert $L[i]$ in M einsortiert werden soll ;
    – Sortiere $L[i]$ an Position p in M ein ;
  }
  RETURN M ;
}

Abbildung A.2: Pseudocode-Beschreibung des Algorithmus zum Sortieren einer Liste L.

A.1.2 Komplexität von Algorithmen

Wenn man einen Algorithmus beschreibt, will man in der Regel angeben, wieviel Rechenzeit und Speicherplatz er benötigt, um ein gegebenes Problem zu lösen. Diese Angaben kann man natürlich nicht in Millisekunden oder Kilobytes machen, da der Verbrauch Zeit und Speicherplatz von vielen Faktoren abhängt: vor allem natürlich von der Größe und Zusammensetzung der Eingabedaten ab, aber auch von der verwendeten Hardware, von Details der Implementierung u.s.w. Man ist daher daran interessiert, einen mathematischen Zusammenhang zwischen der Größe der Eingabedaten und dem Verbrauch von Zeit und Speicherplatz anzugeben, damit man abschätzen kann, wie der Algorithmus sich auf großen Datenmengen verhält. Auch diesen Zusammenhang kann man meistens nicht exakt beschreiben, man kann aber gewisse "obere Grenzen" für den Verbrauch an Ressourcen angeben. Dies soll am Beispiel des Algorithmus INSERTION_SORT veranschaulicht werden.

Wir betrachten zunächst die Rechenzeit, die der Algorithmus benötigt, um eine Liste mit n Objekten zu sortieren. Dazu betrachten wir die Rechenschritte, die insgesamt ausgeführt werden müssen. Der Algorithmus nimmt nacheinander Objekte $L[i]$ aus der ursprünglich vorgegebenen Liste L und sortiert sie in die neu entstehende Liste M ein. Insgesamt werden also n Objekte aus der Liste L bearbeitet. Wieviele Schritte müssen nun durchgeführt werden, um ein "neues" Objekt $L[i]$ in die Liste M einzusetzen? Die Liste M ist so konstruiert, dass sie zu jedem Zeitpunkt sortiert ist. Wir müssen also die richtige Stelle p in M finden, um das neue Objekt $L[9]$ einzusortieren. Dafür geht INSERTION_SORT von links nach rechts durch die Liste M, bis die richtige Stelle gefunden ist. M kann selber eine Länge von bis zu n haben, also sind im schlimmsten Fall n Schritte nötig, um die richtige Position zu finden. Beachte, dass es nicht möglich ist, genau vorherzusagen, wie lange man tatsächlich die Liste M "entlanggehen" muß, um die korrekte Position für das neue Objekt zu finden. Dies hängt davon

ab, ob bzw. wie weit die ursprüngliche Liste schon sortiert war. Daher gibt man hier an, wie viele Schritte der Algorithmus "im schlimmsten Fall" durchführen muß (sog. *worst-case*-Komplexität).

Ist die Position gefunden, an der das neue Objekt einsortiert werden soll, müssen in der Liste M alle Objekte von dieser Position an um eine Position nach oben verschoben werden, damit Platz für das neue Objekt geschaffen wird. Das kann ebenfalls bis zu n Rechenschritte benötigen. Für *jedes* neue Objekt müssen also bis zu $2 \cdot n$ Schritte ausgeführt werden, um das x in M einzusortieren. Da wir insgesamt n Objekte aus der Liste L in M einsortieren müssen, werden also bis zu $2 \cdot n^2$ Rechenschritte durchgeführt. Hierbei ist der Faktor 2 relativ unwichtig. Die Tatsache, dass die Laufzeit des Algorithmus – im schlimmsten Fall – proportional zu n^2 ist, ist jedoch entscheidend. Hieraus kann man abschätzen, wie sich die Laufzeit eines Programms entwickeln wird, wenn man die Größe der Eingabedaten ändert. Wird die Länge n der eingegebenen Liste L zum Beispiel verdoppelt, dann kann man erwarten, dass sich die Laufzeit des Programms etwa vervierfacht, wird n verzehnfacht, dann wird sich die Laufzeit etwa verhundertfachen etc. Etwas allgemeiner kann man sagen: Es existiert eine Konstanten c, so dass für die Zahl der Rechenschritte $Z(n)$ die unser Algorithmus braucht, um eine Liste der Länge n zu sortieren, für jedes beliebige n gilt:

$$Z(n) \leq c \cdot n^2$$

Hierfür schreibt man auch: Die Zeitkomplexität des Algorithmus ist $O(n^2)$.

Um allgemein auszudrücken, wie die Laufzeit eines Algorithmus $\mathcal{A}$ durch eine Funktion $f : N \to R$ abgeschätzt werden kann, definiert man: Die Zeitkomplexität von $\mathcal{A}$ ist $O(f(n))$, wenn es eine Konstante c gibt, so dass für alle $n \in N$ gilt: Die Zahl der Rechenschritte $Z_{\mathcal{A}}(n)$, die $\mathcal{A}$ zur Bearbeitung eines Datensatzes der Größe n benötigt, kann durch

$$Z_{\mathcal{A}}(n) \leq c \cdot f(n). \tag{A.1}$$

abgeschätzt werden. Analog definiert man die *Speicherplatzkomplexität* von $\mathcal{A}$. Als kleine Verallgemeinerung soll noch der Fall betrachtet werden, dass die Größe der Eingabendaten nicht durch eine einzelne Zahl n beschrieben werden kann. Wenn $\mathcal{A}$ z.B. Ein Algorithmus ist, der k Proteinsequenzen miteinander vergleicht, dann wird die Laufzeit von $\mathcal{A}$ im Allgemeinen von den Längen *aller* Sequeqenzen, also von k Zahlen abhängen. In diesem Fall hat man eine Funktion

$$f : N^k \to R,$$

und man sagt, dass die Zeitkomplexität von $\mathcal{A}$ durch $O(f(n_1, \ldots, n_k)$ gegeben ist, wenn man eine Konstante c hat, so dass

$$Z_{\mathcal{A}}(n) \leq c \cdot f(n_1, \ldots, n_k)$$

für alle $n \in N$ gilt.

Literaturverzeichnis

[1] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, USA, London, England, 2001.

[2] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[3] T. F. Smith and M. S. Waterman. Comparison of biosequences. *Advances in Applied Mathematics*, 2:482–489, 1981.