

Vorlesung

Algorithmen in der Bioinformatik

Wintersemester 2002/03

Clemens Gröpl

4. Februar 2003

1 Einleitung

Worum soll es in dieser Vorlesung gehen, was ist das eigentlich: Bioinformatik? – Eine subjektive Sicht.

1.1 Daten: Womit haben wir es zu tun?

- *Strings* – Sequenzdaten, beschreiben Nukleinsäuren (DNA) und Aminosäuren (Proteine).
- *Punktkoordinaten* – Atome, 3D-Struktur von Proteinen.
- *Ähnlichkeitsmaße* – z.B. aus paarweisen Vergleichen.

1.2 Aufgaben: Was ist zu tun?

- *Auffinden von Ähnlichkeiten* zwischen Teilstrukturen und Finden einer Zuordnung.
 - *Auswahl der Teilstrukturen*: z.B. Substring oder Atommenge.
 - *Explizite Zuordnung*: z.B. String-Alignment, 3D-Überlagerung.
- *Strukturierung der Ähnlichkeitsbeziehungen*.
 - *Assembly*: z.B. ein komplettes Genom aus sequenzierbaren Teilstücken zusammenbauen.
 - *Clustering*: finde Gruppen von Dingen, die (1.) untereinander sehr ähnlich, aber (2.) unähnlich zu den übrigen sind.
 - *Phylogenetik*: finde einen (Stamm-) Baum, der die Ähnlichkeiten widerspiegelt.
 - *Globale und lokale Eigenschaften des Ähnlichkeitsgraphen*.

1.3 Methoden: Wie tun wir es?

- *String-Algorithmen und -Datenstrukturen*.
 - dynamische Programmierung.
 - Suffixbäume.
- *Singularwertzerlegung*. Die Singulärwerte einer Matrix kann man als verallgemeinerte Eigenwerte auffassen. Aus den zugehörigen Eigenvektoren kann man Cluster ableiten. Anwendungen: 3D-Überlagerung, Clustering (Google).
- *Graphentheorie* – taucht an vielen Stellen auf, z.B. Eulertour (sequence shuffling), Zusammenhangszahl (Clustering).
- *Den Zufall verstehen*.
 - Abgrenzung *signifikanter* Ergebnisse von Zufallstreffern.
 - Die *wahrscheinlichste Erklärung* finden: z.B. Hidden Markov Models, Maximum Likelihood Trees.

2 Strings

Bezeichnungen Sei Σ ein endliches Alphabet mit $|\Sigma| \geq 2$. (Z.B. die Menge der Nuklein- oder Aminosäuren.) Mit Σ^n bezeichnen wir die Menge der Wörter der Länge n über dem Alphabet Σ . Wie üblich sei $\Sigma^* := \bigcup_{n=0}^{\infty} \Sigma^n$, wobei $\Sigma^0 = \{\epsilon\}$, ϵ das leere Wort. Für $S \in \Sigma^n$ sei $|S| = n$ die Länge von S . Die Begriffe Wort, String und Sequenz sind werden in diesem Kontext praktisch synonym gebraucht, aber zwischen Substring und Subsequenz (auch Teil- statt Sub-) ist gut zu unterscheiden.

Definition 1. Seien $S, T \in \Sigma^*$ Strings.

- $S[i]$ = i -tes Zeichen von S .
(Statt $S[i]$ sieht man oft auch $S(i)$ oder S_i .)

- $S[i..j] = S[i]S[i + 1] \dots S[j]$.
($S[i..j] = \epsilon$ für $i > j$.)
- S heißt *Substring* von T , falls es i und j gibt, so dass $S = T[i..j]$.
- S heißt *Subsequenz* von T , falls es Positionen $i_1 < \dots < i_{|S|}$ gibt, so dass $S = T[i_1]T[i_2] \dots T[i_{|S|}]$.
- $S[1..i]$ ist das *Präfix* von S bis zum i -ten Zeichen.
- $S[i..|S|]$ ist das *Suffix* von S ab dem i -ten Zeichen.

2.1 String Matching

Gegeben: Pattern $P \in \Sigma^m$ und Text $T \in \Sigma^n$.

Gesucht: Kleinstes i mit $P = T[i .. i + |P| - 1]$, oder $i = 0$, falls P kein Substring von T ist.

Wir können $m < n$ voraussetzen. (In der Regel ist $m \ll n$.) Durch Suche im verbleibenden Text findet man alle Vorkommnisse von P in T .

2.1.1 Brute Force

Die erste Lösung, die einem zum String Matching Problem einfällt, ist wohl der

Algorithmus Brute Force String Matching

For $i = 1, \dots, n - m + 1$: $O(n)$
 Vergleiche, ob $P = T[i .. i + m - 1]$. $O(m)$

Im Beispiel $P = a^{m-1}b$, $T = a^{n-1}b$ werden alle mn Paare von Positionen miteinander verglichen, die Laufzeit ist also im schlechtesten Fall nur $O(mn)$. Tatsächlich ist das String Matching Problem in $O(m + n)$ lösbar, also um einen Faktor m schneller.

Erwartete Laufzeit Andererseits sieht man leicht, dass die erwartete Laufzeit des Brute Force Algorithmus, wenn $P \in \Sigma^m$ und $T \in \Sigma^n$ gleichverteilt zufällig gewählt sind, ebenfalls $O(m + n)$ ist: Sei $s := |\Sigma| \geq 2$. Dann ist

$$\Pr(P[j] = T[i + j - 1]) = 1/s.$$

Dann ist für jedes i ist die erwartete Anzahl von Vergleichen, bis feststeht, dass $P \neq T[i .. i + m - 1]$, beschränkt durch

$$\sum_{j=1}^m 1/s^{j-1} = \frac{1 - 1/s^m}{1 - 1/s} \leq 2.$$

Die erwartete Anzahl von Vergleichen in jedem Durchlauf der for-Schleife ist also konstant ($O(1)$ statt wie im schlechtesten Fall $O(m)$). Aus der Linearität des Erwartungswertes folgt, dass die erwartete Gesamtlaufzeit $O(n) = O(m + n)$ ist.

Rivest (1977) hat gezeigt, dass jeder String Matching Algorithmus im schlechtesten Fall mindestens $n - m + 1$ viele Zeichen von T lesen muss, bis das Ergebnis feststeht.

```

i = 1 // läuft über T
j = 1 // läuft über P
while (i ≤ n) and (j ≤ m)
  if (P[j] = T[i])
    i = i + 1 // weiter in T
    j = j + 1 // weiter in P
  else
    i = i - j + 2 // nächster Versuch in T
    j = 1 // zurück zum Anfang von P
if (j > m)
  i = i - m // Anfang des gefundenen Matches
else
  i = 0 // P ist kein Substring von T
    
```

Abbildung 1: Brute Force String Matching

2.1.2 Knuth-Morris-Pratt

Ein einfaches Beispiel zeigt, wie der Brute Force Algorithmus (Abb. 1) verbessert werden kann.

	i									
T:	a	b	c	d	a	b	c	f	g	b
	1	2	3	4	5	6	7	8		
P:	a	b	c	d	a	b	c	e		
	j									
next _j	0	1	1	1	1	2	3	4		
next' _j	0	1	1	1	0	1	1	4		

Abbildung 2: Beispiel

Der Brute Force Algorithmus würde in der Situation in Abb. 2 als nächstes $j = 2$ und $i = 1$

setzen. Aufgrund der vorangegangenen Vergleiche wissen wir aber, dass $P[1..7] = T[1..7]$ ist. Da $P[1..3] = P[5..7]$ gilt, können wir das Pattern gleich um 4 Positionen nach rechts verschieben.

Idee des Algorithmus von KMP In der Zeile $next_j$ ist angegeben, welche Position des Patterns als nächstes mit $T[i]$ zu vergleichen ist, wenn der Vergleich an Stelle j des Pattern einen Unterschied liefert. Der Wert $next_j = 0$ bedeutet, dass i um 1 erhöht wird. Konkret ist

$$next_j = \max\{ k \mid P[1 .. k-1] = P[j-k+1 .. j-1] \}.$$

Der Hauptteil des KMP-Algorithmus ist in Abb. 3 angegeben.

```

i = 1 // läuft über T
j = 1 // läuft über P
while (i ≤ n) and (j ≤ m)
  while (j > 0) and (P[j] ≠ T[i])
    j = next'_j // Pattern P verschieben
    i = i + 1 // weiter in T
    j = j + 1 // weiter in P
  if (j > m)
    i = i - m // Anfang des gefundenen Matches
  else
    i = 0 // P ist kein Substring von T
    
```

Abbildung 3: KMP-Algorithmus, Hauptteil

Das „and“ in der inneren while-Schleife ist ein „conditional and“, d.h. die rechte Seite der Bedingung wird nur ausgewertet, falls die linke zutrifft.

Im Beispiel aus Abb. 2 wird in der Situation $i = j = 8$ zunächst $j = next_8 = 4$ gesetzt. Da $T[8] = f \neq d = P[4]$, wird als nächstes $j = next_4 = 1$ gesetzt und wegen $T[8] = f \neq a = P[1]$ schließlich $j = 0$. Im nächsten Schleifendurchlauf geht es dann weiter mit $T[9]$ und $P[1]$.

Falls $P[next_j] = P[j]$, so steht der Ausgang des nächsten Vergleiches bereits im voraus fest. Mit der folgenden Definition werden solche unnötigen Vergleiche geschickt vorweggenommen.

$$next'_j = \max\{ k \mid P[1 .. k-1] = P[j-k+1 .. j-1] \wedge P[j] \neq P[k] \} \cup \{0\}.$$

In Worten: $next'_j - 1$ ist die Länge des längsten Präfixes von P , das zugleich ein Suffix von $P[1 .. j-1]$ ist, und zudem $P[next'_j] \neq P[j]$ erfüllt.

Im Beispiel aus Abb. 2 ergibt sich derselbe Ablauf. Im Beispiel aus Abb. 4 wird dagegen der Vergleich

	i	
T:	a b c d a b a b c	
P:	a b c d a b c e	
		(*)
	a b c d a b c e	

Abbildung 4: Beispiel zu $next'_j$ vs. $next_j$.

in Zeile (*) eingespart. Gemäß Spalte 7 in Abb. 2 ist nämlich $next_7 = 3$, aber $next'_7 = 1$.

Wie man in einem Preprocessing die $next'_j$ -Werte in $O(m)$ berechnet, sehen wir gleich.

Ende VL 2002-10-21

Korrektheit des KMP-Hauptalgorithmus Wir betrachten gleich die Variante mit den $next'_j$ -Werten. (Wegen $next'_j \leq next_j$ folgt daraus auch die Korrektheit für $next_j$.)

Wie weit darf der KMP-Algorithmus das Pattern verschieben, wenn in einem Durchlauf der äußeren while-Schleife bei $P[j] \neq T[i]$ erstmals ein mismatch auftrat? Offenbar mindestens um

$$s = \min\{ t \mid \underbrace{P[1 .. j-t]}_{=P[1 .. j-t-1]P[j-t]} = \underbrace{T[i-j+1+t .. i]}_{=P[t+1 .. j-1]T[i]} \}.$$

Mit der Substitution $j-t \leftrightarrow k$ wird daraus

$$\begin{aligned}
 s &= \min\{ j-k \mid P[1 .. k-1] = P[j-k+1 .. j-1] \\
 &\quad \wedge \underbrace{P[k] = T[i]}_{\Rightarrow P[k] \neq P[j]} \} \\
 &\geq \min\{ j-k \mid P[1 .. k-1] = P[j-k+1 .. j-1] \\
 &\quad \wedge P[k] \neq P[j] \}.
 \end{aligned}$$

Wir dürfen also um s schieben, aber der KMP Algorithmus schiebt tatsächlich nur um $j - next'_j \leq s$.

