Algorithmen in der Biologie

Dr. Hans-Joachim Böckenhauer
Dr. Dennis Komm

Zusammenfassung des 4. Abends

Zürich, 14. Mai 2014

1 DNA-Sequenzierung

In diesem Abschnitt wollen wir einen kurzen Überblick geben über die verschiedenen Ansätze zur Sequenzierung langer DNA-Moleküle. Das Ziel ist es also, zu einem gegebenen DNA-Molekül die zugehörige Basensequenz als einen String über `A`, `C`, `G`, `T` zu bestimmen. Wir haben bereits am ersten Abend gesehen, dass dies für kurze DNA-Moleküle mit einer Sequenz von 500–1000 Basen mit Labormethoden wie der Kettenabbruchmethode möglich ist. Allerdings kann ein DNA-Molekül, auf dem die gesamte Erbinformation eines Organismus abgespeichert ist, eine Grössenordnung von einer Million Basen erreichen. Ein Ansatz zur Sequenzierung solch langer DNA-Moleküle ist die sogenannte *Shotgun-Sequenzierung*, die nach dem folgenden Schema arbeitet.

Eingabe: Ein DNA-Molekül $\mathcal{D}$.
Schritt 1: Vervielfältige $\mathcal{D}$, zum Beispiel mit der Polymerase-Kettenreaktion.
Schritt 2: Zerlege die Kopien von $\mathcal{D}$ in zufällige kleine Fragmente, die einander überlappen. Dies kann zum Beispiel durch Erwärmen und Schütteln erreicht werden.
Schritt 3: Sequenziere die Fragmente im Labor, zum Beispiel mit der Kettenabbruchmethode. Falls ein Fragment zu lang sein sollte, um vollständig sequenziert zu werden, kann es auch ausreichen, nur die beiden Enden zu sequenzieren. Erhalte hiermit die Teilstrings $f_1, \ldots, f_n$ des gesuchten Strings.
Schritt 4: Löse das *Fragment-Assembly-Problem*, das heisst setze die Fragmente so zusammen, dass sich eine Näherung der gesuchten Sequenz von $\mathcal{D}$ ergibt. Dies ist der aufwändigste und schwierigste Teil, der sich in drei Phasen untergliedern lässt:

1. Bestimme die paarweisen *Overlaps* (Überlappungen) der Fragmente, berechne also, wie weit sich zwei Fragmente „übereinanderschieben“ lassen. Dies ist effizient möglich, zum Beispiel mit einer Abwandlung des Alignment-Verfahrens, das wir am zweiten Abend kennengelernt haben.

2. Nutze diese Overlap-Information, um ein *Layout* zu berechnen, also eine Anordnung der Fragmente. Dies führt zu einem schwierigen kombinatorischen Problem, auf das wir noch eingehen werden.

3. In dem Layout kann es jetzt vorkommen, dass an einigen Stellen die übereinander platzierten Fragmente nicht vollständig übereinstimmen. Bestimme also aus dem Layout einen *Consensus*, also einen eindeutigen String, der dann die berechnete Näherung der Sequenz von $\mathcal{D}$ darstellt.

Ausgabe: Der berechnete Consensus-String.

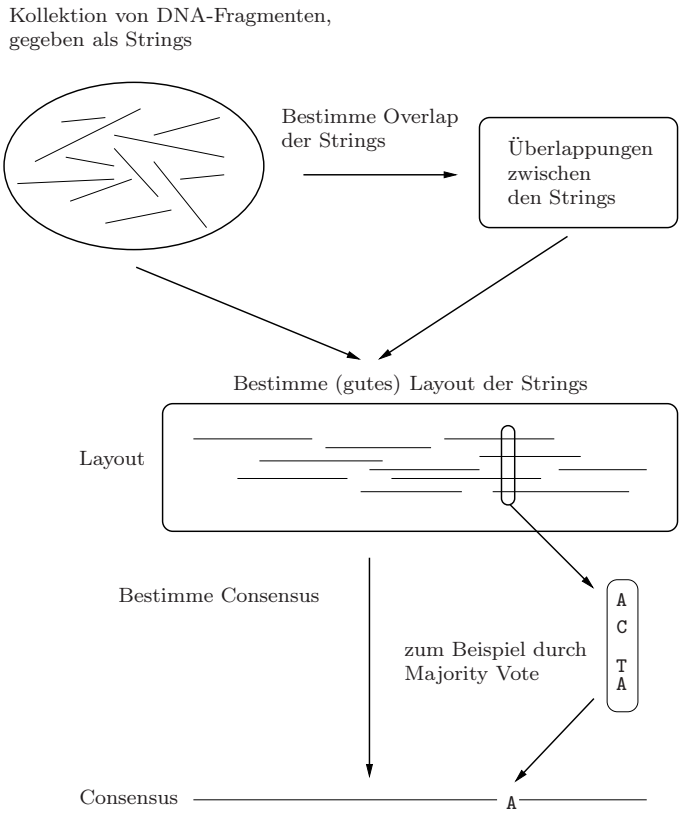


Abbildung 1. Schematische Darstellung der Shotgun-Sequenzierung

Dieses Verfahren ist graphisch in Abbildung 1 dargestellt.

2 Layout-Berechnung

Die Aufgabe, aus den gegebenen Fragmentsequenzen ein passendes Layout zu berechnen, aus dem die Gesamtsequenz erkennbar wird, führt zu schwierigen Berechnungsproblemen. Wir wollen im Weiteren nur die folgende idealisierte Variante betrachten, die allein schon so schwierig ist, dass wir in der Regel keine exakte Lösung mehr berechnen können.

Wir nehmen also im Folgenden an, dass alle Fragmente korrekt sequenziert wurden und zusammen die komplette DNA-Sequenz überdecken. Weiter nehmen wir an, dass die gesuchte Sequenz diejenige ist, die alle Fragmente exakt enthält und dabei die kleinstmögliche Länge aufweist. Eine solche Sequenz nennen wir einen *kürzesten Superstring* (shortest common superstring, SCS) der Fragmente. Unser Berechnungsproblem besteht nun darin, zu einer gegebenen Menge von Fragmenten den SCS zu finden. Dieses Berechnungsproblem ist nach heutigem Wissensstand nicht effizient lösbar, jeder bekannte exakte Algorithmus benötigt exponentielle Zeitkomplexität, und es ist auch nicht anzunehmen, dass irgendwann ein effizienter Algorithmus gefunden wird. Deshalb versuchen wir, zumindest eine Näherungslösung zu finden. Hierfür verwenden wir den folgenden einfachen Algorithmus: Finde zwei Strings mit maximalem Overlap, verschmelze diese zu einem neuen String und wiederhole dies solange, bis nur noch ein String übrig ist, dies ist dann ein Superstring für die gegebenen Fragmente.

