

31. Algorithmus der Woche

Dynamische Programmierung: Evolutionäre Distanz

Autor

Norbert Blum, Uni Bonn
Matthias Kretschmer, Uni Bonn

Da vor 150 Jahren nahe bei Düsseldorf im Neandertal erstmals Skeletteile des so genannten Neandertalers gefunden wurden, ist 2006 das Jahr des Neandertalers. Seit diesem Fund stellte sich u.a. die Frage, inwiefern der Homo sapiens, von dem wir abstammen, und der Neandertaler miteinander verwandt sind. Lange hielt man es für möglich, dass wir vom Neandertaler abstammen. Mittlerweile haben Wissenschaftler herausgefunden, dass die Entwicklungslinie des Homo neanderthalensis sich vor ca. 315.000 Jahren von jener trennte, die schließlich zum Homo sapiens führte. Dies hat man aufgrund der Unterschiede im Erbgut des Homo sapiens und des Homo neanderthalensis festgestellt. Neue Verfahren ermöglichen signifikante Anteile des Erbguts aus den mehr als 30.000 Jahre alten Knochen zu extrahieren. Dieses erhält man in Form von DNA-Sequenzen. DNA-Sequenzen kann man sich als Baupläne für Tiere und Menschen vorstellen. Während der evolutionären Geschichte haben sich DNA-Sequenzen aufgrund von Mutationen geändert. Hat man DNA-Sequenzen verschiedener Spezies, dann kann man die Ähnlichkeit dieser Sequenzen mit Hilfe des Computers schätzen. Wir werden die Ähnlichkeit von zwei DNA-Sequenzen über die Distanz dieser DNA-Sequenzen ausdrücken. Umso ähnlicher zwei DNA-Sequenzen sind, umso geringer ist die Distanz zwischen diesen.

Wie kann man einen Algorithmus entwickeln, der die Distanz zweier DNA-Sequenzen schätzt?

Mit dieser Frage wollen wir uns zunächst beschäftigen. Hierzu benötigen wir als erstes ein mathematisches Modell. D.h., wir müssen DNA-Sequenzen und Mutationen modellieren, um die Distanz zweier Sequenzen formal definieren zu können.

Eine DNA-Sequenz ist eine Zeichenkette bestehend aus den Zeichen A, G, C und T . Formal sagt man, eine DNA-Sequenz ist ein String x über dem Alphabet $\Sigma = \{AGCT\}$. So ist zum Beispiel

CAGCGGAAGGTCACGGCCGGGCCTAGCGCCTCAGGGGTG

ein Ausschnitt der DNA-Sequenz des Huhnes. Eine *Mutation* führt eine DNA-Sequenz x in eine DNA-Sequenz y über. Wir gehen davon aus, dass es nur drei verschiedene Mutationen (die wir auch Operationen nennen werden) gibt:

1. Streichen eines Zeichens,
2. Einfügen eines Zeichens und
3. Ersetzen eines Zeichens durch ein anderes.

Zum Beispiel wenn $x = AGCT$, dann würde die Mutation Ersetzen von G durch C die DNA-Sequenz x nach der DNA-Sequenz $y = ACCT$ transformieren. Die Position an der ein Zeichen gestrichen, eingefügt oder ersetzt wird, ist, wie wir später sehen werden, aus dem Kontext direkt ersichtlich. Deswegen werden wir die Position nicht explizit angeben. Wir verwenden $a \rightarrow b$, um das Ersetzen des Zeichens a durch das Zeichen b darzustellen. $a \rightarrow$ bezeichnet das Streichen des Zeichens a und $\rightarrow b$ das Einfügen des Zeichens b .