Laufzeit des KMP-Hauptalgorithmus Das Pattern wird in der Zeile $j = next_j$ nach rechts verschoben (relativ zum Text), nachdem ein Mismatch gefunden wurde. Die Zeile $j = next_j$ wird jedoch höchstens so oft ausgeführt wie die Zeile $i = i + 1$, da stets $j < next_j$ gilt und j nur zusammen mit i inkrementiert wird. Da i nur n mal inkrementiert werden kann, ist die gesamte Laufzeit (nach dem Preprocessing) also $O(n)$.

Preprocessing Der Schlüssel zur Berechnung von next_j ist die Beobachtung, dass am Ende jedes Durchlaufes der äußeren while-Schleife im Hauptteil gilt $P[1 .. j - 1] = T[i - j + 1 .. i - 1]$. Die next'_j -Werte können daher analog zum Hauptteil berechnet werden, indem man das Pattern mit sich selbst vergleicht (Abb. 5).

```

i = 1 // läuft über P (statt T)
j = 0 // läuft über P
next'_1 = 0 // spezieller Wert
while (i ≤ m)
  while (j > 0) and (P[j] ≠ P[i])
    j = next'_j
  i = i + 1
  j = j + 1
  // einfacher wäre: next_i = j
  // besser ist:
  if (P[j] = P[i])
    next'_i = next'_j
  else
    next'_i = j
    
```

Abbildung 5: KMP-Algorithmus, Preprocessing

Korrektheit und Laufzeit des KMP-Preprocessings Die Korrektheit ergibt sich durch Induktion über $i = 1, \dots, m$, da man zum Berechnen von next_i nur die next_j -Werte für $j < i$ verwendet. Der Wert $\text{next}'_1 = 0$ ist nach Definition korrekt. Wir schließen nun von i auf $i + 1$. Aus der Korrektheit des Hauptalgorithmus folgt, dass am Ende jedes Durchlaufes der äußeren while-Schleife gilt: $P[1 .. j - 1]$ ist das längste Präfix von P , das zugleich ein Suffix $P[i - j + 1 .. i - 1]$ von $P[1 .. i - 1]$ ist. Daraus folgt sofort die Korrektheit für den Fall $P[j] \neq P[i]$. Im Fall $P[j] = P[i]$ ist $P[1 .. j] = P[i - j + 1 .. i]$. Daher ist

$$\begin{aligned} \text{next}'_i &= \max \{ k \mid P[1 .. k-1] = \underbrace{P[i-k+1 .. i-1]}_{=P[j-k+1 .. j-1]} \\ &\quad \wedge \underbrace{P[i]}_{=P[j]} \neq P[k] \} \cup \{0\} \\ &= \text{next}'_j. \end{aligned}$$

Die Laufzeit $O(m)$ für das Preprocessing ergibt sich völlig analog zum Hauptteil, da nun P an die Stelle von T tritt.

2.1.3 BLAST

Das Akronym BLAST steht für basic local alignment search tool. Blast wurde von Altschul, Gish, Mil-

ler, Myers und Lipman 1990 vorgestellt und hat weit verbreitete Anwendung gefunden. Um zu beschreiben, was Blast tut, schicken wir ein paar Definitionen vorweg.

Definition 2. Seien $S, T \in \Sigma^*$ Strings.

- Ein Segmentpaar besteht aus Substrings $S' \subseteq S, T' \subseteq T$ mit $|S'| = |T'|$.
- Der Wert eines Segmentpaares ist

$$d(S', T') := \sum_{i=1}^{|S'|} \delta(S'[i], T'[i]),$$

wobei $\delta \in \mathbb{Z}^{\Sigma \times \Sigma}$ ein Ähnlichkeitsmaß ist. (Z.B. eine Aminosäuren-Substitutionsmatrix wie PAM250 oder BLOSUM62.)

- Ein Segmentpaar heißt *lokal maximal*, falls sein Wert nicht vergrößert werden kann, indem man links oder rechts Zeichen hinzu- oder wegnimmt.
- Ein Segmentpaar heißt *maximal*, falls sein Wert maximal ist. Abkürzung: MSP (für maximal segment pair).

Funktionalität

Gegeben: Anfragesequenz S , Sequenzdatenbank $\{T_1, \dots\}$, kritischer Wert C .

Gesucht: Alle Sequenzen der Datenbank, die zusammen mit S ein MSP vom Wert $\geq C$ enthalten.

Blast ist ein heuristisches Programm, es ist nicht garantiert, dass alle Treffer gefunden werden. Andererseits ist die mathematische Problemformulierung eine Abstraktion dessen, wonach man in der Praxis sucht.

Methode

- Konstruiere die Menge $\{P_1, \dots, P_N\}$ aller Strings, die zu einem Substring von S der Länge w einen Ähnlichkeitswert $\geq t$ haben. (w und t sind heuristische Parameter.)
- Finde alle (exakten) Vorkommnisse von $\{P_1, \dots, P_N\}$ in der Datenbank mittels (einer Variante) des Algorithmus von Aho und Corasick.
- Versuche, die gefundenen Segmentpaare zu einem lokal maximalen Segmentpaar mit Wert $\geq C$ zu erweitern.

Laufzeit (im wesentlichen ...)

- Linear in N .
- Linear in $\sum |T_i|$.
- Die Suche nach lokal maximalen Segmentpaaren wird heuristisch abgebrochen. (Exakt in $O(|T_i|)$ möglich.¹)

2.1.4 Aho-Corasick: Multiple String Matching

Wir wenden uns nun dem zweiten Teilproblem (aus der Skizze) des Blast-Algorithmus zu.

Gegeben: Menge von Patterns $\{P_1, \dots, P_N\} \subseteq \Sigma^*$ und Text $T \in \Sigma^n$.

Gesucht: Aufzählung der Paare (i, p) , so dass an Position i des Textes das Pattern P_p beginnt, d.h., $P_p = T[i .. i + |P_p| - 1]$.

Das Multiple String Matching Problem kann offenbar durch N -maligen Aufruf des Algorithmus von Knuth-Morris-Pratt in $O(m + Nn)$ gelöst werden, wobei $m := \sum_{p=1}^N |P_p|$ die Gesamtlänge aller Patterns ist. Dabei startet man KMP auf dem verbleibenden Text jedesmal neu, um *alle* Treffer zu finden.

Der Algorithmus von Aho-Corasick löst das Multiple String Matching Problem in $O(m + n + k)$, wobei k die Anzahl der Treffer ist. Er ist eine Verallgemeinerung des Algorithmus von Knuth-Morris-Pratt und macht wie dieser keine Rückwärtsschritte im Text. Die Patterns werden in einem keyword tree zusammengefasst. Die next-Werte werden durch „failure links“ ersetzt.

Ende VL 2002-10-28

Definition 3. Ein *keyword tree* für eine Menge von Patterns $\{P_1, \dots, P_N\} \subseteq \Sigma^*$ ist ein Baum $K = (V, E)$ mit einer Wurzel $root(K)$, in dem die Kanten von der Wurzel weggerichtet und mit Buchstaben gelabelt sind. Alle ausgehenden Kanten eines Knotens haben verschiedene Labels. Der String aus den Buchstaben auf den Kanten auf dem Pfad von der Wurzel zu einem Knoten v sei mit $L(v)$ bezeichnet. ($L(root) = \epsilon$.) Falls $L(v) = P_p$, so trägt v einen entsprechenden Vermerk $pat(v) = p$, ansonsten $pat(v) = 0$. Alle $L(v)$ sind Präfixe von Patterns und jedes Pattern P_p kommt als $P_p = L(v)$ für ein $v \in V$ vor.

¹Übungsaufgabe!

Aufgrund dieser Definition überzeugt man sich leicht, dass die Abbildung $v \mapsto L(v)$ injektiv ist.

Ein Beispiel für einen keyword tree ist in Abb. 6 gegeben.

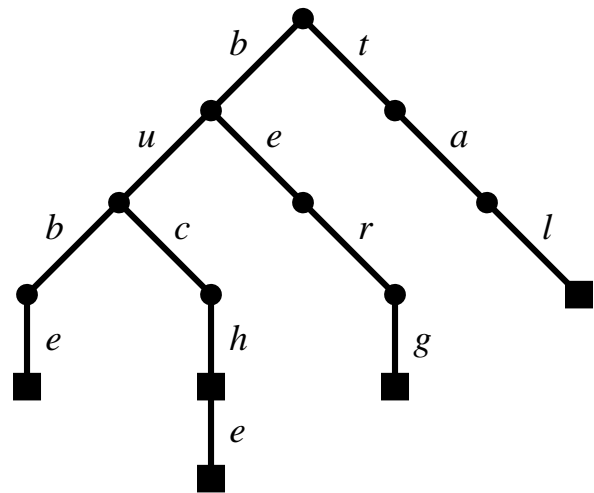


Abbildung 6: Keyword tree für {berg, bube, buch, buche, tal}

Zur Beschreibung des Algorithmus von Aho und Corasick (Abb. 7) brauchen wir einige Definitionen.

Definition 4. Sei $K = (V, E)$ keyword tree für $\{P_1, \dots, P_N\}$ und $v \in V(K)$.

- Die *Übergangsfunktion* $succ : V \times \Sigma \rightarrow V \cup \{fail\}$ ist definiert durch

$$succ(v, a) := \begin{cases} w \in V, & \text{mit } L(v)a = L(w), \text{ oder} \\ fail, & \text{falls kein solches } w \text{ existiert.} \end{cases}$$

- Die *failure links* $flink : V \rightarrow V$ sind definiert durch $flink(v) := w$, wobei $L(w)$ das längste Präfix eines P_p ist, das zugleich ein echtes Suffix von $L(v)$ ist. (Spezialfall: $flink(root) = root - \epsilon$ hat kein echtes Suffix.)
- Zusätzlich hat jeder Knoten v noch einen *output link* $olink(v) \in V \cup \{fail\}$, der (falls vorhanden) auf den nächsten Knoten w mit $pat(w) \neq 0$ zeigt, der von v aus über failure links erreichbar ist.

Korrektheit des AC-Hauptalgorithmus Analog zum KMP-Algorithmus zeigt man, dass am Ende jedes Durchlaufes der äußeren while-Schleife gilt: $L(v)$ ist das längste Präfix eines Patterns, das zugleich ein Suffix von $T[1 .. i - 1]$ ist. [...]

```

i = 1 // läuft über T
v = root // L(v) läuft über Präfixe von Patterns
while (i ≤ n)
  while (v ≠ root) and (succ(v, T[i]) = fail)
    v = flink(v) // zurückspringen in K
  if (succ(v, T[i]) ≠ fail)
    w = v = succ(v, T[i]) // weiter in K
  if (pat(v) ≠ fail) // Matches ausgeben
    print (i - |Ppat(v)|, pat(v))
  while (olink(w) ≠ fail)
    print (i - |Ppat(w)|, pat(w))
    w = olink(w)
i = i + 1 // weiter in T
    
```

Abbildung 7: AC-Algorithmus, Hauptteil

Laufzeit des AC-Hauptalgorithmus Die Laufzeit $O(n)$ ergibt sich sehr ähnlich wie im KMP-Algorithmus. Anstelle von „next_j < j“ tritt „|L(flink(v))| < |L(v)|“.

AC-Preprocessing Der keyword tree für die Patterns $\{P_1, \dots, P_N\}$ kann offenbar in $O(m)$, wobei $m = \sum_{p=1}^N |P_p|$, aufgebaut werden. Für eine konstante Alphabetgröße ist dabei in jedem Knoten v die Initialisierung von $\text{succ}(v, \cdot)$ in $O(1)$ möglich.