Beispiel 1. Wir betrachten die Fragmente `ATATAA`, `CATA`, `AAGGG`, `AATCA` und `AACAT`. Dann ergeben sich die folgenden paarweisen Overlap-Längen.

	ATATAA	CATA	AAGGG	AATCA	AACAT
ATATAA	1	0	2	2	2
CATA	3	0	1	1	1
AAGGG	0	0	0	0	0
AATCA	1	2	1	1	1
AACAT	2	3	0	0	0

Ein maximaler Overlap ist also der zwischen den Fragmenten `CATA` und `ATATAA`. Wir verschmelzen also diese beiden Strings zu einem neuen String `CATATAA` und löschen die beiden Strings `CATA` und `ATATAA`. Damit ergibt sich die folgende Tabelle von Overlap-Längen:

	CATATAA	AAGGG	AATCA	AACAT
CATATAA	0	2	2	2
AAGGG	0	0	0	0
AATCA	2	1	1	1
AACAT	3	0	0	0

Hier gibt es nun einen maximalen Overlap zwischen `AACAT` und `CATATAA`. Das Verschmelzen dieser beiden Strings liefert den neuen String `AACATATAA` und es bleiben noch die folgenden Overlap-Längen übrig.

	AACATATAA	AAGGG	AATCA
AACATATAA	2	2	2
AAGGG	0	0	0
AATCA	1	1	1

Ein maximaler Overlap ist nun der von `AACATATAA` und `AAGGG`, es ergibt sich der neue String `AACATATAAGGG`, vor den wir noch den verbliebenen String `AATCA` hängen können, um das Endergebnis `AATCAACATATAAGGG` zu erhalten.

In diesem Beispiel berechnet unser Algorithmus sogar die optimale Lösung. Dies muss aber nicht immer so sein, wie das folgende Beispiel zeigt.

Beispiel 2. Wir betrachten die drei Fragmente `CATATAT`, `TATATA` und `ATATATC`. Der grösste Overlap tritt hier zwischen `CATATAT` und `ATATATC` auf, also verschmilzt unser Algorithmus diese zu dem String `CATATATC` und kann dann den dritten String `TATATA` nur noch vorn oder hinten anhängen und erhält damit zum Beispiel die Lösung `TATATACATATATC` der Länge 14. Hingegen hat die optimale Lösung `CATATATATC` nur die Länge 10.

Dieses Beispiel lässt sich leicht so verallgemeinern, dass die berechnete Lösung sogar fast doppelt so lang ist wie die optimale. Auch wenn kein schwereres Beispiel für diesen Algorithmus bekannt ist, schafft man es nur zu zeigen, dass eine Lösung garantiert werden kann, die nicht länger als 3.5-mal so lang wie das Optimum ist.

Wir bemerken, dass dieser Algorithmus zur Klasse der sogenannten *Greedy-Algorithmen* gehört, wie auch der am zweiten Abend vorgestellte Algorithmus zur Spannbaumberechnung. Greedy-Algorithmen treffen in jedem Schritt eine Wahl, die *lokal* am meisten hilft, in Richtung des Ziels voranzukommen. Hier ist dies der maximale Overlap, im Spannbaum-Algorithmus war dies die Auswahl einer kürzesten Kante. Nur für wenige Probleme (wie beispielsweise die Spannbaumberechnung) führt diese Strategie wirklich zur *global* optimalen Lösung, aber wegen der Einfachheit dieser Strategie wird sie oft zum Entwurf von Näherungsverfahren eingesetzt.

Wir stellen aber fest, dass die Berechnung eines SCS und selbst einer brauchbaren Näherung schwierig ist. Zusätzlich gibt es zahlreiche weitere Hürden bei der Berechnung eines aussagekräftigen Layouts, wie zum Beispiel die folgenden.

- Sequenzierungsfehler bei der Sequenzierung der Fragmente
- Zusammenlagerung von Fragmenten, die im ursprünglichen Molekül weiter voneinander entfernt waren (sogenannte *Chimären*)
- Unvollständige Überdeckung der DNA durch die Fragmente
- Unbekannte Orientierung der Fragmente
- Teilstrings, die in der DNA an mehreren Stellen vorkommen (*Repeats*)

All diese Probleme führen dazu, dass die Berechnung eines sinnvollen Layouts ohne Zusatzwissen sehr schwierig ist und höchstens für relativ kurze DNA-Moleküle durchgeführt werden kann.

3 Physikalische Kartierung

Um sich jetzt für das Layout-Problem zusätzliche Information zu verschaffen, die das Anordnen der Fragmente vereinfachen kann, nutzt man häufig die Methode der *physikalischen Kartierung*. Eine *physikalische Karte* (physical mapping) der DNA enthält eine Menge von genetischen *Markern* (kurzen bekannten Teilstrings) und ihre genauen Positionen in der DNA. Als Beispiel für solche Marker verwendet man in der Regel die Restriktionsstellen eines Restriktionsenzyms. *Restriktionsenzyme* sind spezielle Proteine, die ein DNA-Molekül an bestimmten Stellen schneiden können. Ein Beispiel hierfür ist das Enzym *HindIII*, das eine DNA genau an den Stellen schneiden kann, an denen die Sequenz `AAGCTT` vorliegt (genauer gesagt, schneidet es die DNA hinter dem ersten `A` dieses Musters).

Unser Ziel ist es nun, in einem gegebenen DNA-Molekül die genauen Positionen aller Restriktionsstellen eines bestimmten Restriktionsenzyms zu bestimmen. Damit kennen wir dann die genaue Position dieses kurzen Musters in der gesamten DNA und dieses Wissen kann uns helfen, das Layout-Problem zu lösen.

4 Das Partial-Digest-Problem

Eine Möglichkeit, die Restriktionsstellen zu bestimmen, liefert der sogenannte *Partial-Digest-Ansatz*. Dieses Verfahren lässt sich wie folgt zusammenfassen, es ist in Abbildung 2 graphisch dargestellt.

Eingabe: Ein DNA-Molekül $\mathcal{D}$ und ein Restriktionsenzym $\mathcal{A}$.
Schritt 1: Erzeuge viele Kopien von $\mathcal{D}$.
Schritt 2: Teile diese Kopien auf eine Anzahl von Reagenzgläser auf und füge jeweils das Restriktionsenzym $\mathcal{A}$ hinzu. Lasse das Enzym unterschiedlich lange auf die DNA einwirken. Dadurch entstehen unterschiedlich lange Fragmente von $\mathcal{D}$, die alle an den Restriktionsstellen von $\mathcal{A}$ geschnitten sind.
Schritt 3: Bestimme die Längen dieser Fragmente.
Schritt 4: Berechne aus diesen Längen die Positionen der Restriktionsstellen in $\mathcal{D}$.