Um ein Maß für die Distanz zweier DNA-Sequenzen zu erhalten ordnen wir jeder Mutation Kosten zu. Diese Kosten haben etwas mit der Wahrscheinlichkeit, dass eine korrespondierende Mutation auftritt, zu tun. Je wahrscheinlicher die Mutation umso geringer die Kosten. Wir ordnen den drei verschiedenen Mutationen folgende Kosten zu:

- Streichen: Kosten 2

- Einfügen: Kosten 2
- Ersetzen: Kosten 3

Wenn wir zwei verschiedene DNA-Sequenzen x und y miteinander vergleichen, dann benötigen wir in der Regel mehrere Mutationen, um x in y zu überführen. Wähle zum Beispiel $x = AG$ und $y = T$. Wir sehen, dass eine einzelne Mutation nicht x nach y transformieren kann. Wir sehen aber auch, dass die Mutationenfolge $S = A \rightarrow, G \rightarrow T$ (Streichen von A und Ersetzen von G durch T), x nach y transformiert. Wir definieren die Kosten einer Mutationenfolge $S = s_1, \dots, s_t$, als die Summe der Kosten der Mutationen s_1 bis s_t . D.h., formal sind die Kosten c für eine Mutationenfolge S :

$$c(S) := \sum_{i=1}^t c(s_i) = c(s_1) + \dots + c(s_t)$$

Wir wollen die Kosten einer Mutationenfolge verwenden, um die Distanz zweier DNA-Sequenzen zu definieren. Dabei müssen wir beachten, dass es mehrere verschiedene Mutationenfolgen geben kann, die eine DNA-Sequenzen x nach einer DNA-Sequenz y transformieren. Falls zum Beispiel $x = AG$ und $y = T$, wie oben ist, dann gibt es unter anderem folgende Möglichkeiten:

- $S_1 = A \rightarrow, G \rightarrow T$; $c(S_1) = c(A \rightarrow) + c(G \rightarrow T) = 2 + 3 = 5$
- $S_2 = A \rightarrow T, G \rightarrow$; $c(S_2) = c(A \rightarrow T) + c(G \rightarrow) = 3 + 2 = 5$
- $S_3 = A \rightarrow, G \rightarrow, \rightarrow T$; $c(S_3) = c(A \rightarrow) + c(G \rightarrow) + c(\rightarrow T) = 2 + 2 + 2 = 6$
- $S_4 = A \rightarrow C, G \rightarrow, C \rightarrow, \rightarrow T$; $c(S_4) = c(A \rightarrow C) + c(G \rightarrow) + c(C \rightarrow) + c(\rightarrow T) = 3 + 2 + 2 + 2 = 9$

Es gibt beliebig viele weitere Mutationenfolgen, die allerdings alle keine niedrigeren Kosten haben als S_1 und S_2 . Wir verwenden zur Bestimmung der Distanz einfach die Mutationenfolge mit den geringsten Kosten. Dabei soll die Distanz umso größer sein, umso größer die Kosten sind. D.h. wir definieren die Distanz $d_c(x, y)$:

$$d_c(x, y) := \min\{c(S) \mid S \text{ transformiert } x \text{ nach } y\}$$

In unserem Beispiel von oben, ist die Distanz $d_c(x, y) = 5$, da S_1 und S_2 die Mutationenfolgen mit den geringsten Kosten sind, die x nach y transformieren.

Wie berechnet man nun die evolutionäre Distanz? D.h., wie berechnen wir $d_c(x, y)$? Wir haben nur die Operationen Streichen, Einfügen und Ersetzen zur Verfügung, um x nach y zu transformieren. Die Operationen und das Kostenmodell sind so definiert, dass eine Folge von mehreren Operationen an derselben Position kostengünstiger durch eine einzelne Operation ersetzt werden kann. So kann zum Beispiel des Streichen eines Zeichens a und das Einfügen eines Zeichens b an derselben Position durch das Ersetzen von a durch b kostengünstiger durchgeführt werden (Kosten $2 + 2 = 4$ für eine Streiche- und eine Einfüge-Operation gegenüber Kosten 3 für eine Ersetze-Operation). Die Mutationen an den verschiedenen Positionen beeinflussen sich gegenseitig nicht. Wir können diese Mutationen somit in beliebiger Reihenfolge durchführen. Wir gehen nun davon aus, dass die letzte Mutation immer an den letzten Positionen der beiden Sequenzen stattfindet.