Die Berechnung von $\text{flink}(\cdot)$ erfolgt ähnlich wie bei KMP durch Breitensuche ausgehend von der Wurzel (Abb. 8). Die output links kann man dabei gleich mit erstellen.

Korrektheit des AC-Preprocessings Wir führen Induktion über die Breitensuche. $\text{flink}(\text{root}) = \text{root}$ ist korrekt nach Definition. Sei $v \in V$ und $\text{flink}(v)$ bereits für alle w mit $|L(w)| < |L(v)|$ korrekt berechnet. Sei $L(v) = L(v')a$. Offenbar ist $L(\text{flink}(v))$ ein Suffix von $L(\text{flink}(v'))a$, denn ein längeres Suffix von $L(v)$ als $L(\text{flink}(v'))a$ kann es nicht als Präfix eines Patterns geben, da sonst bereits $\text{flink}(v')$ falsch gewesen wäre. Sei $w = \text{flink}(v')$.

Falls $\text{succ}(w, a) \neq \text{fail}$, so ist also $\text{flink}(v) = \text{succ}(w, a)$ korrekt.

Falls $\text{succ}(w, a) = \text{fail}$, so ist $L(w)a$ kein Präfix eines Patterns. Also muss $L(\text{flink}(v))$ ein echtes Suffix von $L(w)a$ sein. Also ist $L(\text{flink}(v))$ ein Suffix von $L(\text{flink}(w))a$.

Damit tritt w an die Stelle von v' , und wir suchen das längste Suffix von $L(\text{flink}(w))a$, das zugleich ein Präfix eines Patterns ist. Die innere while-Schleife wird verlassen, wenn $w = \varepsilon$ oder

wenn $L(w)a$ Präfix eines Patterns ist. In beiden Fällen wird anschließend $\text{flink}(v)$ entsprechend zugewiesen.

```

flink(root) = root // per def. ok.
olink(root) = fail
for all v ∈ V in BFS order (origin=root)
  let L(v) =: L(v')a // v' früher besucht, a ∈ Σ
  w = flink(v')
  while (w ≠ root) and (succ(w, a) = fail)
    w = flink(w)
  if (succ(w, a) = fail) // flink zuweisen
    flink(v) = root
  else
    flink(v) = succ(w, a)
  if (pat(flink(v)) ≠ fail) // olink zuweisen
    olink(v) = flink(v)
  else
    olink(v) = olink(flink(v))
    
```

Abbildung 8: AC-Algorithmus, Preprocessing

Ende VL 2002-11-04

2.2 Sequenzen mischen

Wie kann man feststellen, ob ein Alignment signifikant ist? Genauer: Was ist die bei der Suche in einer großen Datenbank aufgrund des Zufalls zu erwartende Anzahl von Treffern? Eine exakte mathematische Modellierung des Begriffs eines zufälligen Strings ist schwierig, da biologische Sequenzen viele Eigenschaften haben, die sie von „einfachen“ Zufallsstrings unterscheiden. Insbesondere sind benachbarte Positionen nicht stochastisch unabhängig. Für DNA-Sequenzen ($\Sigma = \{A, C, G, T\}$) ist z.B. durchweg $\text{TA}\downarrow$ (\downarrow =unter-, \uparrow =überrepräsentiert), in Vertebraten ist $\text{CG}\downarrow$, $\text{TG}\uparrow$, $\text{CA}\uparrow$, in Eukaryoten ist $\text{CCA}\uparrow$, $\text{TGG}\uparrow$, in Eukaryoten und Bakterien ist $\text{CTAG}\downarrow$. Wir werden nun zwei Ansätze kennenlernen, wie man zufällige Strings erzeugen kann, die dieselben k -let Häufigkeiten haben wie die Originaldaten. Unter einem k -let versteht man dabei einfach einen Substring der Länge k . Bei der hier betrachteten Anwendung kann man k als konstant annehmen. Aus dem Vergleich der Resultate für Originalsequenzen mit denen für zufällig gemischte Sequenzen kann man auf deren Signifikanz schließen.

Sequenzen mischen

Gegeben: String $S \in \Sigma^n, k \in \mathbb{N}$.

Gesucht: Zufälliger String T , der gleichverteilt ist unter allen Strings in

$$\Omega(S, k) := \{ T \mid \#_a(T) = \#_a(S) \text{ für alle } a \in \Sigma^k \},$$

wobei $\#_a(T) := \#\{ i \mid T[i .. i + |a| - 1] = a \}$ angibt, wie oft a in T vorkommt.

Proposition 5. Für $T \in \Omega(S, k)$ gilt

$$|T| = \sum_{a \in \Sigma^k} \#_a(T) + k - 1 = \sum_{a \in \Sigma^k} \#_a(S) + k - 1 = |S|.$$

2.2.1 Vertauschungsalgorithmus zum Mischen von Sequenzen

Der Vertauschungsalgorithmus (Abb. 9) strebt eine Gleichverteilung an, indem er ausgehend von $T = S$ wiederholt Substrings von T , deren Beginn und Ende an mindestens $k - 1$ Positionen übereinstimmen, miteinander vertauscht (s. Abb. 10). Bei einer solchen Vertauschung bleiben die k -let Häufigkeiten erhalten.

```

T = S
repeat
    • Wähle  $1 \leq a < b < c < d \leq n - k + 2$ 
      gleichverteilt zufällig.
    • if  $((T[a .. a + k - 2] = T[c .. c + k - 2])$ 
       $\wedge (T[b .. b + k - 2] = T[d .. d + k - 2]))$ 
      then
        swap( $T[a .. b + k - 2], T[c .. d + k - 2]$ )
until ( nobody knows )
output T
    
```

Abbildung 9: Vertauschungsalgorithmus zum Mischen von Sequenzen

```

TTACTACTGAATTCTAAGTTAAAT
TTACTAAGTTAATTCTACTGAAAT
    
```

Abbildung 10: Beispiel zum Vertauschungsalgorithmus, $k = 3$

Falls $k > 2$, so können sich die Substrings $T[a .. b + k - 2]$ und $T[c .. d + k - 2]$ überlappen.

```

TTACTGCTGACTGACTTAGCATACTCAT
TTACTGACTTAGCATACTGCTGACTCAT
    
```

Abbildung 11: Vertauschung bei überlappenden Substrings

Die Vertauschung kann dennoch durchgeführt werden (Abb. 11).

Die vier Positionen a, b, c, d können mit geeigneten Datenstrukturen gleichverteilt zufällig unter allen solchen ausgewählt werden, für die die nachfolgende if-Abfrage zutrifft.

Etwas komplizierter wird es im Fall, wo $S[1 .. k - 1] = S[n - k + 2 .. n]$. Solche Sequenzen heißen *zyklisch*. Sei z.B. $S = ACGTAC$ und $k = 3$ und $T = GTACGT$. Dann ist keine Vertauschung möglich, obwohl die Triplet-Häufigkeiten übereinstimmen. Abhilfe schafft in dieser Situation eine „random rotation“: Wähle $m \in [1 .. n - k + 1]$ zufällig und initialisiere $T = (SS[k .. n])[m .. m + n - 1]$. Man kann sich das so vorstellen, dass S im Kreis geschrieben wird und T mit Verschiebung m ausgelesen wird.

Allerdings ändert eine random rotation regelmäßig die $(k - 1)$ -let Häufigkeiten, denn z.B. der Substring der Länge $k - 1$, der zuvor am Anfang und am Ende stand, kommt nun einmal weniger oft vor. Man muss sich bei zyklischen Sequenzen also entscheiden, ob man genau die k -let Häufigkeiten bewahren will – dann ist random rotation erforderlich, um ganz Ω zu erzeugen – oder ob man in der Definition von Ω die Häufigkeiten von ℓ -lets für $\ell \leq k$ bewahren will – dann bleiben die ersten und letzten $k - 1$ Zeichen gleich.

Man kann beweisen, dass für genügend große Laufzeiten die Verteilung des ausgegebenen Strings T gegen die Gleichverteilung auf Ω konvergiert. Für die benötigte Laufzeit, bis man hinreichend „nah“ (in einem bestimmten Distanzmaß für Wahrscheinlichkeitsverteilungen) an der Gleichverteilung ist, sind jedoch leider keine brauchbaren oberen Schranken bekannt, obwohl Experimente vermuten lassen, dass der Algorithmus „schnell“ mischt. (Stichwort: rapidly mixing Markov chain.)

2.2.2 Exaktes Mischen von Sequenzen mittels Eulertouren

Die Idee des Algorithmus, den wir hier beschreiben, kann man sich als ein Dominospiel vorstellen, bei

dem die Steine mit Strings der Länge $k - 1$ beschriftet sind und zwei Steine aneinandergelegt werden können, wenn sie in $k - 2$ Zeichen übereinstimmen.

Definition 6. Sei $S \in \Sigma^*$ und $k \in \mathbb{N}$. Der k -let Graph $D_k(S)$ von S hat die Knotenmenge

$$V = \{ a \in \Sigma^{k-1} \mid a \text{ ist Substring von } S \}.$$

Für jeden Substring $S[i .. i + k - 1]$ der Länge k enthält E eine Kante von $S[i .. i + k - 2]$ nach $S[i + 1 .. i + k - 1]$. (Schleifen sind möglich.)

Reduktionsschritt 1: Von Zufallsstrings zu Eulertouren

Wenn man alle Dominosteine aneinanderlegen kann, heißt das in der Graphentheorie eine Eulertour.

Definition 7. Sei $G = (V, E)$ ein gerichteter Graph. Eine *Eulertour* ist eine Folge von Kanten $u_1v_1, \dots, u_mv_m \in E$, in der jede Kante genau einmal vorkommt und an jeder Stelle $v_i = u_{i+1}$ gilt. Falls $v_m = u_1$, so heißt die Eulertour *geschlossen*, und man nennt den Graphen *eulersch*. Der Graph kann mehrfache parallele und antiparallele Kanten sowie Schleifen haben.

Den Beweis des folgenden Satzes ist eine nette Übungsaufgabe. (Ein gerichteter Graph heißt *stark zusammenhängend*, falls es zu je zwei Knoten x und y einen gerichteten Pfad von x nach y gibt.)

Satz 8 (GA1²). Sei $G = (V, E)$ ein gerichteter Graph und $u, v \in V$.

1. Eine geschlossene Eulertour existiert genau dann, wenn G stark zusammenhängend ist und $\forall w \in V : \text{deg}^+(w) = \text{deg}^-(w)$.
2. Eine Eulertour von u nach v existiert genau dann, wenn der Graph $(V, E \cup \{vu\})$ eine geschlossene Eulertour hat, wobei vu eine zusätzliche Kante von v nach u ist.

Bei geschlossenen Eulertouren ($\Leftrightarrow S$ zyklisch) betrachten wir dennoch einen „Anfang=Ende“-Knoten als mitinbegriffen. („Rotierte“ Eulertouren sind also verschieden.) Für konstantes k hat der k -let Graph nur konstant viele Knoten. Man wird daher die Adjazenzen am einfachsten in Form einer $(|\Sigma|^{k-1} \times |\Sigma|)$ -Matrix mit Einträgen in \mathbb{N} verwalten, die die Vielfachheit einer Kante angeben. Wir bezeichnen die Vielfachheit einer Kante e (d.h., die Anzahl zu e paralleler Kanten, wobei e mitgezählt wird) mit $f(e)$ (für „frequency“).

²Das Skript zur Vorlesung „Graphen und Algorithmen 1“ ist online verfügbar.