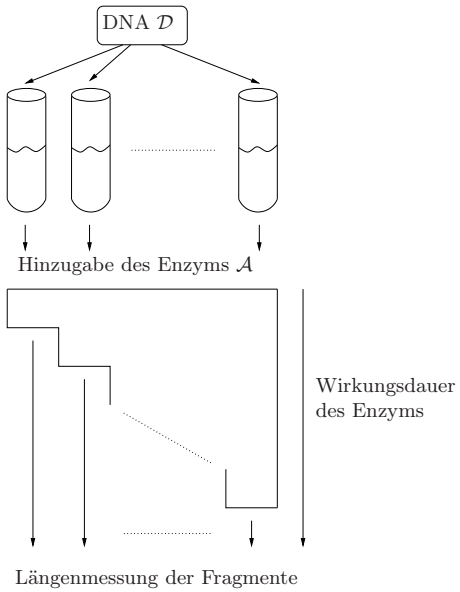
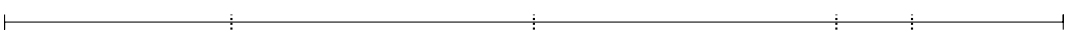


Abbildung 2. Schematische Darstellung des Partial-Digest-Verfahrens

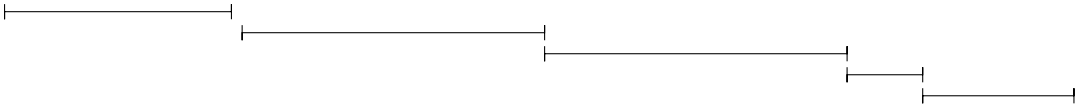
Es bleibt die Frage, wie man aus den Längen der Fragmente die Positionen der Restriktionsstellen in der DNA berechnen kann. Dieses Problem ist als *Partial-Digest-Problem* (PDP) bekannt. Wir nehmen hierfür wieder einmal an, dass wir ideale Daten zur Verfügung haben, dass also unsere Experimente uns die genauen Längen der Fragmente liefern. Weiterhin kann es verschiedene solche Fragmente geben, die exakt die gleiche Länge haben. In diesem Fall nehmen wir weiterhin an, dass wir zu jeder Länge auch wissen, wieviele verschiedene Fragmente diese Länge haben. Damit ergibt sich das folgende abstrakte Berechnungsproblem, dessen Eingabe eine sogenannte *Multimenge*, also eine Menge, die dasselbe Element mehrmals enthalten kann, ist.

Eingabe: Eine Multimenge von Längen.
Ausgabe: Die Positionen der Restriktionsstellen.

Wir haben also eine Multimenge von verschiedenen Längen und wissen, dass diese gewissen Fragmentlängen der gegebenen DNA entsprechen, die entstehen, wenn diese an Restriktionsstellen zerschnitten wird. Die Positionen der Restriktionsstellen könnten nun aus den kürzesten Fragmenten (also denen in dem Reagenzglas mit der längsten Einwirkungszeit) berechnet werden. Wenn wir also zum Beispiel die gepunkteten Restriktionsstellen



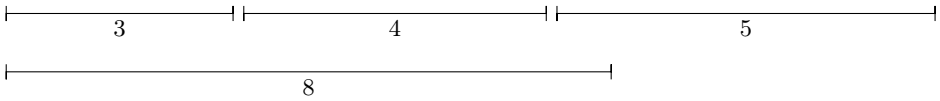
annehmen erhalten wir, unter anderem, die folgenden Fragmente.



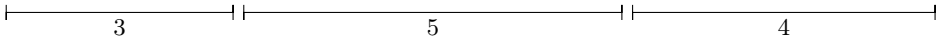
Das Problem ist allerdings, dass die Information über die Reihenfolge dabei verloren geht. Hierbei hilft uns nun, dass wir nicht nur alle Teilstrings erhalten haben, die durch das Schneiden an direkt aufeinander folgenden Restriktionsstellen entstehen, sondern auch

alle solchen, die durch das Schneiden an Restriktionsstellen entstehen, die einen grösseren Abstand besitzen (als Konsequenz einer geringeren Einwirkungszeit).

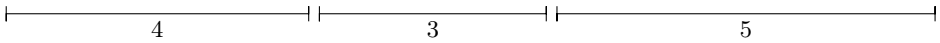
Beispiel 3. Um das Problem beispielhaft zu untersuchen, betrachten wir kleine Zahlen. Nehmen an, der ganze String hat die Länge 12 und wir haben die Teilstrings



erhalten. Nun wissen wir also, dass wir die Restriktionsstellen berechnen können, indem wir die Längen der drei kürzesten Strings anordnen. Die Reihenfolge 3, 4, 5 der Längen kann dabei keine gültige Lösung sein, denn sonst wäre kein String der Länge 8 entstanden. Eine mögliche Lösung ist zum Beispiel gegeben durch



und wir schlussfolgern, dass die Restriktionsstellen sich an den Positionen 3 und 8 befinden können. Allerdings entspricht auch die Anordnung



einer Lösung und es gibt noch weitere.

Es ist wie in obigem Beispiel auch allgemein schnell ersichtlich, dass es nicht nur eine Lösung für eine Instanz des PDP gibt, so dass unter Umständen nach mehreren (im schlimmsten Fall allen) gesucht werden muss. Wir vereinfachen nun weiter und gehen davon aus, dass es ausreichend ist, wenn eine Lösung gefunden wird.

Trotz all dieser Abstraktionen und Vereinfachungen bleibt das PDP ein schweres Problem. Die Eingabelänge wächst offensichtlich mit der Anzahl der Fragmente, die wir mit n bezeichnen, womit sich $n-1$ Restriktionsstellen ergeben. Ein naiver Ansatz, alle Reihenfolgen der Fragmente, die von direkt aufeinanderfolgenden Restriktionsstellen stammen, auszuprobieren, ist extrem aufwändig. Für die erste Stelle kommt zunächst einmal jedes Fragment in Frage, für jede dieser n Möglichkeiten kann jedes der verbleibenden $n-1$ Fragmente an der zweiten Stelle sein etc. Insgesamt kommen wir somit auf

$$n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 1 = n!$$

Möglichkeiten. Diese Funktion wächst noch schneller als eine exponentielle: während 2^{50} einer Zahl mit 15 Dezimalstellen entspricht, sind es bei 50! bereits 64 Stellen.

