Anwendungen dynamischer Programmierung in der Biologie

1 Einleitung RNS und DNS

RNS und DNS Moleküle bestehen aus vier Basen: Adenin, Guanin, Cytosin, Uracil (bei RNS und Thymin bei DNS). Diese Basen können sich zu Paaren zusammenschließen. Guanin mit Cytosin und Adenin mit Uracil bzw. Thymin.
Im Gegensatz zu DNS-Molekülen, bei denen sich zwei einzelne Stränge miteinander paaren, bilden RNS-Moleküle mit sich selbst eine sogenannte Sekundärstruktur. Diese Sekundärstruktur hält sich an einige Regeln.

- Sie ist immer ein Matching, d.h. jede Base paart sich höchstens mit einer anderen.
- Die Paare dürfen sich nicht kreuzen. Es kann also keine zwei Paare (i, j) und (k, l) geben mit $i < k < j < l$.
- Es gibt keine scharfen Knicke. Für jedes Paar (i, j) gilt $i < j - 4$.

In der Natur gibt es zwar auch Ausnahmen, aber diese Regeln bilden eine gute Heuristik, um die Sekundärstrukturen zu finden, die die Moleküle auch tatsächlich annehmen. Außerdem bilden die Moleküle ihre Sekundärstrukturen so, dass möglichst wenige Basen ungepaart bleiben.

2 Algorithmus zum Finden der wahrscheinlichsten Sekundärstruktur eines RNS Moleküls

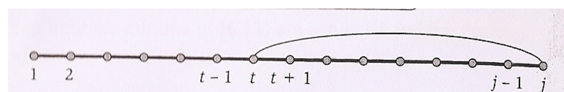
Zweck des Algorithmus ist es, zu einem RNS-Molekül die Sekundärstruktur zu finden, die es in der Natur annimmt. Betrachtet man das Molekül als Graph, wobei die Basen die Knoten sind, kann man ein maximales Matching suchen, dass den oben genannten Regeln folgt.

2.1 Idee

Sei $OPT(j)$ die maximale Anzahl von Basenpaaren in einer Sekundärstruktur von $b_1, .., b_j$. Gesucht ist dann $OPT(n)$. Aus Regel drei können wir schließen: $OPT(j) = 0$ für $j \leq 5$. Dann gibt es zwei Möglichkeiten für j:

a) j gehört nicht zu einem Paar
Dann brauchen wir nur die Lösung $OPT(j - 1)$ betrachten. Sie ist identisch mit $OPT(j)$.

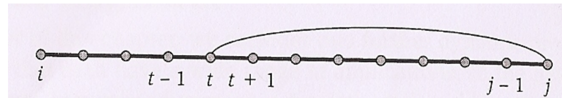
b) j ist gepaart mit einem $t < j - 4$
Auf Grund der Kreuzungsfreiheit haben wir unser Problem in zwei Teilprobleme zerlegt: Das erste kann formuliert werden als $OPT(t - 1)$. Das zweite lässt sich so nicht formulieren.



Es reicht also nicht Teilprobleme der Form $b_1, ..., b_j$ zu betrachen.Stattdessen müssen wir alle Teilprobleme $b_i, ..., b_j$ für $i \leq j$ berücksichtigen.Nennen wir die größte Anzahl an möglichen Basenpaaren für ein Teilproblem $b_i, ..., b_j$ $OPT(i, j)$ Nun können wir $OPT(i, j) = 0$ initialisieren für alle Paare (i, j) mit $i \geq j - 4$ Für $b_i, ..., b_j$ gibt es die gleichen Möglichkeiten wie zuvor.

a) Für den Fall, dass j nicht gepaart ist gilt wieder: $OPT(i, j) = OPT(i, j - 1)$

b) Für den Fall, dass j ein Paar mit einem t bildet, können wir jetzt beide Teilprobleme benennen: $OPT(i, t-1)$ und $OPT(t+1, j-1)$



2.2 Rekursion

$OPT(i,j) = max(OPT(i,j-1), max(1 + OPT(i,t-1) + OPT(t+1,j-1)))$

2.3 Laufzeit und Platzbedarf

Es sind $O(n^2)$ Teilprobleme zu lösen, da i und j jeweils von eins bis n laufen. Jedes benötigt $O(n)$ Zeit, da t von eins bis n läuft. Die Laufzeit beträgt insgesamt $O(n^3)$.
Da es $O(n^2)$ Teilprobleme sind, die berechnet werden und für jedes konstant viel Platz gebraucht wird, ist der Platzbedarf $O(n^2)$.
Die genauen Paare können durch backtracing gefunden werden.