Lemma 9. Es gilt

$$\#\Omega(S) = \prod_{e \in E} f(e)! \cdot \#\{ \text{Eulertouren} \},$$

und eine entsprechende Bijektion ist effizient berechenbar.

Beweis. Von einer Eulertour in D_k liest man sofort den String $T \in \Omega$ auf den durchlaufenen Knoten ab. Dabei spielt es jedoch keine Rolle, in welcher Reihenfolge parallele Kante in der Eulertour stehen. – Umgekehrt konstruiert man zu einem String $T \in \Omega$ eine Eulertour, indem man die Kanten von D_k in der Reihenfolge entfernt, wie sie in T vorkommen. Aufgrund von Satz 8 kommt man immer aus einem Knoten heraus, wenn man hineingelaufen ist, und aus Anzahlgründen (ähnlich wie in Proposition 5) werden auch tatsächlich alle Kanten verbraucht. Man hat jedoch noch zusätzlich die Freiheit, parallele Kanten in beliebiger Reihenfolge zu entnehmen. \square

Zufällige Permutationen kann man leicht generieren. Daher haben wir mit Lemma 9 die Ausgangsfrage reduziert auf das Problem, zufällige Eulertouren in $D_k(S)$ zu erzeugen.

Ende VL 2002-11-11

Reduktionsschritt 2: Von Eulertouren zu Arboreszenzen

Proposition 10. Sei $\text{Eul} = e_1, \dots, e_m$ eine Eulertour von u nach v . Für Knoten $w \in V$ bezeichnen wir die letzte Kante, über die w in Eul verlassen wird, mit $\text{le}(w)$ (für „last exit“). Dann ist

$$A(\text{Eul}) = (V, \{ \text{le}(w) \mid w \in V \setminus v \})$$

eine *Arboreszenz mit Wurzel* v , d.h. ein (spannender) Baum, in dem alle Kanten in Richtung von v zeigen.

Beweis. A hat $\#V - 1$ Kanten und ist kreisfrei, da ein Kreis nicht alle Knoten enthalten kann und also irgendwann von der Eulertour wieder verlassen werden müsste. Die letzte aus dem Kreis herausführende Kante widerspräche der Definition von $\text{le}(\cdot)$. Dass die Kanten zur Wurzel hin gerichtet sind, sieht man daran, dass für alle Knoten x die Kante $\text{le}(\text{head}(\text{le}(x)))$ später in der Eulertour auftritt als die Kante $\text{le}(x)$. Wenn man also die Knoten $x, \text{head}(\text{le}(x)), \text{head}(\text{le}(\text{head}(\text{le}(x))), \dots$ betrachtet, so bilden diese als Teilfolge der Eulertour einen gerichteten Pfad zur Wurzel. \square

Lemma 11. Sei B eine Arboreszenz mit Wurzel v . Dann gibt es genau

$$\deg^+(v)! \prod_{w \in V \setminus v} (\deg^+(w) - 1)! \quad (1)$$

viele Eulertouren Eul, die in v enden, mit $A(\text{Eul}) = B$.

Beweis. Wir wählen für jeden Knoten eine Permutation der herausführenden Kanten, die nicht in B liegen. Die Anzahl der Möglichkeiten dabei ist gerade (1). Durch den Endknoten v für eine Eulertour ist zugleich deren Anfangsknoten festgelegt. Sei dies u . Ausgehend von u folgen wir nun den Kanten in der Reihenfolge, die durch die Permutationen der herausführenden Kanten gegeben ist, und benutzen die Kante aus B nur, nachdem alle anderen bereits verbraucht sind. Die resultierende Eulertour erfüllt offenbar $A(\text{Eul}) = B$ und ist für jede Wahl von Permutationen verschieden. – Umgekehrt legt jede Eulertour von u nach v entsprechende Permutationen der herausführenden Kanten eindeutig fest. \square

Da, wie schon gesagt, Permutationen leicht zu erzeugen sind, haben wir nun das Ausgangsproblem darauf reduziert, eine zufällige Arboreszenz mit Wurzel v zu erzeugen. Siehe Abbildung 12.

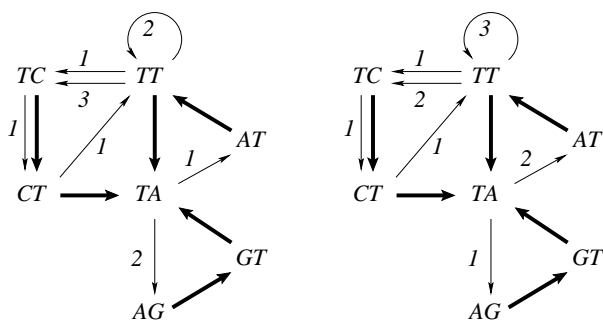


Abbildung 12: Beispiel zum Beweis von Lemma 11. Die dicken Kanten sind in der Arboreszenz. Die dünnen Kanten sind beschriftet mit den Permutationen für $TTCTTCTATTAGTA$ (links) und $TTCTTCTAGTATTTA$ (rechts).

Erzeugung zufälliger Arboreszenzen mittels eines „Backward Random Walk“

Die Idee des Algorithmus Backward Random Walk (BRW) ist einfach: Wir bewegen uns rückwärts entlang von Kanten von $D_k(S)$. In jedem Schritt wird der jeweils nächste Knoten bestimmt, indem zufällig

gleichverteilt unter allen Kanten, die in den aktuellen Knoten w hineinführen, eine ausgewählt wird. Falls der neue Knoten x zum ersten mal besucht wird, wird die dabei benutzte Kante als $le(x)$ zugewiesen.

```

for all  $w \in V$ : besucht( $w$ ) = no
 $w = v$ 
while ( noch nicht alle Knoten besucht )
    besucht( $w$ ) = yes
    wähle  $e \in E$  mit head( $e$ ) =  $w$  gleichverteilt
     $x = \text{tail}(e)$ 
    if (visited( $x$ ) = no)
        le( $x$ ) =  $e$ 
         $w = x$ 
    
```

Abbildung 13: Algorithmus BRW (Backward Random Walk) zur Erzeugung einer zufälligen Arboreszenz mit Wurzel v

Wir haben nun zwei Dinge zu beweisen:

Satz 12. Die vom Backward Random Walk gefundene Arboreszenz mit Wurzel v ist gleichverteilt unter allen solchen.

Satz 13. Die erwartete Anzahl von Schritten, bis der Backward Random Walk alle Knoten besucht hat, ist beschränkt durch $\#V^2 \#E$.

Die Laufzeitgarantie $\#\Sigma^{2k-2}(n-k+1)$ ist bei konstantem $\#\Sigma$ und k linear in $|S|$.

Grundlegendes zu Random Walks auf Graphen

Sei $P(x, y)$ die Wahrscheinlichkeit, dass der Random Walk im nächsten Schritt in y ist, wenn er zuletzt in x war. Offenbar gilt

$$\forall x : \sum_y P(x, y) = 1,$$

das heißt wir können P als eine $(V \times V)$ -Matrix auffassen, in der alle Zeilensummen gleich 1 sind.

Die Einträge der Matrix P^t sind die t -Schritt Übergangswahrscheinlichkeiten. P heißt *irreduzibel*, falls

$$\forall x, y \in V \exists t \geq 0 : P^t(x, y) > 0.$$

Dies bedeutet, dass man von jedem Knoten zu jedem anderen mit positiver Wahrscheinlichkeit in endlicher Zeit einmal kommt.

$D_k(S)$ ist eulersch (hat eine geschlossene Eulertour) genau dann, wenn S zyklisch ist. In diesem Fall sei $D_k(S) = D'_k(S)$. Ansonsten fügen wir eine zusätzliche („künstliche“) Kante von

$S[n - k + 2 .. n]$ nach $S[1 .. k - 1]$ ein. Sei $D'_k(S)$ in jedem Fall der resultierende Graph und P' die Übergangsmatrix für den Random Walk auf $D'_k(S)$. Dann ist offenbar P' irreduzibel, da $D'_k(S)$ nach Konstruktion eine geschlossene Eulertour hat. Für später merken wir uns, dass in jedem Knoten von $D'_k(S)$ gleich viele Kanten hinein- wie hinausgehen.

Es könnte nun sein, dass $D'_k(S)$ z.B. nur aus einem gerichteten Kreis besteht. Dann würde der Random Walk periodisch zum Ausgangspunkt zurückkehren. P heißt *aperiodisch*, falls

$$\forall x, y \in V \text{ gcd}\{t \mid P^t(x, y) > 0\} = 1.$$

(gcd = greatest common divisor, größter gemeinsamer Teiler)

Durch einen einfachen Trick kann man ein irreduzibles P' stets aperiodisch „machen“: Wir setzen $P := (P' + I)/2$. Das heißt, mit Wahrscheinlichkeit $1/2$ tut der Random Walk nichts, sondern bleibt stehen. (Man spricht auch von einer „lazy“ Markov chain.) Es gilt:

- P' irreduzibel
- $\Rightarrow P$ irreduzibel
- $\Rightarrow \forall x, y \in V \exists t : P^t(x, y) > 0$
- $\Rightarrow \forall x, y \in V \exists t \forall t' \geq t : P^{t'}(x, y) > 0$
- $\Rightarrow \exists t \forall x, y \in V \forall t' \geq t : P^{t'}(x, y) > 0$
- $\Rightarrow P$ aperiodisch.

Bei der Vertauschung der Quantoren wir ausgenutzt, dass es nur endlich viele Paare (x, y) gibt.

Es zeigt sich nun, dass diese beiden Eigenschaften bereits hinreichend dafür sind, dass die Verteilung des Random Walks konvergiert. Der folgende Satz 14 ist von fundamentaler Bedeutung für die Theorie der Markoffketten, soll an dieser Stelle aber nicht bewiesen werden.

Satz 14. Sei P irreduzibel und aperiodisch. Dann ist P *ergodisch*, d.h.,

$$\forall x, y \in V : P^t(x, y) \longrightarrow \pi(y) \text{ für } t \rightarrow \infty,$$

wobei π die eindeutig bestimmte *stationäre Verteilung* von P ist, d.h. $\pi P = \pi$.

Die Gleichung $\pi P = \pi$ bedeutet: Wenn die Wahrscheinlichkeit, mit der sich der Random Walk in y aufhält, gleich $\pi(y)$ ist, dann ist diese Wahrscheinlichkeit im nächsten Schritt ebenfalls gleich $\pi(y)$. Ergodizität bedeutet, dass die Markovkette gegen ihre stationäre Verteilung konvergiert. Diese ist eindeutig bestimmt und hängt insbesondere

auch nicht vom Startpunkt x ab. Die „lazy“ Modifikation einer ergodischen Markoffkette hat dieselbe stationäre Verteilung wie diese selbst.

Ende VL 2002-11-18

Korrektheit der Verteilung von Algorithmus BRW (Satz 12)

Wir wissen also, dass die stationäre Verteilung des Backward Random Walks existiert, aber wie sieht sie aus? Wir schauen uns die Gleichung $\pi P = \pi$ genauer an.

Sei $\tilde{\pi} := (\text{deg}^+(x) \mid x \in V)$. Es gilt

$$\begin{aligned} (\tilde{\pi} P)(y) &= \sum_x \tilde{\pi}(x) P(x, y) \\ &= \sum_x \text{deg}^+(x) \frac{f(y, x)}{\text{deg}^-(x)} \\ &= \sum_x f(y, x) \\ &= \text{deg}^+(y) = \tilde{\pi}(y). \end{aligned}$$

Also ist $\pi = \tilde{\pi}/Z$ für einen Normierungsfaktor Z ; die Aufenthaltswahrscheinlichkeit des Backward Random Walks in einem Knoten ist proportional zu dessen Grad. Diese Wahrscheinlichkeiten addieren sich zu 1 auf, d.h. $\sum_y \pi(y) = 1$. Daraus folgt $Z = \sum_y \tilde{\pi}(y) = \sum_y \text{deg}^+(y) = m$. Wir haben gezeigt:

Korollar 15. Sei G ein eulerscher gerichteter Graph mit m Kanten. Dann hat der (Backward) Random Walk auf G die stationäre Verteilung $\pi(y) = \text{deg}^+(y)/m$ auf den Knoten von G .