Sei $x = a_1 a_2 \dots a_m$ und $y = b_1 b_2 \dots b_n$. D.h. x besteht aus m und y aus n Zeichen. Gemäß unseren drei Operationen gibt es drei Möglichkeiten, x nach y zu transformieren:

1. *Streichen*: Wir streichen a_m und $a_1 a_2 \dots a_{m-1}$ wird nach $b_1 b_2 \dots b_n$ transformiert.
2. *Einfügen*: Wir fügen b_n ein und $a_1 a_2 \dots a_m$ wird nach $b_1 b_2 \dots b_{n-1}$ transformiert.
3. *Ersetzen*: Wir ersetzen a_m durch b_n und $a_1 a_2 \dots a_{m-1}$ wird nach $b_1 b_2 \dots b_{n-1}$ transformiert.

Für die Berechnung der evolutionären Distanz nehmen wir die kostengünstigste der drei Varianten.

Wir werden jetzt obige Beobachtung zu einem Algorithmus ausarbeiten. Sei nun $x[1 : i]$ der Präfix von x , der aus den ersten i Zeichen besteht. Der Präfix der Länge 0 von x ist der leere String $x[1 : 0]$. Der Präfix der Länge m von x ist x selber, da x gerade m Zeichen lang ist. Entsprechend ist $y[1 : j]$ der Präfix von y , der aus den ersten j Zeichen besteht. Wir können nun die obige Beobachtung wie folgt umformulieren:

1. $x[1 : m - 1]$ wird nach y transformiert und wir streichen a_m .
2. x wird nach $y[1 : n - 1]$ transformiert und wir fügen b_n ein.
3. $x[1 : m - 1]$ wird nach $y[1 : n - 1]$ transformiert und wir ersetzen a_m durch b_n .

Jetzt stellt sich die Frage: Wie transformieren wir $x[1 : m - 1]$ nach y , x nach $y[1 : n - 1]$ und $x[1 : m - 1]$ nach $y[1 : m - 1]$? Hierzu können wir dasselbe Schema anwenden. Zur Bestimmung der evolutionären Distanz $d_c(x[1 : i], y[1 : j])$ von $x[1 : i]$ und $y[1 : j]$ benötigen wir somit die evolutionären Distanzen $d_c(x[1 : i - 1], y[1 : j])$, $d_c(x[1 : i], y[1 : j - 1])$ und $d_c(x[1 : i - 1], y[1 : j - 1])$. Wir erhalten dann $d_c(x[1 : i], y[1 : j])$ durch

$$d_c(x[1 : i], y[1 : j]) := \min \begin{cases} d_c(x[1 : i - 1], y[1 : j]) + c(a_i \rightarrow), & \text{(Streichen)} \\ d_c(x[1 : i], y[1 : j - 1]) + c(\rightarrow b_j), & \text{(Einfügen)} \\ d_c(x[1 : i - 1], y[1 : j - 1]) + c(a_i \rightarrow b_j) & \text{(Ersetzen)} \end{cases}$$

Wir haben somit das Problem der Berechnung von $d_c(x[1 : i], y[1 : j])$ auf drei Teilprobleme zurückgeführt.

Beachte, dass für $i = 0$ und $j > 0$ nur eine Einfüge-Operation (d.h. Fall 2) und für $j = 0$ und $i > 0$ nur eine Streiche-Operation (d.h. Fall 1) in Frage kommen. Das rührt daher, dass wir in einem leeren String kein Zeichen streichen und auch keines ersetzen können. Um den leeren String zu erhalten, können wir in einem nichtleeren String nur Zeichen streichen. Im Fall $i = 0$ und $j = 0$ müssen wir den leeren String $x[1 : 0]$ nach den leeren String $y[1 : 0]$ transformieren. Dafür benötigen wir keine einzige Operation, da beide Strings identisch sind. Somit ergibt sich $d_c(x[1 : 0], y[1 : 0]) = 0$.