Wieder verfolgen wir, um die Laufzeit zu reduzieren, nun die Idee, die Lösung schrittweise aufzubauen, indem wir zunächst kleinere Teil-Instanzen betrachten. Dies ist ein ähnlicher Ansatz wie bei der bereits bekannten dynamischen Programmierung, allerdings gibt es wesentliche Unterschiede. Die im Folgenden beschriebene Methode probiert eine Lösung des PDP zu finden, indem sie die Längen der Grösse nach (kleiner werdend) anzuordnen versucht. Es kann gezeigt werden, dass es dann immer ausreichend ist, die Längen entweder links oder rechts einzufügen. Anders als bei der dynamischen Programmierung können hierbei in Zwischenschritten allerdings „Fehler“ passieren, das heisst Anordnungen konstruiert werden, die zu keiner Lösung führen. In einem solchen Fall muss unser Algorithmus ein paar der letzten Schritte rückgängig machen, weswegen dieses Prinzip als *Backtracking* (zu übersetzen beispielsweise mit „Zurückverfolgung“) bekannt ist.

Um den folgenden Backtracking-Algorithmus genauer zu beschreiben, brauchen wir das Konzept eines sogenannten *Stacks*. Hierbei handelt es sich um einen besonderen Speicher,

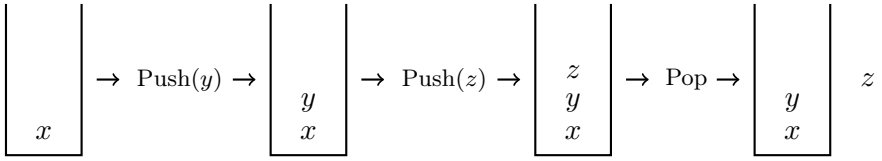


Abbildung 3. Wenn die Daten x, y und z mittels Push in einem Stack gespeichert werden, so wird anschliessend mit Pop das Datum z zurückgegeben und aus dem Stack gelöscht

auf den mit zwei Funktionen "Push" (schreiben) und "Pop" (lesen) zugegriffen werden kann. Das Besondere ist hierbei, dass immer das Datum gelesen wird, welches als letztes gespeichert wurde. Ein Stack ist schematisch in Abbildung 3 dargestellt. Ein weiteres Prinzip, das wir brauchen, ist das der *Rekursion*. Hier ruft ein Programm sich selber mit veränderten Parametern auf, wobei wir natürlich gewährleisten müssen, dass dies nicht unendlich oft passiert.

Eingabe: Eine Multimenge A mit n Zahlen, die Längen entsprechen.
Schritt 1: Initialisiere S als leeren Stack, initialisiere X als leere Menge, die später alle Restriktionsstellen enthalten soll.
Schritt 2: Sortiere A.
Schritt 3: Entferne das grösste Element aus A (hierbei handelt es sich um den ganzen String) und füge es zusammen mit 0 der Menge X zu.
Schritt 4: Rufe die Prozedur „FragmentEinfügen" rekursiv auf.

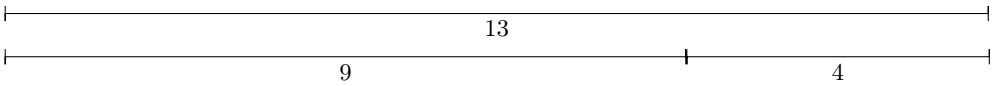
Die eigentliche Arbeit wird nun von der folgenden Prozedur übernommen.

```
Prozedur FragmentEinfügen
Eingabe: (Multi-) Mengen A und X, Stack S.
Teste, ob A leer ist.
Wenn ja
    Ausgabe: „Lösung gefunden: X."
    Ende
Setze MAX auf das grösste Element aus A.
Teste, ob MAX links eingesetzt werden kann.
Wenn ja
    Verändere A, X und S entsprechend
    FragmentEinfügen mit A, X und S
Sonst
   Teste, ob MAX rechts eingesetzt werden kann.
   Wenn ja
       Verändere A, X und S entsprechend
       FragmentEinfügen mit A, X und S
   Sonst
      Teste, ob noch ein Element in S vorhanden ist
      Wenn ja      (Wir haben noch nicht alle Möglichkeiten ausprobiert)
          Verändere A, X und S entsprechend
          Return      (Hier erfolgt nun ein Backtracking-Schritt)
      Sonst
          Ausgabe: „Keine Lösung gefunden."
          Ende
```

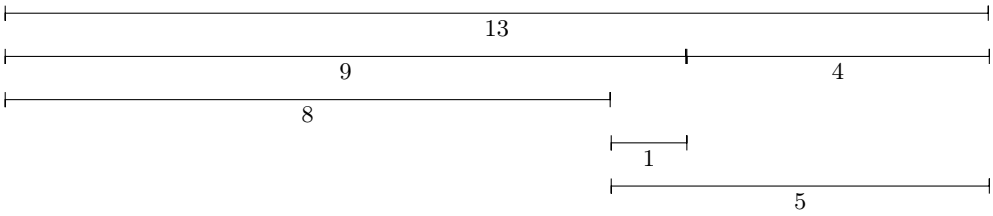
Hierbei ist zu beachten, dass der Befehl **Return** (anders als **Ende**) nicht die gesamte Ausführung stoppt, sondern den jeweiligen Aufruf von **FragmentEinfügen**. Mit „verändere A, X und S entsprechend“ ist gemeint, dass, wenn eine Länge links oder rechts hinzugefügt werden konnte, die benötigten Fragmente aus A entfernt werden, die neuen Restriktionsstellen zu X hinzugefügt werden und das hinzugefügte Fragment zu S hinzugefügt wird. Bei einem Backtracking-Schritt, wird das letzte Element von S genommen und die Situation von A und X wiederhergestellt, die vorlag, bevor dieses betrachtet wurde. Veranschaulichen wir nun die Arbeit des Algorithmus auf einer konkreten Eingabe für PDP, die nicht sofort zu einer Lösung führt.

Beispiel 4. Die gegebene Multi-Menge sei $A = \{1, 3, 4, 4, 5, 6, 7, 8, 9, 13\}$. Zunächst initialisiert unser Algorithmus S und X und sortiert die Multi-Menge A. Dann nimmt er 13 und 0 in X auf. Anschliessend beginnt der rekursive Aufruf der Funktion FragmentEinfügen.