(Da der Graph die geforderten Eigenschaften behält, wenn man alle Kanten herumdreht, steht das „Backward“ in Klammern.)

Wir können uns nun vorstellen (ersparen uns hier aber eine explizite mathematische Konstruktion), dass der Backward Random Walk bereits seit unendlich langer Zeit mit der stationären Verteilung läuft und dies auch noch unendlich lange weiter tun wird. Gleichzeitig wollen wir auch festhalten, welche Kante in jedem Schritt benutzt worden ist.³ Wir haben also eine Zufallsvariable Y , deren Wert eine Abbildung

$$Y : \mathbb{Z} \rightarrow E, t \mapsto Y(t)$$

³Leider ist die Analyse in [KMUW] (insbesondere Theorem 2 dort) nur für den Fall ausgeführt, dass es keine parallelen Kanten gibt. Es gibt aber verschiedene Möglichkeiten, den Beweis zu reparieren. Die einfachste ist vielleicht, den Random Walk als Kantenfolge statt als Knotenfolge zu modellieren.

ist, so dass

$$\forall t \in \mathbb{Z} : \text{head}(Y(t)) = \text{tail}(Y(t-1)).$$

Jede solche Abbildung (auch als *Trajektorie* bezeichnet) beschreibt einen möglichen Verlauf des Random Walks. Die durchlaufenen Knoten sind dabei

$$X(t) := \text{head}(Y(t)).$$

Wir sagen, zum Zeitpunkt t befindet sich der Random Walk im Knoten $X(t)$ und wählt die Kante $Y(t)$, um ihn zu verlassen. Es gilt für alle $t \in \mathbb{Z}$, $w, x, y \in V$:

$$\begin{aligned} \Pr(X(t) = w) &= \pi(w) = \frac{\text{deg}^+(w)}{m} \\ \Pr(X(t) = y \mid X(t-1) = x) &= P(x, y) = \frac{f(x, y)}{\text{deg}^+(x)} \end{aligned}$$

Für beliebige Ereignisse A, B gilt:

$$\begin{aligned} \Pr(A \mid B) &\triangleq \frac{\Pr(A \cap B)}{\Pr(B)} \\ &= \frac{\Pr(A \cap B) \Pr(A)}{\Pr(A) \Pr(B)} \triangleq \frac{\Pr(B \mid A) \Pr(A)}{\Pr(B)}. \end{aligned}$$

Also gilt für alle $t \in \mathbb{Z}$, $x, y \in V$ auch:

$$\begin{aligned} \Pr(X(t-1) = x \mid X(t) = y) &= \\ \frac{\frac{f(x, y)}{\text{deg}^+(x)} \cdot \frac{\text{deg}^+(x)}{m}}{\frac{\text{deg}^+(y)}{m}} &= \frac{f(x, y)}{\text{deg}^+(y)}. \end{aligned}$$

Zu jeder Trajektorie Y des Random Walks und jedem Zeitpunkt $t \in \mathbb{Z}$ erhalten wir eine Arboreszenz mit Wurzel $X(t)$, indem wir $\text{le}()$ analog zum Algorithmus BRW mit Startknoten $v = X(t)$ definieren, dabei jedoch deterministisch den Pfad $Y(t), Y(t+1), \dots$ entlang der Rückwärtskanten in $D'_k(S)$ ablaufen, anstatt jeweils $e \in E$ mit $\text{head}(x) = w$ gleichverteilt zufällig zu wählen. Die so erhaltene Arboreszenz bezeichnen wir mit $T(t)$. Da der Backward Random Walk mit Wahrscheinlichkeit 1 alle Knoten des Graphen irgendwann einmal besucht (obwohl es evtl. Trajektorien gibt, die das nicht tun!), ist $T(t)$ *fast sicher* ($:\Leftrightarrow$ mit Wahrscheinlichkeit 1) wohldefiniert.

Wie unterscheiden sich $T(t)$ und $T(t-1)$ voneinander? Offenbar ist $Y(t-1) \in T(t-1)$, da die erste Kante immer zu einem unbesuchten Knoten führt, also $\text{le}(X(t)) = Y(t-1)$ gesetzt wird. Andererseits fällt dafür die Kante $\text{le}_{T(t)}(X(t-1)) \in T(t)$ heraus, denn $X(t-1)$ ist die (neue) Wurzel von $T(t-1)$ und wird gleich zu Beginn von Algorithmus BRW als besucht markiert.

Wir haben bereits gesehen, dass

$$\Pr(X(t-1) = x \mid X(t) = y) = \frac{f(x, y)}{\text{deg}^+(y)}.$$

Jede der Kanten, die nach $X(t)$ hineinführen, ist mit gleicher Wahrscheinlichkeit in $T(t-1)$ enthalten. Wir können daher T als einen Random Walk auf der Menge der Arboreszenzen mit beliebigen Wurzeln auffassen, wobei aber die Zeit *rückwärts* läuft.

Was wissen wir von T , aufgefasst als Random Walk auf der Menge der Arboreszenzen von $D'_k(S)$, der in umgekehrter Zeitrichtung läuft? Jede Arboreszenz $T(t)$ hat $\text{deg}^-(\text{root}(T(t)))$ viele mögliche Nachfolger (!) $T(t-1)$, die alle gleichwahrscheinlich sind – soviele Kanten führen in die Wurzel von $T(t)$ hinein. Jede Arboreszenz (t) hat aber auch $\text{deg}^+(\text{root}(T(t)))$ viele mögliche Vorgänger (!) $T(t+1)$, die alle gleichwahrscheinlich sind – soviele Kanten führen aus der Wurzel von $T(t)$ heraus. Also ist T ein Random Walk auf einem Graphen H , dessen Knotenmenge die Menge aller Arboreszenzen auf $D'_k(S)$ ist, und in dem für alle Knoten A gilt:

$$\begin{aligned} \text{deg}_H^+(A) &= \text{deg}_H^-(A) \\ &= \text{deg}_{D'_k(S)}^+(\text{root}(A)) = \text{deg}_{D'_k(S)}^-(\text{root}(A)). \end{aligned}$$

Der Random Walk T auf H ist irreduzibel, da H stark zusammenhängend ist. Dies sieht man z.B. so: Seien A und B Arboreszenzen. Erweitere beide zu geschlossenen Eulertouren in $D'_k(S)$ wie im Beweis von Lemma 11. Hänge beide Eulertouren hintereinander. Dieses Teilstück eines möglichen Random Walks zeigt, dass man von A nach B kommt über Kanten in H .

Wir haben gerade gesehen, wie die stationäre Verteilung ψ des Random Walks auf einem eulerschen Graphen H aussieht (Korollar 15): $\psi(A)$ ist proportional zu $\text{deg}_H^+(A)$. *Insbesondere haben alle Arboreszenzen mit derselben Wurzel die gleiche Wahrscheinlichkeit.* Daraus folgt, dass Algorithmus BRW, gestartet in v , eine gleichverteilt zufällige Arboreszenz mit Wurzel v liefert. Wir haben Satz 12 bewiesen.

Laufzeit von Algorithmus BRW (Satz 13)

Wir schätzen nun ab, wie lange der Backward Random Walk braucht, um alle Knoten zu besuchen. Zunächst benötigen wir den Begriff der *hitting time*. Sei $X_w : \mathbb{N}_0 \rightarrow V(D'_k(S))$ der Backward Random Walk auf $D'_k(S)$, der zum Zeitpunkt 0 im Knoten w startet, d.h. $X_w(0) = w$. Dann ist $\text{hit}(w, u)$ gleich

der Zeit, die X_w benötigt, um nach u zu kommen. Allgemeiner definieren wir für $U \subseteq V$:

$$\text{hit}(w, U) := \min\{t \geq 1 \mid X_w(t) \in U\}.$$

Lemma 16. Sei $U \subseteq V$ mit $\emptyset \neq U \neq V$. Dann gibt es ein $w \in V \setminus U$, so dass $E(\text{hit}(w, U)) \leq m - 1$.

Beweis. Sei $D'_k(S)/U$ der Graph, der aus $D'_k(S)$ dadurch hervorgeht, dass alle Knoten in U zu einem neuen Knoten u kontrahiert werden. $D'_k(S)/U$ ist eulersch, da gleich viele Kanten nach U hinein wie aus U heraus führen. Nach Korollar 15 ist die Dichte der stationären Verteilung des Backward Random Walks auf $D'_k(S)/U$ proportional zum Knotengrad. Da $\deg^+(u) \geq 1$ und $\sum_{w \in V \setminus U} \deg^+(w) \leq m - 1$, folgt $\pi(u) \geq 1/m$.

Für jede ergodische Markoffkette $(X(t) \mid t \geq 0)$ gilt für $t \rightarrow \infty$ das folgende (Beweis weggelassen, anschaulich klar!):

$$\frac{\#\{s \leq t \mid X(t) = u\}}{t} \rightarrow \pi(u)$$

$$\frac{\#\{s \leq t \mid X(t) = u\}}{\#\{s \leq t \mid X(t) = u\}} \rightarrow E(\text{hit}(u, u))$$

Insbesondere ist also $E(\text{hit}(u, u)) = 1/\pi(u)$.

Daraus folgt $E(\text{hit}(u, u)) \leq m$, das heißt, die erwartete Zeit, die der in u gestartete Random Walk braucht, um nach u zurückzukehren, ist $\leq m$. Im ersten Schritt wird aber u verlassen. Also muss es unter den möglichen Nachfolgern von u einen Knoten w geben, von dem aus u in erwartet $\leq m - 1$ Schritten erreicht wird. \square

Lemma 17.

$$\forall x, u \in V \ E(\text{hit}(x, u)) \leq (n - 1)(m - 1).$$

Beweis. Setze $U_1 := \{u\}$. Nach Lemma 16 gibt es ein $w \in V \setminus U_1$ mit $E(\text{hit}(w, U_1)) \leq m - 1$. Setze $U_2 := U_1 \cup \{w\}$ und iteriere den Prozess, bis $U_n = V$ erreicht ist. Da $x \in V$, folgt die Behauptung. \square

Aus Lemma 17 folgt, dass die erwartete Zeit, die der Backward Random Walk braucht, um alle Knoten in einer beliebigen Reihenfolge v_1, v_2, \dots, v_n zu besuchen, durch $(n - 1)^2(m - 1)$ beschränkt ist. Wir haben Satz 13 bewiesen.

Überblick und Laufzeit Wir fassen noch einmal den gesamten Algorithmus zusammen:

1. Konstruiere $D_k(S)$. (Z.B. als $(\#\Sigma^{k-1} \times \#\Sigma)$ -Matrix mit Einträgen in \mathbb{N}).
2. Falls S zyklisch ist, wende eine „random rotation“ an, um einen Anfangs=Endknoten der Eulertour auszuwählen, und setze $D'_K(S) = D_K(S)$.
Falls S azyklisch ist, füge eine künstliche Kante ein, um einen eulerschen $D'_K(S)$ zu erhalten.
3. Finde eine zufällige Arboreszenz in $D'_K(S)$ mittels Algorithmus BRW.
4. Wähle zufällige Permutationen für die restlichen Kanten.
5. Lese die Eulertour ab.