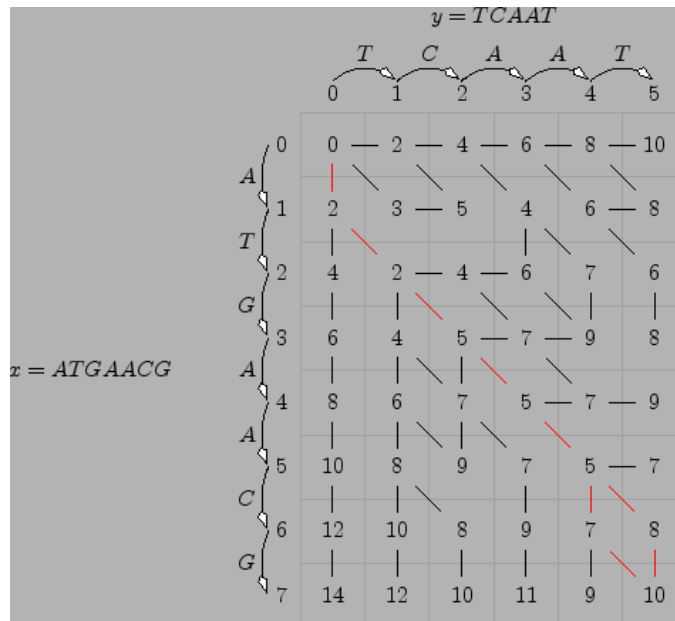
Wähle zum Beispiel $x = AGT$ und $y = CAT$. Nehmen wir an, wir hätten schon $d_c(x[1 : 1], y[1 : 2]) = d_c(A, CA)$, $d_c(x[1 : 2], y[1 : 1]) = d_c(AG, C)$ und $d_c(x[1 : 1], y[1 : 1]) = d_c(A, C)$ bestimmt. Dann können wir daraus die evolutionäre Distanz von $d_c(x[1 : 2], y[1 : 2]) = d_c(AG, CA)$ nach obigem Muster bestimmen. Dazu bestimmen wir das Minimum von

- $d_c(x[1 : 1], y[1 : 2]) + c(a_2 \rightarrow) = d_c(A, CA) + c(G \rightarrow) = d_c(A, CA) + 2$
- $d_c(x[1 : 2], y[1 : 1]) + c(\rightarrow b_2) = d_c(AG, C) + c(\rightarrow A) = d_c(AG, C) + 2$ und
- $d_c(x[1 : 1], y[1 : 1]) + c(a_2 \rightarrow b_2) = d_c(A, C) + c(G \rightarrow A) = d_c(A, C) + 3$

Wenn wir $d_c(A, CA)$, $d_c(AG, C)$ und $d_c(A, C)$ vorher bestimmt haben, können wir somit $d_c(AG, CA)$ bestimmen.

Wie können wir nun diese Überlegungen verwenden, um einen Algorithmus zu formulieren? Die evolutionären Distanzen von Präfixen von x und y werden mehrfach benötigt. So wird zum Beispiel $d_c(x[1 : i - 1], y[1 : j - 1])$ zur Berechnung von $d_c(x[1 : i], y[1 : j - 1])$, $d_c(x[1 : i - 1], y[1 : j])$ und $d_c(x[1 : i], y[1 : j])$ benötigt. Wir möchten die Berechnung von $d_c(x[1 : i - 1], y[1 : j - 1])$ nur einmal durchführen. Demzufolge müssen wir diesen Wert speichern, so dass wir ihn mehrfach verwenden können. Wir verwenden hierzu eine Tabelle, in der wir die schon berechneten evolutionären Distanzen zwischenspeichern. In dieser Tabelle verwenden wir die Zelle (i, j) (Zeile i und Spalte j) zur Speicherung der evolutionären Distanz $d_c(x[1 : i], y[1 : j])$. Der Vorteil dieser Methode ist, dass es wesentlich effizienter ist, die Werte direkt aus der Tabelle zu lesen, als sie jedesmal erneut zu berechnen. Wir benötigen die evolutionären Distanzen für alle $0 \leq i \leq m$ und $0 \leq j \leq n$. Also besteht unsere Tabelle aus $m + 1$ Zeilen und aus $n + 1$ Spalten.