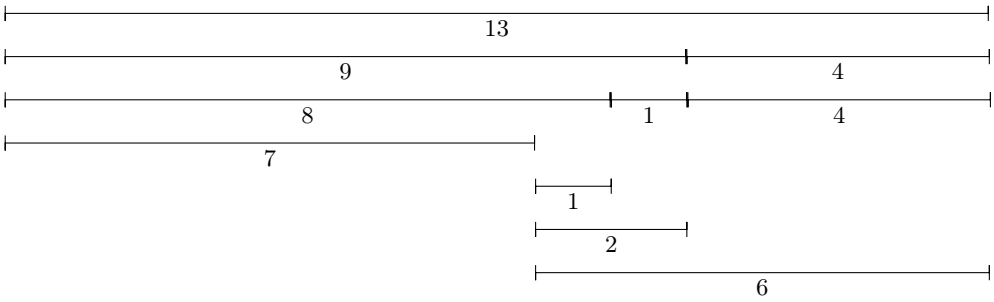
Zunächst wird FragmentEinfügen mit $S = ()$, $X = \{0, 13\}$ und $A = \{1, 3, 4, 4, 5, 6, 7, 8, 9\}$ ausgeführt. Da 9 das grösste Element in A ist, wird probiert, es links aussen einzufügen. Wenn dies funktionieren soll, so muss es auch eine verbleibende Länge $13 - 9 = 4$ geben, was der Fall ist. Also ist eine Zwischenlösung durch die Anordnung



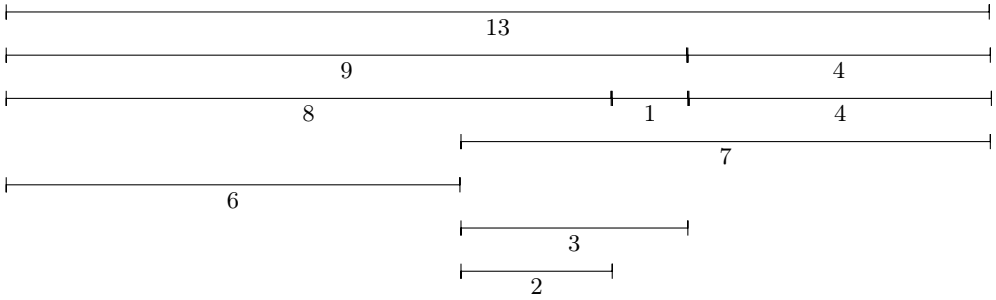
gegeben. Im Stack S wird gespeichert, dass wir eine Restriktionsstelle an der Position 9 vermuten. Somit ergeben sich die Werte $S = (9)$, $X = \{0, 9, 13\}$ und $A = \{1, 3, 4, 4, 5, 6, 7, 8\}$, mit denen erneut FragmentEinfügen aufgerufen wird. Da jetzt 8 das grösste Element in A ist, wird versucht, dieses ebenfalls links einzufügen. Damit dies wiederum funktioniert, müssen zusätzlich die Längen 1 und 5 in A enthalten sein, was zutrifft und zu



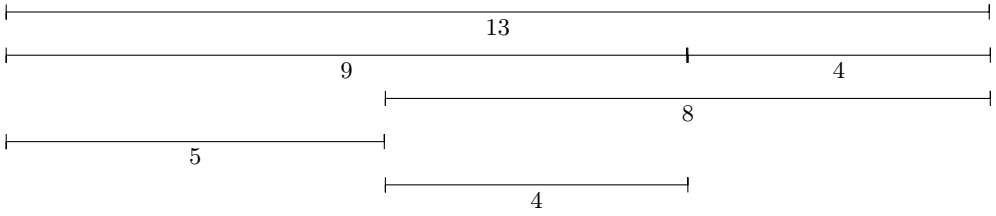
führt. Wieder werden die Variablen angepasst, was zu $S = (9, 8)$, $X = \{0, 8, 9, 13\}$ und $A = \{3, 4, 6, 7\}$ führt. Mit diesen Werten wird FragmentEinfügen wieder ausgeführt und nun ist 7 die grösste Länge in A. Allerdings können wir diese nicht links einfügen, da wir dann ferner die Längen 1, 2 und 6 bräuchten, aber weder 1 noch 2 in A enthalten ist.



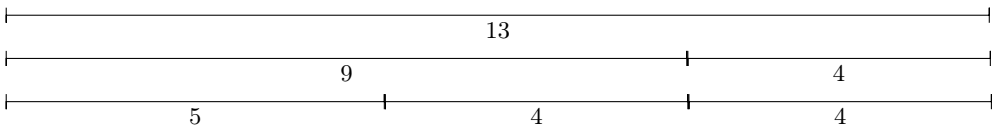
Deswegen wird nun versucht, die 7 rechts einzufügen. Auch dies führt allerdings zu keiner zulässigen Lösung, weil hierfür auch wieder eine Länge 2 benötigt wird.



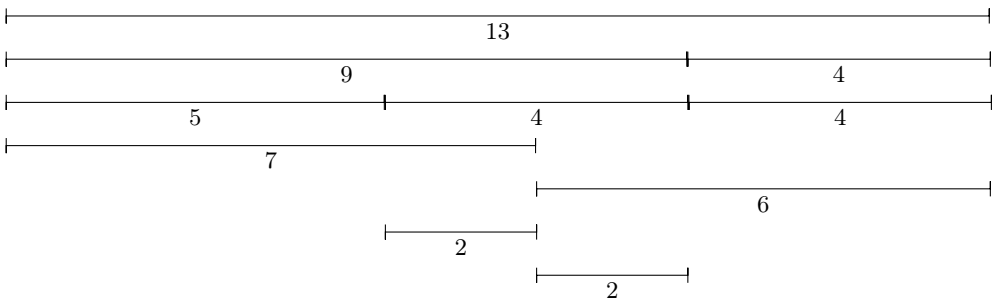
Dies bedeutet nichts anderes, als dass wir vorher einen Fehler gemacht haben müssen (wenn es eine zulässige Lösung gibt). Nun kommt es also zum eigentlichen Backtracking. Im Stack S stehen die eingefügten Längen und mit einem „Pop“ erhalten wir das letzte Element, das wir eingefügt haben. Dieses ist die 8, die wir wieder in A aufnehmen und wir stellen ausserdem die (Multi-) Mengen X und A wieder her, als wäre die 8 noch nicht betrachtet worden, also gilt $S = (9)$, $X = \{0, 9, 13\}$ und $A = \{1, 3, 4, 5, 6, 7, 8\}$. Der Algorithmus versucht nun, die 8 nicht links, sondern rechts einzufügen. Wir betrachten die Situation



und folgern, dass wir zusätzlich die Längen 4 und 5 benötigen, was möglich ist, und somit erhalten wir nun



und $S = (9, 8)$, $X = \{0, 8, 9, 13\}$ und $A = \{1, 3, 6, 7\}$. Wieder ist jetzt die 7 die grösste Länge in A und es wird probiert, sie links einzufügen, was erneut nicht funktioniert, da die 2 dann auch zweimal gebraucht werden müsste.