Die Schritte 1, 2, 5 können offenbar in $O(\#S + \#\Sigma^k)$ Zeit und Platz durchgeführt werden. Die Laufzeit von BRW ist $O(\#\Sigma^{2k} \#S)$ nach Satz 13. Eine zufällige Permutation von p Zahlen kann in $O(p)$ erzeugt werden⁴, also benötigt Schritt 4 insgesamt $O(\#S)$ Zeit. Die Gesamtlaufzeit wird also dominiert von BRW. In Anwendungen wird man in der Regel k so wählen, dass die meisten k -lets mindestens einmal vorkommen. In diesem Fall ist $\#\Sigma^k = O(\#S)$ und die Laufzeit $O(\#S)$, d.h. linear in der Stringlänge.

3 Proteinstruktur

Bei der Untersuchung der dreidimensionalen Struktur von Molekülen stellt sich die Aufgabe, zu zwei gegebenen Punktmengen im Raum eine starre Bewegung zu finden, die ähnliche Teile aus beiden Mengen zur Deckung bringt. Das Problem zerfällt in zwei Teilschritte:

1. Auswahl eines Matchings zwischen beiden Punktmengen. (Das Matching muss nicht alle Atome überdecken.)
2. Finden der starren Bewegung, die den mittleren quadratischen Abstand zwischen den einander zugeordneten Atomen minimiert.

Wir betrachten zunächst den zweiten Teilschritt.

⁴Es soll immer noch Leute geben, die nicht wissen, wie das geht. Also: Initialisiere ein Array $A[1..p]$ mit $A[i] = i$. Für $i = p, \dots, 2$: Vertausche $A[i]$ mit $A[\text{random}(i)]$, wobei $\text{random}(i)$ eine Zufallszahl im Bereich $[1..i]$ zurückgibt.

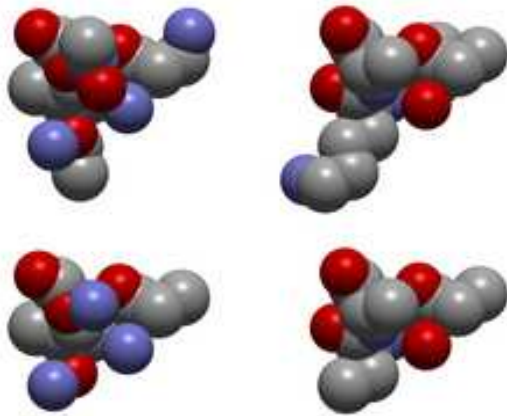


Abbildung 14: Zwei molecular surface patches (oben) und ihre überlagerten Teile (unten).

3.1 Optimale Überlagerung bei gegebenem Matching

Seien $A, B \in \mathbb{R}^{3 \times m}$ die Koordinaten der gematchten Atome. Wir schreiben

$$A_i := \text{Koordinaten } i\text{-tes Atom von Patch } A$$

$$= \begin{pmatrix} A_{1,i} \\ A_{2,i} \\ A_{3,i} \end{pmatrix}.$$

Für zwei Punkte $x, y \in \mathbb{R}^3$ ist der *euklidische Abstand* durch die ℓ_2 -Norm gegeben, d.h.,

$$|x - y| = \sqrt{\langle x - y, x - y \rangle}$$

$$= \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2}$$

$$= \left(\sum_{j=1}^3 (x_j - y_j)^2 \right)^{1/2}.$$

Hiebei bezeichnet $\langle a, b \rangle = a^\top b = \sum_{i=1}^n a_i b_i$ das Standardskalarprodukt im \mathbb{R}^n .

Für Folgen $A, B \in \mathbb{R}^{3 \times m}$ von Punkten ist der *mittlere quadratische Abstand* (*root mean square distance, rms*) gleich dem Mittelwert gemäß der ℓ_2 -Norm aus den paarweisen Abständen der gematchten Punkte, d.h.

$$\text{rms}(A, B) = \left(\frac{1}{m} \sum_{i=1}^m |A_i - B_i|^2 \right)^{1/2}$$

$$= \left(\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^3 (A_{i,j} - B_{i,j})^2 \right)^{1/2}.$$

Man kann nun zeigen, dass jede starre Bewegung im \mathbb{R}^3 die Form

$$x \mapsto Rx + t$$

hat, wobei $R \in \mathbb{R}^{3 \times 3}$ eine *Rotation* ist und $t \in \mathbb{R}^3$ ein *Translationsvektor* (Verschiebung) ist.

Ein Beispiel für eine Rotation im \mathbb{R}^2 : $R = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$, $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto Rx = \begin{pmatrix} -x_2 \\ x_1 \end{pmatrix}$ ist die Drehung um 90° nach links. Im \mathbb{R}^3 beschreibt $R = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ die entsprechende Drehung, wobei die dritte Koordinate die Drehachse bildet.

Aus der Linearen Algebra weiß man, dass die Rotationsmatrizen gerade die Matrizen R mit den Eigenschaften $R^\top R = I$ (Orthonormalität) und $\det(R) = 1$ sind. Die Bedingung $\det(R) = 1$ dient dazu, Spiegelungen auszuschließen. Wir erhalten die folgende Problemstellung (für Atomkoordinaten ist $n = 3$):

Überlagerung für ein gegebenes Matching

Gegeben: Punktmengen $A, B \in \mathbb{R}^{n \times m}$

Gesucht: Starre Bewegung, gegeben durch $R \in \mathbb{R}^{n \times n}$ mit $R^\top R = I$ und $\det(R) = 1$ sowie $t \in \mathbb{R}^n$, welche

$$\left(\sum_{i=1}^m |A_i - (RB_i + t)|^2 \right)^{1/2}$$

minimiert.

Bestimmung des Translationsvektors t

Wir bezeichnen die *Schwerpunkte* der beiden Punktmengen mit $\tilde{A} := \frac{1}{m} \sum_{i=1}^m A_i$, für \tilde{B} analog.

Lemma 18. Die optimale starre Bewegung bildet die Schwerpunkte aufeinander ab, erfüllt also $t = \tilde{A} - R\tilde{B}$.

Beweis. Wir können ohne Beschränkung der Allgemeinheit annehmen, dass $\tilde{B} = 0$ ist. Andernfalls ermitteln wir zunächst die optimale starre Bewegung $x \mapsto Rx + \tilde{A}$ für die Punkte A_i und $B'_i := B_i - \tilde{B}$. Dann leistet $x \mapsto R(x - \tilde{B}) + \tilde{A} = Rx + \tilde{A} - R\tilde{B}$ das gewünschte. Sei also $\tilde{B} = 0$; wir zeigen, dass $t = \tilde{A}$.

Für beliebige Vektoren $a, b \in \mathbb{R}^n$ ist $|a + b|^2 = \langle a + b, a + b \rangle = \langle a, a \rangle + \langle b, b \rangle + 2\langle a, b \rangle = |a|^2 + |b|^2 + 2\langle a, b \rangle$ und für $a, b, t \in \mathbb{R}^n$ gilt also $|a - (b + t)|^2 = |a|^2 + |b + t|^2 - 2\langle a, b + t \rangle = |a|^2 + |b|^2 + |t|^2 + 2(\langle b, t \rangle -$

$\langle a, b \rangle - \langle a, t \rangle$). Daher ist

$$\sum_{i=1}^m |A_i - RB_i - t|^2 = \sum_{i=1}^m (|A_i|^2 + |RB_i|^2 + |t|^2) - 2 \sum_{i=1}^m (\langle RB_i, t \rangle - \langle A_i, RB_i \rangle - \langle A_i, t \rangle)$$

Aufgrund der Definitionen gilt $\langle Mx, y \rangle = \langle x, M^T y \rangle$ für beliebige Vektoren x, y und Matrizen M . Also ist $\sum_{i=1}^m \langle RB_i, t \rangle = \sum_{i=1}^m \langle B_i, R^T t \rangle = \langle \sum_{i=1}^m B_i, R^T t \rangle = \langle m\bar{B}, R^T t \rangle = \langle 0, R^T t \rangle = 0$ und $|RB_i|^2 = \langle RB_i, RB_i \rangle = \langle B_i, R^T RB_i \rangle = \langle B_i, B_i \rangle = |B_i|^2$. Nach Umgruppieren folgt

$$\sum_{i=1}^m |A_i - RB_i - t|^2 = \sum_{i=1}^m |B_i|^2 \tag{2}$$

$$+ \sum_{i=1}^m |A_i - t|^2 \tag{3}$$

$$- 2 \sum_{i=1}^m \langle A_i, RB_i \rangle. \tag{4}$$

Der Term (2) ist hängt weder von R noch von t ab, spielt also für die Optimierung keine Rolle. Der Term (3) hängt nur von t ab, und der Term (4) hängt nur von R ab. Damit haben wir das Problem in zwei Teilprobleme zerlegt, die unabhängig voneinander optimiert werden können.

Ende VL 2002-12-02

Das optimale t minimiert (3). Eine notwendige Bedingung dafür ist, dass die partiellen Ableitungen nach t verschwinden, das heißt

$$0 = \frac{\partial}{\partial t_j} \sum_{i=1}^m |A_i - t|^2 \quad \text{für } j = 1, \dots, n.$$

Wir schreiben $|A_i - t|^2 = |A_i|^2 + |t|^2 - 2\langle A_i, t \rangle$. Es gilt

$$\frac{\partial}{\partial t_j} |t|^2 = \frac{\partial}{\partial t_j} (t_1^2 + \dots + t_n^2) = 2t_j$$

und

$$\frac{\partial}{\partial t_j} \langle A_i, t \rangle = \frac{\partial}{\partial t_j} (A_{i1}t_1 + \dots + A_{in}t_n) = A_{ij}.$$

Wir erhalten also die Bedingung

$$0 = 2 \sum_{i=1}^m (t - A_i),$$

woraus sofort $mt = \sum_{i=1}^m A_i$, also $t = \frac{1}{m} \sum_{i=1}^m A_i = \bar{A}$ folgt. Das war zu zeigen. \square

Bemerkung. Im Beweis von Lemma 18 haben wir verwendet, dass es aus geometrischen Gründen ein Optimum geben muss. $t = \bar{A} - R\bar{B}$ ist der einzige „Kandidat“, der die notwendige Bedingung erfüllt.

Bestimmung der Rotationsmatrix R

Gemäß (4) aus dem Beweis von Lemma 18 suchen wir ein $R \in \mathbb{R}^{n \times n}$ mit $R^T R = I$ und $\det R = 1$, welches $\sum_{i=1}^m \langle A_i, RB_i \rangle$ maximiert. Wegen

$$- 2 \sum_{i=1}^m \langle A_i, RB_i \rangle = \sum_{i=1}^m |A_i - RB_i|^2 - \underbrace{\sum_{i=1}^m |A_i|^2 - \sum_{i=1}^m |RB_i|^2}_{\text{konstant}}$$

ist dies äquivalent zur Minimierung von

$$\sum_{i=1}^m |A_i - RB_i|^2 = \sum_{i=1}^m \sum_{j=1}^n (A - RB)_{j,i}^2 = |A - RB|_F^2. \tag{5}$$

Hierbei bezeichnet

$$|M|_F := \left(\sum_{i,j} M_{i,j}^2 \right)^{1/2}$$

die *Frobenius-Norm* einer Matrix. Die Frobenius-Norm einer Matrix ist einfach die ℓ_2 -Norm der Matrix, wenn man diese als Vektor auffasst.