Sei zum Beispiel $x = ATGAACG$ und $y = TCAAT$. Die dazugehörige Tabelle wäre, bei der obigen Kostenfunktion (Streichen und Einfügen kosten 2, Ersetzen kostet 3):



Wir fangen bei der Berechnung der Distanzen mit $d_c(x[1:0], y[1:0])$ an. Wir wissen schon, dass $d_c(x[1:0], y[1:0]) = 0$ ist. Also notieren wir uns in der Zelle $(0,0)$ den Wert 0. Wenn wir die Spalte 0 von oben nach unten durchgehen, können wir nur Streiche-Operation durchführen. Das haben wir oben bereits bemerkt (j ist in diesem Fall gleich 0). Man kann nur Zeichen streichen, da wir $x[1:i]$ nach den leeren String $y[1:0]$ transformieren wollen. Entsprechend können wir in der Zeile 0 nur Einfüge-Operationen durchführen.

Die anderen Tabelleneinträge berechnen sich nun nach der Formel von oben. Betrachten wir zum Beispiel die Zelle $(2,1)$. Wir wollen den String $x[1:2] = AT$ nach den String $y[1:1] = T$ transformieren. Wir sehen direkt, dass es mit einer Streiche-Operation erledigt ist. Wir müssen nur A entfernen und danach T durch T ersetzen. Das ist auch intuitiv die Lösung mit den minimalen Kosten. Der Algorithmus muss demnach auch diese Distanz (eine Streiche-Operation: Kosten = 2) berechnen. Dies ist auch der Fall, denn er bildet das Minimum aus den drei Werten, die den drei Operationen entsprechen:

1. $d_c(x[1:1], y[1:1]) + c(a_2 \rightarrow) = d_c(A, T) + c(a_2 \rightarrow) = 3 + 2 = 5$ (Streichen von T),
2. $d_c(x[1:2], y[1:0]) + c(\rightarrow b_1) = d_c(AT, y[1:0]) + c(\rightarrow b_1) = 4 + 2 = 6$ (Einfügen von T) und
3. $d_c(x[1:1], y[1:0]) + c(a_2 \rightarrow b_1) = d_c(A, y[1:0]) + c(a_2 \rightarrow b_1) = 2 + 0 = 2$ (Ersetzen von T durch T).

Die Operation Ersetzen von T durch T , ist keine Mutation und wird nur der Einfachheit halber verwendet. In Wirklichkeit ersetzen wir T nicht durch T sondern tun nichts. Daher betragen die Kosten hierfür 0. Die Operation Streiche A ist hier nicht explizit aufgeführt. Dies liegt daran, dass diese Operation implizit bei dem Schritt $x[1:1] = A$ nach $y[1:0]$ zu transformieren enthalten ist, der durch den Tabelleneintrag $(1,0)$ repräsentiert wird und vorher schon berechnet wurde.

Die kleinen Striche innerhalb der Tabelle sollen verdeutlichen über welche „Wege“ die optimale Transformation von $x[1:i]$ nach $y[1:j]$ ablaufen kann. In dem Fall der Zelle $(2,1)$ sind wir gerade diagonal von $(1,0)$ über eine Ersetzen-Operation zu den minimalen Kosten gekommen. Diese Wege sind nicht notwendigerweise eindeutig, wie man anhand der Tabelle sehen kann. Die Striche können direkt bei dem Aufbau der Tabelle bestimmt werden, da sie nur angeben, welcher der drei möglichen Schritte an den letzten Positionen der Präfixe, zu minimalen Kosten führt.

Die rot markierten Wege, geben die „Wege minimaler Kosten“ für die Transformation von x nach y an. D.h. sie geben die Wege an, über die man mit minimalen Kosten x nach y transformieren und somit die evolutionäre Distanz bestimmen kann.

Da wir nicht wissen, welche Tabelleneinträge zu einem „Weg minimaler Kosten“ für die Transformation von x nach y beitragen, müssen wir alle bestimmen. Wir benötigen zur Berechnung des Eintrags von Zelle (i, j) die Werte der Zellen $(i-1, j)$, $(i, j-1)$ und $(i-1, j-1)$. Um sicherzustellen, dass diese evolutionären Distanzen bereits berechnet sind, wenn wir $d_c(x[1:i], y[1:j])$ berechnen wollen, bauen wir die Tabelle zeilenweise von oben nach unten oder spaltenweise von links nach rechts auf. Wenn wir die Tabelle zeilenweise aufbauen, müssen wir jede Zeile von links nach rechts durchlaufen. Entsprechend muss jede Spalte beim spaltenweisen Aufbau von oben nach unten durchlaufen werden. Am Ende steht dann in Zelle (m, n) das gewünschte Ergebnis: die evolutionäre Distanz $d_c(x[1:m], y[1:n])$ von $x[1:m] = x$ und $y[1:n] = y$.