3 Sequence Alignment

Für Sequence Alignment betrachten wir zwei Anwendungsfälle. Erstens die Suche in einem Wörterbuch nach einem Wort dessen Schreibung nicht exakt bekannt ist und zweitens den Vergleich zweier DNS-Stränge.

3.1 Abstandsmaß

Egal welche Anwendung wir betrachten, benötigen wir auf jeden Fall ein Maß um die Ähnlichkeit zweier Wörter bzw. die Güte eines Alignments zu messen. Das verwendete Maß setzt sich zusammen aus einer „gap penality“, die zum Tragen kommt wenn ein Buchstabe gegen eine Lücke ausgerichtet wird und einer „mismatch cost“ Funktion, die die Kosten für jedes Buchstabenpaar angibt, das gegeneinander ausgerichtet werden kann. Die Gesamtkosten für ein Alignment setzen sich aus der Summe der gap penalities für alle Lücken und den mismatch Kosten für alle falsch gegeneinander ausgerichteten Buchstaben zusammen.

3.2 Idee

Für die letzten beiden Stellen der Wörter gilt: entweder ist (m,n) im Matching oder (m,n) ist nicht im Matching Das reicht nicht um einen Ansatz zur dynamischen Programmierung zu formulieren.
Es gilt auch die schärfere Aussage: Ist (m,n) nicht im Matching, ist die m-te Stelle in X nicht gepaart oder die n-te Stelle in Y ist nicht gepaart.
Seien $OPT(i,j)$ die minimalen Kosten eines Alignments zwischen $x_1, ..., x_i$ und $y_1, ..., y_j$ Dann können wir drei Fälle unterscheiden:

- (i,j) ist im Matching
  Kosten $(a(i,j))$ für (i,j) werden bezahlt und $OPT(i-1, j-1)$ bestimmt
- i ist nicht gematcht
  Gap Kosten (d) werden bezahlt und $OPT(i-1, j)$ bestimmt

- j ist nicht gematcht
  Gap Kosten (d) werden bezahlt und $OPT(i, j-1)$ bestimmt

3.3 Rekursion

$OPT(i, j) = min(a(i, j) + OPT(i-1, j-1), d + OPT(i-1, j), d + OPT(i, j-1))$

3.4 Laufzeit und Platzbedarf

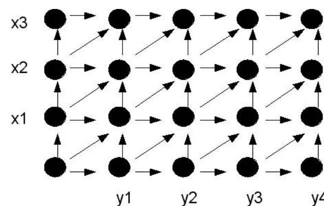
Es gibt $O(mn)$ Paare, deren Wert sich jeweils in konstanter Zeit berechnen lässt. Die Laufzeit beträgt also $O(mn)$ Ein $m \times n$ Feld reicht aus um alle nötigen Werte zu speichern. Der Platzbedarf beträgt also $O(m \times n)$.

4 Verbesserung von Sequence Alignment auf $O(n+m)$ Platzbedarf

Für den Anwendungsfall Wörterbuch sind diese Werte gut genug, wenn aber zwei DNS Stränge verglichen werden sollen, sprengt der Platzbedarf den Rahmen. Betrachtet man z.B. den Vergleich von 100.000 Symbolen, stellt die Ausführung von einer Million trivialen Optionen auf einem modernen Prozessor kein Problem mehr dar. Die Erstellung und Verwaltung eines 10 Gigabyte großen Feldes dagegen schon. Deshalb ist es wünschenswert den Platzbedarf des Algorithmus zu verringern.

4.1 Darstellung als Graphenproblem

Um unser Problem etwas anschaulicher darzustellen betrachten wir die Überführung in ein Graphenproblem. Betrachtet man einen Graph, wie in der Abbildung, an den horizontal und vertikal die beiden Wörter geschrieben sind und belegt die horizontalen und vertikalen Kanten mit der gap penality und die diagonalen jeweils mit den entsprechenden mismatch Kosten, reicht es aus einen kürzesten Weg von $(0, 0)$ nach (n, m) zu finden.