Die Minimierung von (5) ist insofern etwas schwieriger, als man dabei die Nebenbedingungen $R^T R = I$ und $\det R = 1$ zu beachten hat. Abhilfe schafft in dieser Situation ein genialer Trick, die sogenannten *Lagrange-Multiplikatoren*. Die Matrix R erfüllt nämlich genau dann die Nebenbedingungen, wenn die partiellen Ableitungen von

$$\sum_{i,j=1}^n (R^T R - I)_{i,j} L_{i,j} + (\det R - 1)\lambda$$

nach L und λ verschwinden. (Da $R^T R$ eine symmetrische Matrix ist, können wir auch L als symmetrische Matrix ansetzen.) Wir müssen also die partiellen Ableitungen von

$$Z(R, L, \lambda) := |A - RB|_F^2 + \sum_{i,j=1}^n (R^T R - I)_{i,j} L_{i,j} + (\det R - 1)\lambda$$

nach R , L und λ bestimmen.

Wie schon die ℓ_2 -Norm, können wir auch das Standardskalarprodukt auf Matrizen anwenden, die wir zu diesem Zweck als Vektoren auffassen. Dann gilt $\|A - RB\|_F^2 = \langle A - RB, A - RB \rangle = \langle A, A \rangle - 2\langle A, RB \rangle + \langle RB, RB \rangle$. Hier ist

$$\begin{aligned} \langle A, RB \rangle &= \sum_{i,j} A_{i,j} \sum_h R_{i,h} B_{h,j} \\ &= \sum_{i,h} R_{i,h} \underbrace{\sum_j A_{i,j} B_{h,j}}_{=(AB^\top)_{i,h}} \end{aligned}$$

Daraus folgt $\frac{\partial}{\partial R} \langle A, RB \rangle = AB^\top$. Außerdem ist

$$\begin{aligned} \langle RB, RB \rangle &= \sum_{i,j} \sum_g R_{i,g} B_{g,j} \sum_h R_{i,h} B_{h,j} \\ &= \sum_{i,g,h} R_{i,g} R_{i,h} \underbrace{\sum_j B_{g,j} B_{h,j}}_{=(BB^\top)_{g,h}} \end{aligned}$$

woraus folgt

$$\frac{\partial}{\partial R_{k,\ell}} \langle RB, RB \rangle = 2 \sum_h \underbrace{R_{k,h} (BB^\top)_{\ell,h}}_{=(RBB^\top)_{k,\ell}}$$

also $\frac{\partial}{\partial R} \langle RB, RB \rangle = 2RBB^\top$. Des weiteren ist

$$\begin{aligned} \langle R^\top R, L \rangle &= \sum_{i,j} \sum_h R_{h,i} R_{h,j} L_{i,j} \\ &= \sum_{h,j} R_{h,j} \underbrace{\sum_i R_{h,i} L_{i,j}}_{=(RL)_{h,j}} \end{aligned}$$

also $\frac{\partial}{\partial R} \langle R^\top R, L \rangle = 2RL$. Man braucht ein bisschen mehr Lineare Algebra, um zu zeigen, dass $\frac{\partial}{\partial R} \det R = R$ gilt, falls R eine Rotation ist. (Diese Identität gilt also nur für solche R , die die anderen Nebenbedingungen erfüllen.) Das geht so: Für eine beliebige quadratische Matrix M definiert man die komplementäre Matrix \tilde{M} durch $\tilde{M}_{i,j} = \det M_{(j,i)}$, wobei wir mit $M_{(i,j)}$ die Matrix

$$\begin{pmatrix} M_{1,1} & \dots & M_{1,j-1} & 0 & M_{1,j+1} & \dots & M_{1,n} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ M_{i-1,1} & \dots & M_{i-1,j-1} & 0 & M_{i-1,j+1} & \dots & M_{i-1,n} \\ 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ M_{i+1,1} & \dots & M_{i+1,j-1} & 0 & M_{i+1,j+1} & \dots & M_{i+1,n} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ M_{n,1} & \dots & M_{n,j-1} & 0 & M_{n,j+1} & \dots & M_{n,n} \end{pmatrix}$$

bezeichnen. (Beachte die Vertauschung der Indizes in der Definition von \tilde{M} .) Aus dem Laplaceschen Entwicklungssatz folgt unmittelbar, dass $\frac{\partial}{\partial M} \det M = \tilde{M}^\top$. Ebenso überzeugt man sich durch elementare Rechnungen (siehe [Fis]), dass

$$M\tilde{M} = \tilde{M}M = \det(M) \cdot I.$$

Falls M invertierbar ist, gilt also $M^{-1} = \tilde{M} / \det M$. Da R eine orthonormale Matrix ist, gilt $R^{-1} = R^\top$, und zusammen mit $\det R = 1$ folgt $\tilde{R} = R^\top$. Also ist $\frac{\partial}{\partial R} \det R = \tilde{R}^\top = R^{\top\top} = R$.

Wir erhalten also aus dem Ansatz mit den Lagrange-Multiplikatoren die Bedingungen:

$$0 = \frac{\partial Z}{\partial R} = -2AB^\top + 2RBB^\top + 2RL + R\lambda, \quad (6)$$

$$0 = \frac{\partial Z}{\partial L} = R^\top R - I, \quad (7)$$

$$0 = \frac{\partial Z}{\partial \lambda} = \det R - 1. \quad (8)$$

Ende VL 2002-12-09

Definition 19. Sei $M \in \mathbb{R}^{n \times n}$.

1. Dann existieren eine Diagonalmatrix $D = \text{diag}(d)$, $d_1 \geq d_2 \geq \dots \geq d_n \geq 0$ und orthonormale Matrizen U und V , so dass $M = UDV^\top$. Die Einträge von d heißen *singuläre Werte* (Singularwerte) von M und sind eindeutig bestimmt.
2. Falls $M = M^\top$, so existieren eine Diagonalmatrix $\tilde{D} = \text{diag}(\tilde{d})$, $|\tilde{d}_1| \geq |\tilde{d}_2| \geq \dots \geq |\tilde{d}_n|$ und eine orthonormale Matrix Q , so dass $M = Q\tilde{D}Q^\top$. Die Einträge von \tilde{d} sind die *Eigenwerte* von M und eindeutig bestimmt.

Anmerkung. Die Forderung, dass die Diagonaleinträge sortiert sind, kann man leicht dadurch erfüllen, dass man eine „unsortierte“ Diagonalmatrix D zunächst mit einer Permutationsmatrix P in die gewünschte Form $D' := PDP^\top$ bringt. Setze $U' := UP^\top$, etc., dann gilt $M = (UP^\top)(PDP^\top)(PV^\top) = U'D'V'^\top$ mit $U'^\top U' = PU^\top UP^\top = I$, etc..

Beide Zerlegungen können effizient berechnet werden.

Sei nun

$$AB^\top = UDV^\top$$

eine Singulärwertzerlegung der Matrix AB^\top . Wir schreiben (6) als

$$0 = -2AB^\top + 2R \underbrace{(BB^\top + L + \frac{\lambda}{2}I)}_{=: K}.$$

Dann gilt $RK = AB^\top$ und (nach Transponierung – K ist symmetrisch) $KR^\top = BA^\top$. Daraus folgt

$$K^2 = KR^\top RK = BA^\top AB^\top = VDU^\top UDV^\top = VD^2V^\top.$$

Andererseits hat K als symmetrische Matrix eine orthonormale Diagonalisierung

$$K = Q\tilde{D}Q^\top.$$

Dann ist aber $K^2 = Q\tilde{D}^2Q^\top$ eine weitere orthonormale Diagonalisierung von K^2 . Da die Eigenwerte von K^2 eindeutig bestimmt sind, folgt zunächst einmal, dass $D^2 = \tilde{D}^2$. Es gibt also eine Diagonalmatrix $S = \text{diag}(s)$ mit $s_j = \pm 1$, so dass

$$\tilde{D} = DS.$$

Damit gilt also $K = QDSQ^\top$. Wir „wollen“ aber $K = VDSV^\top$.

Wir machen nun die *Annahme*, dass die Singulärwerte von AB^\top alle verschieden sind. Dann sind die Diagonaleinträge von D alle verschieden, ebenso für \tilde{D} . Daher sind die zugehörigen Eigenräume von K sämtlich eindimensional. Daraus folgt, dass die Spalten Qe_j und Ve_j Eigenvektoren von K^2 zum Eigenwert $\tilde{d}_j^2 = d_j^2$ sind und zudem $|Qe_j| = |Ve_j| = 1$ erfüllen. Also ist $Qe_j = \pm Ve_j$. Es gibt also eine Diagonalmatrix $\hat{S} = \text{diag}(\hat{s})$, $\hat{s}_i = \pm 1$, so dass $Q = V\hat{S}$. Wir sehen dass

$$K = QDSQ^\top = V\hat{S}DS\hat{S}^\top = VDSV^\top,$$

da Diagonalmatrizen kommutieren und $\hat{S}^2 = I$.

Wie sieht nun die Vorzeichenmatrix S aus? Wir bestimmen zunächst die Determinante. Es gilt

$$\det(K) = \det(VDSV^\top) = \det(D)\det(S)$$

$$\det(K) = \det(KR) = \det(AB^\top)$$

und da $d \geq 0$ ist, gilt $\det(D) \geq 0$. Daraus folgt

$$\det(AB^\top) > 0 \Rightarrow \det(S) = 1$$

$$\det(AB^\top) < 0 \Rightarrow \det(S) = -1.$$

Den Fall $\det(AB^\top) = 0$ betrachten wir später.

Wir zeigen nun, wie man S zu wählen hat, damit $Z(R, L, \lambda)$ minimiert wird. Die Minimierung von $\|A - RB\|_F^2$ ist, wie gesehen, äquivalent zur Maximierung von $\langle A, RB \rangle$.

Für beliebige Matrizen X, Y gilt $\langle X, Y \rangle = \sum_{i,j} X_{i,j}Y_{i,j} = \text{tr}(XY^\top)$, wobei $\text{tr}(M) := \sum_i M_{i,i}$ die *Spur* („trace“, Summe der Diagonaleinträge) bezeichnet. Offenbar ist $\langle X, Y \rangle = \langle X^\top, Y^\top \rangle =$

$\text{tr}(X^\top Y) = \text{tr}(Y^\top X)$, woraus $\text{tr}(XY) = \text{tr}(YX)$ folgt. Man kann also die Faktoren „rotieren“.

$$\begin{aligned} \langle A, RB \rangle &= \text{tr}(AB^\top R^\top) \\ &= \text{tr}(\underbrace{R^\top AB^\top}_{=K=VDSV^\top}) \\ &= \text{tr}(VDSV^\top) \\ &= \text{tr}(V^\top VDS) \\ &= \text{tr}(DS) \\ &= \sum_j d_j s_j, \end{aligned}$$

da D und S beides Diagonalmatrizen sind. Das optimale s sieht also wie folgt aus: $s_i = 1$ für $i \leq n-1$,

$$s_n = \begin{cases} 1 & \det(AB^\top) > 0 \\ -1 & \det(AB^\top) < 0. \end{cases}$$

Das optimale R ergibt sich daraus wie folgt: Es gilt

$$RK = AB^\top = UDV^\top$$

$$RK = R(VDSV^\top),$$

also

$$RVDSV^\top = UDV^\top$$

$$\begin{array}{l|l} RVDS = UD & | DS = SD \\ RVSD = UD & | \exists D^{-1} \\ RVS = U & | S^{-1} = S \\ RV = US & \\ R = USV^\top. & \end{array}$$

Im Fall $\det(AB^\top)$ existiert D^{-1} nicht mehr, aber wenn $\text{rang}(AB^\top) = n-1$, so ist R immer noch eindeutig bestimmt. Wegen $d_n = 0$ hat die Wahl von s_n in diesem Fall zwar keinen Einfluss auf $\|A - RB\|_F^2$, wir müssen aber die Bedingung $\det(R) = 1$ sicherstellen. Wir haben

$$RVDSV^\top = UDV^\top$$

$$\begin{array}{l|l} RVDS = UD & | DS = D \\ RVD = UD & \\ U^\top RVD = D & \end{array}$$

und durch Spaltenvergleich folgt

$$U^\top RVe_j = e_j \quad \text{für } j = 1, \dots, n-1.$$

Andererseits ist $U^\top RV$ als Produkt von Orthonormalmatrizen eine Orthonormalmatrix, daher folgt

$$U^\top RVe_n = \pm e_n.$$

Aus der Vorzeichenbedingung für $\det(R) = 1$ folgt nun $R = U(U^T R V) V^T = U S V^T$, wobei $s_i = 1$ für $i \leq n - 1$,

$$s_n = \begin{cases} 1 & \det(U) \det(V) > 0 \\ -1 & \det(U) \det(V) < 0. \end{cases}$$

Im Fall $\text{rang}(AB^T) \leq n - 2$ ist die Rotation nicht mehr eindeutig bestimmt. In diesem Fall gilt $d_n = d_{n-1} = 0$, man kann also stets z.B. (s_{n-1}, s_n) durch $(-s_{n-1}, -s_n)$ ersetzen und erhält eine weitere optimale Lösung. Dieser Fall tritt ein, wenn eine der Punktmengen auf einer Geraden im \mathbb{R}^n liegt.