Rekapitulieren wir: Wir haben, beginnend bei dem einfachsten und kleinsten Teilproblem, die Berechnung von $d_c(x[1:0], y[1:0])$, immer größer werdende Teilprobleme gelöst. Wir haben die Präfixe von x und y immer größer werden lassen und für diese die evolutionäre Distanz berechnet. Dabei haben wir zur Bestimmung von $d_c(x[1:i], y[1:j])$ die drei evolutionären Distanzen $d_c(x[1:i-1], y[1:j])$, $d_c(x[1:i], y[1:j-1])$ und $d_c(x[1:i-1], y[1:j-1])$ benötigt. Abstrakt ausgedrückt: Wir haben optimale Teillösungen verwendet, um ein größeres Teilproblem optimal zu lösen.

Die Frage ist, ob das ein allgemeines Prinzip ist, welches ein allgemeines Programmierverfahren zulässt? Dies ist der Fall. Das Prinzip heißt *Optimalitätsprinzip* und sieht folgendermaßen aus:

- Jede Teillösung einer optimalen Lösung, die Lösung eines Teilproblems ist, ist selbst eine optimale Lösung des betreffenden Teilproblems.

Das Programmierverfahren zur Lösung eines Optimierungsproblems, das auf dem *Optimalitätsprinzip* aufbaut, heißt *dynamische Programmierung*. Bei der dynamischen Programmierung teilen wir unser Problem in Teilprobleme auf. Die kleinsten Teilprobleme lösen wir direkt. In unserem Fall war dies gerade die Berechnung von $d_c(x[1:0], y[1:0])$. Die Lösung hierfür war sehr einfach, da immer $d_c(x[1:0], y[1:0]) = 0$ gilt. Aus den optimalen Lösungen zu kleinen Teilproblemen werden dann optimale Lösungen für größere Teilprobleme berechnet. Dies wird solange wiederholt, bis man eine optimale Lösung für das eigentliche Problem berechnet hat. Die dynamische Programmierung ist ein wichtiges allgemeines Verfahren, das beim Algorithmenentwurf häufig seine Anwendung findet.

Das hier vorgestellte Verfahren kann nicht nur zur Berechnung der evolutionären Distanz zweier DNA-Sequenzen verwendet werden, sondern auch um die Ähnlichkeit zweier Wörter der deutschen Sprache zu bestimmen. Dies wird unter anderem bei Rechtschreibkorrekturprogrammen angewendet. Zum Beispiel wird dem Benutzer jedes Wort, das eine Distanz kleiner oder gleich 5 zu einem dem Rechtschreibkorrekturprogramm unbekanntem Wort hat, als Alternative für dieses unbekanntem Wort angeboten.

Wie man heute weiß, stammt der Homo sapiens nicht vom Neandertaler ab. Es wurden derartige Unterschiede im Erbgut des Neandertalers und des Homo sapiens festgestellt, dass daraus geschlossen werden konnte, dass der Homo sapiens kein Nachfahre des Neandertalers ist. Dies hat man mit Computerverfahren, ähnlich dem oben vorgestellten, nachweisen können.

Autoren:

- Prof. Dr. Norbert Blum
<http://theory.cs.uni-bonn.de/blum/blum.var>
- Dipl.-Inform. Matthias Kretschmer
<http://theory.cs.uni-bonn.de/blum/Mitarbeiter/kretschm/welcome.var>

Weiterführende Materialien:

- Java-Applet zur Visualisierung des Verfahrens
(erstellt von Ryan McKinley und von Sascha Meinert und Martin Nöllenburg modifiziert)
<http://www-11.informatik.rwth-aachen.de/~algorithmus/Algorithmen/algo31/applet/evodis.php>

Externe Links:

- Wikipedia über Dynamische Programmierung
http://de.wikipedia.org/wiki/Dynamische_Programmierung