Folglich wird die 7 rechts eingefügt und wir sehen, dass dies möglich ist, denn zusätzlich werden genau die in A enthaltenen Längen 1, 3 und 6 benötigt.

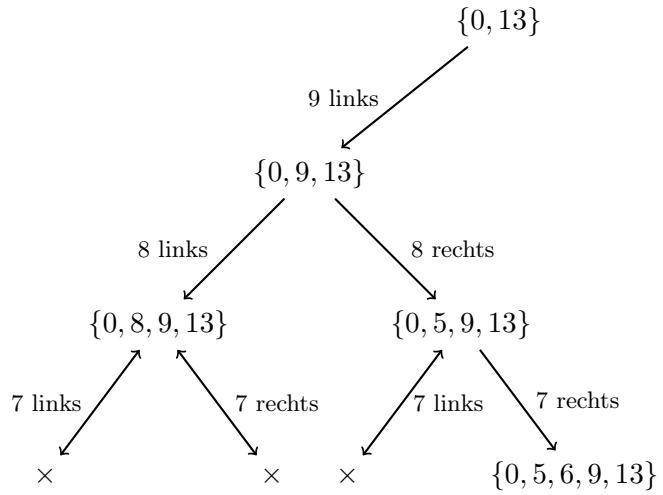
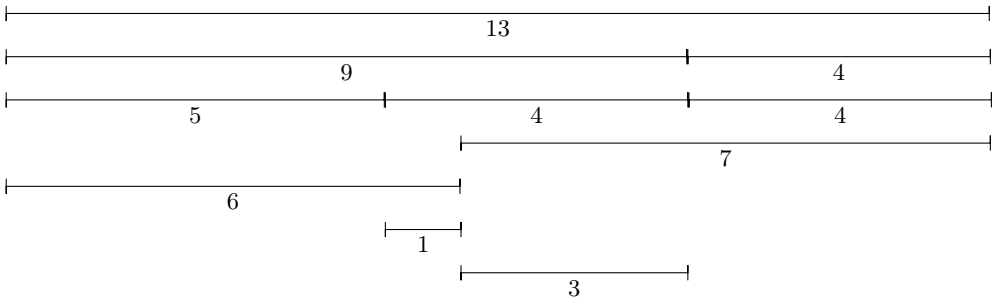
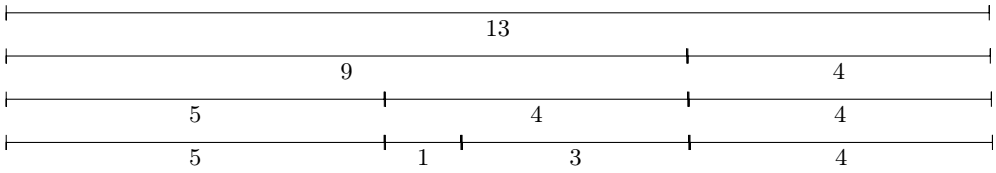


Abbildung 4. Der Baum, der den Schritten des Backtracking-Algorithmus entspricht. Ein Pfeil ein zwei Richtungen deutet hierbei an, dass zuerst nach unten und dann nach oben gegangen wird.



Mit dieser Anordnung entsteht schliesslich also die Abfolge von Lösungen



während der Berechnung des Algorithmus und somit ist jetzt $A = \emptyset$, $S = (9, 8, 7)$ und $X = \{0, 5, 6, 9, 13\}$, wobei X die Lösung der gegebenen Instanz A repräsentiert. Die verschiedenen Schritte des Algorithmus auf der Eingabe A sind in Abbildung 4 dargestellt.

Die Laufzeit des Backtracking-Algorithmus ist im schlimmsten Fall noch immer exponentiell, sie wächst nämlich mit $2^n \cdot n \cdot \log_2 n$ (in diesem Fall müsste der ganze Baum durchsucht werden), ist jedoch wesentlich kleiner als die des naiven Ansatzes. Ausserdem hat sich in der Praxis herausgestellt, dass der Algorithmus auf den meisten Eingaben mit wenigen Backtracking-Schritten auskommt, also recht effizient ist.

5 Strukturvorhersagen für Proteine

Proteine sind, wie wir schon am ersten Abend gesehen haben, lange kettenförmige Moleküle, zusammengesetzt aus 20 verschiedenen Bausteinen, den Aminosäuren. Proteine übernehmen in der Zelle viele verschiedene Aufgaben, zum Beispiel als Botenstoffe oder Enzyme, aber auch die Zellwände sind aus Proteinen aufgebaut. Weil die Aufgaben der Proteine so

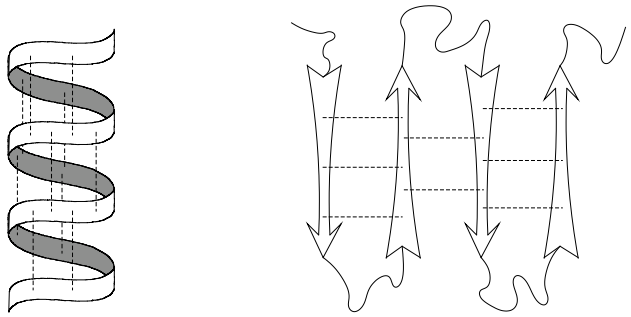


Abbildung 5. Schematische Darstellung der Sekundärstruktur von Proteinen: Links ist eine Helix dargestellt, rechts ein Faltblatt. Stabilisierende Bindungen zwischen den Elementen des Rückgrats sind jeweils durch gestrichelte Linien angedeutet.

vielfältig sind, haben sie (im Gegensatz zur DNA, die unabhängig von der Basenabfolge immer die gleiche Helixstruktur annimmt) sehr unterschiedliche dreidimensionale Strukturen. Ein Protein „faltet“ sich unmittelbar nach seiner Erzeugung im Ribosom in seine 3D-Struktur. Diese ist experimentell nur mit grossem Aufwand zu bestimmen, aber wichtig, um Rückschlüsse auf die Funktion des Proteins zu ziehen. Wir wissen, dass die Struktur vollständig von der Aminosäuren-Sequenz bestimmt wird, andere äussere Umstände spielen kaum eine Rolle. Deshalb ist es für uns von Interesse, Algorithmen zu finden, die aus der Sequenz die dreidimensionale Struktur vorhersagen können. Wir wollen im Folgenden einen ganz kurzen Überblick über die dafür verwendeten Verfahren geben.