In dieser Darstellung kann man gut erkennen, dass die Werte spaltenweise berechnet werden können und immer nur die Werte aus der direkt vorhergegangenen Spalte betrachtet werden müssen. Wir können also die Werte aus den Spalten davor überschreiben und brauchen nur noch ein $m \times 2$ Feld. Damit können wir den gesuchten Wert in $O(m)$ Platz berechnen. Wir haben aber keine Informationen mehr wie der ideale Weg verläuft. Für den DNS Vergleich bedeutet das, dass wir zwar wissen wie ähnlich sich die beiden Stränge sind, aber keine Informationen darüber haben, welche Teilstücke identisch sind und bei welchen größere Unterschiede vorliegen. Diese Informationen werden aber benötigt.

4.2 „Rückwärtsformulierung“

Um auch diese Informationen ermitteln zu können, greift man zu einem Trick. Als erstes formulieren wir unser Problem noch einmal, aber aus der anderen Richtung. Bezeichnen wir unser bisheriges

$OPT(i,j)$ jetzt als $f(i,j)$ und betrachten zusätzlich $g(i,j)$ mit:
$g(i,j) = min(a(i+1,j+1) + g(i+1,j+1), d + g(i,j+1), d + g(i+1,j))$
g sucht den kürzesten Weg nicht wie f von $(0,0)$ nach (n,m), sondern umgekehrt von (n,m) nach $(0,0)$. Es ist offensichtlich, dass das Ergebnis das gleiche ist. Es gibt aber noch weitere nützliche Zusammenhänge zwischen g und f:

Der kürzeste Pfad durch (i,j) hat die Länge $f(i,j) + g(i,j)$

Diese Aussage ist leicht ersichtlich, da $f(i,j)$ der kürzeste Weg von $(0,0)$ nach (i,j) ist und $g(i,j)$ der kürzeste Pfad von (i,j) nach (n,m).

Sei k eine beliebige Zahl zwischen 0 und n und q ein Index, der den Ausdruck $f(q,k) + g(q,k)$ minimiert. Dann gibt es einen Pfad von $(0,0)$ nach (n,m) der minimale Länge hat und durch (i,j) läuft.

Jeder Weg von $(0,0)$ nach (n,m) muss die Spalte k auf jeden Fall an mindestens einer Stelle treffen. Wenn der Weg minimal ist, dann sollte er k so schneiden, dass Summe der beiden Teilstücke bis k und von k minimal ist. Und das ist sie genau bei (q,k) mit $f(q,k) + g(q,k)$ minimal.

4.3 Divide and conquer

Der Graph wird an der mittleren Spalte geteilt und $f(i,n/2)$ und $g(i,n/2)$ für jedes i berechnet. Der Knoten $(i,n/2)$ mit dem optimalen Wert wird gespeichert. Rekursiv werden beide Teile des Graphen wieder halbiert.

4.4 Laufzeit und Platzbedarf

Jeder Aufruf braucht nur $O(m)$ Platz. Der Weg muss gespeichert werden, hat aber maximal $O(m+n)$ Länge. Insgesamt beträgt der Platzbedarf also $O(m+n)$. Aber bleibt die Laufzeit $O(mn)$?

4.5 Beweis

$T(m,n) \leq cmn + T(q,n/2) + T(m-q,n/2)$
$T(m,2) \leq cm$
$T(2,n) \leq cn$
Wir machen die Annahme, dass $T(m,n) \leq kmn$ für ein k.
Der Anker stimmt für $k \geq c/2$.
Nehmen wir nun an $T(m',n') \leq km'n'$ für kleineres Produkt, dann gilt:
$T(m,n) \leq cmn + T(q,n/2) + T(m-q,n/2)$
$\leq cmn + kqn/2 + k(m-q)n/2$
$= cmn + kqn/2 + kmn/2 - kqn/2$
$= (c + k/2)mn$
Ist erfüllt für $k = 2c$.

Literatur

[1] J. Kleinberg, E. Tardos, *Algorithm Design*, Pearson Education 2006

[2] http://de.wikipedia.org/wiki/Desoxyribonukleins%C3%A4ure

[3] http://de.wikipedia.org/wiki/Ribonukleins%C3%A4ure