Eine mögliche Vorgehensweise ist eine genaue Computersimulation des Faltungsprozesses auf atomarer Ebene. Man kann ein Modell entwickeln, das die Kräfte, die zwischen den einzelnen Atomen des Protein-Moleküls wirken, genau beschreibt. Mit Hilfe dieses Modells kann man dann im Prinzip den Faltungsprozess simulieren und so am Ende die 3D-Struktur des fertig gefalteten Proteins bestimmen. Leider sind solche exakten Modelle aber so rechenaufwändig, dass man es bis jetzt auch auf Hochleistungsrechnern nur schafft, winzige Sekundenbruchteile des Faltungsprozesses zu simulieren. Der gesamte Faltungsprozess nimmt aber etwa eine Minute Zeit in Anspruch.

Einen anderen Ansatz liefert das sogenannte *Protein-Threading*. Dies ist im Prinzip eine aufwändigere Form von Alignment-Verfahren, bei der man versucht, die Sequenz eines Proteins, dessen 3D-Struktur man bestimmen möchte, mit bereits bekannten dreidimensionalen Strukturen zu vergleichen. Hierfür verwendet man die sogenannten *Sekundärstrukturen* der Proteine. Als *Primärstruktur* bezeichnet man die Aminosäuren-Abfolge im Protein, die *Tertiärstruktur* bezeichnet die vollständige dreidimensionale Struktur des Proteins und die *Sekundärstruktur* beschreibt die Abfolge gewisser Strukturelemente. Es ist bekannt, dass sich grosse Teile der Proteinsequenz in eines von zwei Muster falten, die *Helix* und das *Faltblatt*. Diese sind schematisch in Abbildung 5 gezeigt und werden üblicherweise durch sogenannte *Schleifen* verbunden.

Wenn für ein Protein die Länge der Sekundärstrukturen und der dazwischen liegenden Schleifen bekannt ist, dann kann man versuchen, die Primärstruktur eines anderen Proteins dagegen auszurichten. Hierbei geht man davon aus, dass die Mismatches eher in den Schleifen auftreten als in den Helices oder Faltblättern, weil sonst die Struktur der Helix oder des Faltblatts nicht aufrecht erhalten werden könnte. Ein solches strukturelles Alignment (auch *Threading* genannt) ist in Abbildung 6 gezeigt.

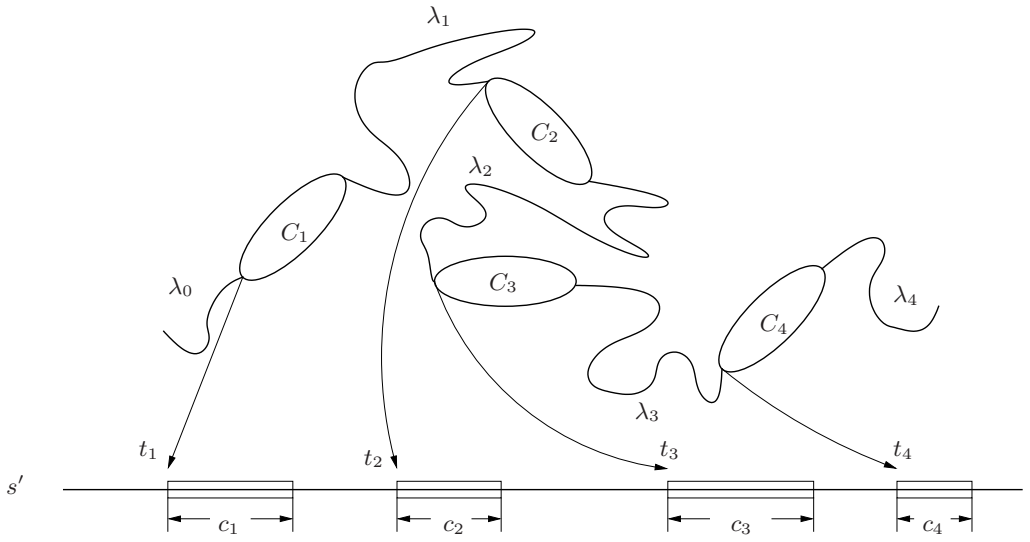


Abbildung 6. Schematische Darstellung des Protein-Threading. Hier bezeichnen $C_1, \ldots, C_4$ die Sekundärstrukturen, $c_1, \ldots, c_4$ die zugehörigen Sequenzen, $\lambda_0, \ldots, \lambda_4$ die Längen der verbindenden Schleifen und $t_1, \ldots, t_4$ die Startpositionen der Teilstrings $c_1, \ldots, c_4$ im zweiten Protein.

6 Gittermodelle

Wenn man aber kein ähnliches Protein mit bekannter 3D-Struktur findet, dann kann man auch versuchen, die Tertiärstruktur direkt aus der Primärstruktur zu berechnen. Um dieses Berechnungsproblem ein wenig zu vereinfachen, versucht man hierfür die einzelnen Aminosäuren des Proteins auf die Punkte eines diskreten Gitters einzubetten. Hierbei sollen zwei in der Sequenz aufeinanderfolgende Aminosäuren auf benachbarte Gitterpunkte abgebildet werden. Unter allen möglichen sochen Einbettungen versucht man dann eine solche zu finden, die die freie Bindungsenergie minimiert.

Diese freie Energie genau zu modellieren, ist wiederum sehr aufwändig, deshalb nutzt man hier vereinfachte Modelle. Eines der am meisten verbreiteten Modelle ist das sogenannte *HP-Modell*. Dabei teilt man die 20 Aminosäuren danach in zwei Klassen auf, ob sie polar (also geladen und damit wasserliebend) oder hydrophob (also wasserabstossend) sind. Die hydrophoben Aminosäuren haben die Tendenz sich zusammenzulagern, ausserdem finden sie sich eher im Innern der dreidimensionalen Struktur, die ja in der Regel in einer wässrigen Lösung zu finden ist. Das HP-Modell geht jetzt davon aus, dass die Anziehung zwischen den hydrophoben Aminosäuren die treibende Kraft hinter dem Faltungsprozess ist. Als Mass für die Güte einer Einbettung in das Gitter wird die Anzahl von benachbarten Paaren von hydrophoben Aminosäuren genommen. Manchmal vereinfacht man das Modell auch dadurch noch weiter, dass man eine Einbettung in ein zweidimensionales Gitter sucht. Dies kann für relativ kurze Proteinsequenzen sogar sinnvolle Vorhersagen liefern. Ausserdem erleichtert das zweidimensionale Gitter den Entwurf von Algorithmen, die eine gute Einbettung finden. Diese Algorithmen können dann oft auf den dreidimensionalen Fall erweitert werden.

Die optimale Einbettung eines Beispiel-Proteins in ein zwei- bzw. dreidimensionales quadratisches Gitter ist in den Abbildungen 7 und 8 gezeigt.

Leider ist es wiederum ein schwieriges kombinatorisches Problem, eine optimale Einbettung im HP-Modell zu berechnen. Deshalb kennt man keine exakten Algorithmen, sondern man versucht mit heuristischen Verfahren, eine gute Lösung zu finden. Eines dieser Verfahren, das man auch für zahlreiche andere kombinatorische Probleme verwenden kann,

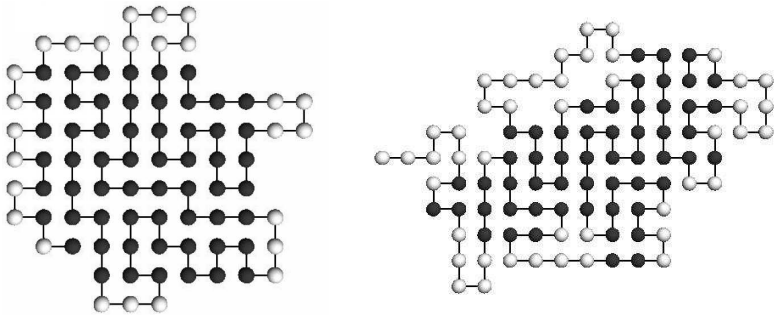


Abbildung 7. Die optimale Einbettung von zwei Proteinen im HP-Modell in ein zweidimensionales quadratisches Gitter. Die schwarzen Punkte entsprechen den hydrophoben Aminosäuren, die weissen Punkte den polaren Aminosäuren.

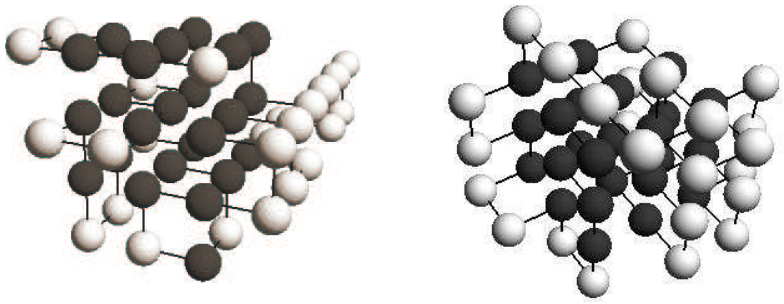


Abbildung 8. Die optimale Einbettung von zwei Proteinen im HP-Modell in ein dreidimensionales quadratisches Gitter. Die schwarzen Punkte entsprechen den hydrophoben Aminosäuren, die weissen Punkte den polaren Aminosäuren.

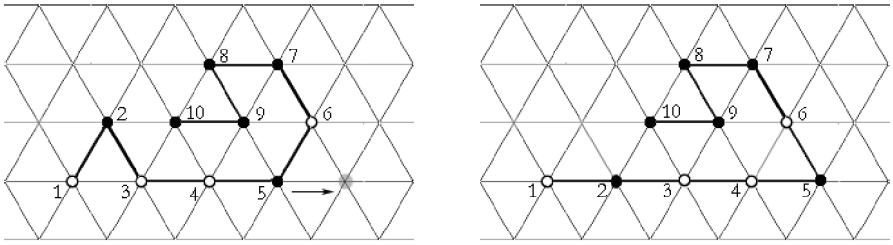


Abbildung 9. Ein Beispiel für eine lokale Änderung einer HP-Einbettung ins zweidimensionale Dreiecksgitter

ist die *lokale Suche*. Dabei startet man mit einer beliebigen Lösung, hier also zum Beispiel mit einer zufälligen Einbettung. Dann verändert man die Lösung lokal ein wenig, falls diese Veränderung zu einer besseren Lösung führt. Dies wiederholt man solange, bis keine lokale Verbesserung mehr möglich ist. Dieses Vorgehen führt natürlich in der Regel nicht zu einer Lösung, die auch *global* optimal ist. Deshalb wiederholt man das Verfahren in der Regel mit vielen zufälligen Startwerten. Ausserdem muss man sicherstellen, dass es mit den verwendeten lokalen Änderungen möglich ist, jede mögliche Lösung zu erreichen.

Ein Beispiel für eine solche lokale Änderung einer Einbettung in einem Dreiecksgitter, ein sogenannter *Pull-Move*, ist in Abbildung 9 gezeigt. Die Menge aller solcher Pull-Moves ist vollständig, man kann also von jeder Einbettung zu jeder anderen mit einer Folge von Pull-Moves kommen. Das Dreiecksgitter hat gegenüber dem quadratischen Gitter den Vorteil, das die Winkel eher den natürlichen Bindungswinkeln im Molekül entsprechen. Ein entsprechender Algorithmus lässt sich auch für eine dreidimensionale Erweiterung des Dreiecksgitters entwerfen.

7 Weiterführende Literatur

[1] Eine Einführung in die DNA-Sequenzierung und die physikalische Kartierung findet sich in den Kapiteln 6 bis 8 des Buchs

H.-J. Böckenhauer, D. Bongartz: *Algorithmische Grundlagen der Bioinformatik*, Teubner-Verlag 2003.

Das Buch enthält in Kapitel 12.3 auch eine Einführung in die Strukturvorhersagen für Proteine.

[2] Die Abbildungen 7 und 8 wurden dem folgenden Originalartikel entnommen:

K. Steinhöfel, A. Skaliotis, A. A. Albrecht: Stochastic protein folding simulation in the d-dimensional HP-model, *Proc. BIRD 2007*, Springer LNBI 4414, pp. 381–394.

[3] Abbildung 9 wurde dem folgenden Originalartikel entnommen:

H.-J. Böckenhauer, A. Z. Dayem Ullah, L. Kapsokalivas, K. Steinhöfel: A local move set for protein folding in triangular lattice models, *Proc. WABI 2008*, Springer LNCS 5251, pp. 369-381.

Alle anderen Abbildungen entstammen dem Buch *Algorithmische Grundlagen der Bioinformatik* [1].