Algorithmus und Laufzeit „Trotz des langen Beweises“ erhalten wir einen einfachen Algorithmus:

1. Gegeben: Punktmengen $A, B \in \mathbb{R}^{n \times m}$.
2. Berechne die Schwerpunkte $\tilde{A}, \tilde{B} \in \mathbb{R}^n$ von A und B , berechne B' mittels $B'_i := B_i - \tilde{B}$.
3. Berechne $AB'^T \in \mathbb{R}^{n \times n}$.
4. Berechne die Singulärwertzerlegung $AB'^T = UDV^T$.
5. Bestimme S , indem die $n - 1$ größten Singulärwerte in D durch 1 ersetzt werden. Falls $\det(AB'^T) > 0$ oder $\det(AB'^T) = 0 \wedge \det(U) \det(V) > 0$, ersetze den kleinsten Singulärwert durch 1, andernfalls durch -1 .
6. $R := USV^T$.
7. $t := \tilde{A} - R\tilde{B}$.
8. Dann ist $x \mapsto Rx + t$ eine starre Bewegung, die $\text{rms}(A, RB + t)$ minimiert.

Für eine konstante Dimension n wird die Laufzeit dominiert von den Schritten 2 und 3 und ist $O(m)$. Die restlichen Schritte können in $O(1)$ erledigt werden.

Ende VL 2002-12-16

Zur Frage der Eindeutigkeit der optimalen starren Bewegung

In Umeyama [Ume] wird behauptet, dass R und t eindeutig bestimmt sind, falls $\text{rang}(AB^T) \geq n - 1$. Das folgende Gegenbeispiel zeigt jedoch, dass sogar im Fall $\text{rang}(AB^T) = n$ mehrere optimale starre Bewegungen existieren können. Genauer: es gibt mehrere gleich gute Rotationsmatrizen R . Das Problem ist die Mehrdeutigkeit der Singulärwertzerlegung $AB^T = UDV^T$ in dem Fall, wo es mehrere

gleiche Singulärwerte gibt. Die optimale starre Bewegung ist nur dann eindeutig, wenn alle Singulärwerte verschieden sind. Diese Bedingung verschärft $\text{rang}(AB^T) \geq n - 1$. [E-mail von Shinji Umeyama, Jan. 2003]

Wir betrachten die Eckpunkte eines um den Ursprung zentrierten Tetraeders:

$$p = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, q = \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix}, r = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}, s = \begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix}.$$

Die Matrizen seien

$$A = (p, q, r, s) \quad \text{und} \quad B = (p, q, s, r).$$

Der optimale Translationsvektor ist also $t = 0$. Wir haben

$$AB^T = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 0 & 4 \\ 0 & 4 & 0 \end{pmatrix}.$$

Mit den Matrizen

$$D = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{pmatrix} \quad \text{und} \quad P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

(beachte, dass $P = P^T, P^2 = I$) erhalten wir die beiden Singulärwertzerlegungen

$$AB^T = IDP^T = PDI^T.$$

Da

$$\det(AB^T) = -64 < 0,$$

ist $S = \text{diag}(s)$ mit $s_1 = s_2 = 1, s_3 = -1$ zu setzen, und wir erhalten die beiden Rotationsmatrizen

$$R^{(1)} = ISP = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$$

und

$$R^{(2)} = PSI = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}.$$

Die rotierten Punkte $R^{(1)}B = (p', q', r', s')$ und $R^{(2)}B = (p'', q'', r'', s'')$ sind in Abbildung 15 gezeigt. Dies sind übrigens nicht die einzigen optimalen Lösungen, beispielsweise erhalten wir mit $R^{(3)} := I$ ebenfalls $\text{rms} = 2$.

Soviel ist sicher: Der Algorithmus findet in jedem Fall eine optimale Lösung. Hinreichend für die Eindeutigkeit ist, dass die Singulärwerte von AB^T alle verschieden sind. In der Praxis kann man wohl davon ausgehen, dass dies zumeist der Fall sein wird.

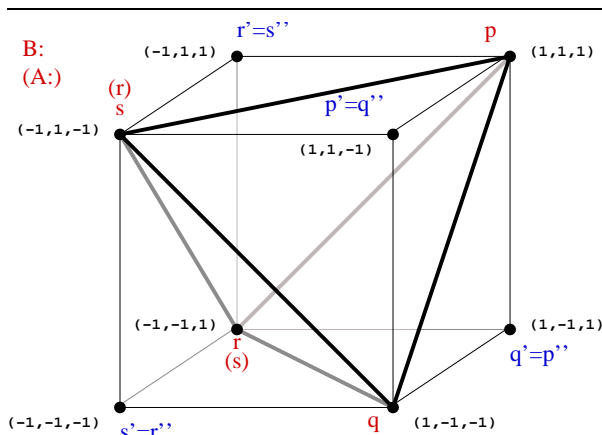


Abbildung 15: Beispiel

3.2 Optimale Überlagerung durch Enumeration

Wir behandeln nun eine Möglichkeit, wie man auch das optimale Matching bestimmen kann. Dabei optimiert man nach zwei Parametern: Anzahl der überlagerten Atome und rms (nach einer optimalen starren Bewegung) der überlagerten Teile. Um aus diesen beiden Parametern in eine einzige Bewertungsfunktion zusammenzufassen, kann man beispielsweise die Funktion

$$\text{score} = (\% \text{ überlagerte Atome}) \cdot e^{-\text{rms}}$$

verwenden, wobei „% überlagerte Atome“ gleich der Anzahl der überlagerten Atompaare, geteilt durch die Anzahl der Atome in der kleineren der beiden Mengen ist.

Die Idee der exakten Enumeration ist (wie immer) sehr einfach:

- Für jedes Atom aus A , probiere der Reihe nach jedes noch ungematchte Atom aus B aus, sowie die Möglichkeit, es ungematcht zu lassen.
- Anschließend rekursiver Abstieg.
- Wenn alle Atome aus A entschieden sind (Terminalfall), bewerte dieses Matching mit dem exakten Algorithmus für ein gegebenes Matching.

Um eine Vorstellung von der Laufzeit zu gewinnen, schätzen die Größe des Enumerationsbaumes ab. Die Anzahl der perfekten Matchings zwischen zwei Mengen der Größe n ist $n!$. Zusätzlich können die vom Matching überdeckten Mengen variieren. Die Anzahl der Matchings für Atommengen

$A \in \mathbb{R}^{3 \times m_A}$ und $B \in \mathbb{R}^{3 \times m_B}$ liegt also zwischen $m!$ und $2^{m_A+m_B} m!$, wobei $m := \min\{m_A, m_B\}$. Nach der Stirlingformel ist $m! \sim \sqrt{2\pi} \sqrt{m} \left(\frac{m}{e}\right)^m = 2^{m \log m + O(m)}$.

Man kann aber schon mit einer trivialen Ungleichung viele Äste des Enumerationsbaumes abschneiden. (Damit kommt man dann bis ca. 17 Atome. ;-)

Proposition 20. Seien $A \in \mathbb{R}^{3 \times m_A}$ und $B \in \mathbb{R}^{3 \times m_B}$ und $x \mapsto RX + t$ eine optimale starre Bewegung für ein Matching $\pi : [k] \rightarrow m_B$, d.h. der rms

$$\left(\frac{1}{k} \sum_{i=1}^k |A_i - RB_{\pi(i)} - t|^2 \right)^{1/2}$$

ist minimal. Dann hat jede starre Bewegung für ein Matching $\pi' : [k'] \rightarrow [m_B]$, das π erweitert, einen

$$\text{rms} \geq \left(\frac{1}{k'} \sum_{i=1}^k |A_i - RB_{\pi(i)} - t|^2 \right)^{1/2}.$$

Beweis. Sei $\pi' : [k'] \rightarrow [m_B]$ injektiv mit $k' \geq k$, $\pi'(i) = \pi(i)$ für $i \leq k$ und $x \mapsto R'x + t'$ die optimale starre Bewegung für π' . Dann gilt

$$\begin{aligned} & \left(\frac{1}{k'} \sum_{i=1}^{k'} |A_i - R'B_{\pi'(i)} - t'|^2 \right)^{1/2} \\ & \geq \left(\frac{1}{k'} \sum_{i=1}^k |A_i - R'B_{\pi(i)} - t'|^2 \right)^{1/2} \\ & \geq \left(\frac{1}{k'} \sum_{i=1}^k |A_i - RB_{\pi(i)} - t|^2 \right)^{1/2}. \end{aligned}$$

□

3.3 Geometrisches Hashing

Das geometrische Hashing ist ursprünglich in der künstlichen Intelligenz entwickelte Methode zur Objekterkennung, die sich auch jedoch vorzüglich zum Überlagern von Atommengen eignet. Die Methode ist sehr allgemein und flexibel einsetzbar und in vielen Anwendungen der „state of the art“. Die Laufzeit ist polynomiell (von niedrigen Grad) und lässt sich mit problemspezifischen Zusatzüberlegungen stark reduzieren. Auch mit verrauschten Daten kommt die Methode gut zurecht.

Die Objekte werden durch *Features* beschrieben. Dies sind im einfachsten Fall Punkte (z.B. Atommittelpunkte), können aber auch andere „lokale“ geometrische Gegebenheiten sein, z.B. Segmente, Ecken oder C_α -Atome aus Proteinen mit „angedeuteten“ Resten (mehr dazu später). Die Dimension ist beliebig (aber konstant).

Als *Transformationen* lassen wir, wie gehabt, alle starren Bewegungen zu, es sind im Prinzip aber beliebige Kombinationen von Translation, Rotation, Skalierung und Scherung einsetzbar.

In diesem Zusammenhang spricht man gerne von *Modellen* und *Szenen*. Die Modelle kennt man im voraus (aus einer Datenbank), wohingegen die Szene in der Regel neu ist (z.B. gerade von einer Kamera erfasst wurde).

Es wird ein *partielles Matching* gefunden, das heißt, es können Teile fehlen (z.B. in der Szene verdeckt sein, oder im Modell nicht vorkommen).

Das geometric hashing verwendet eine Hash-Tabelle, um die Daten der Models in einem Preprocessing zu indizieren. Dabei stellt sich das Problem, welches Koordinatensystem verwendet werden soll. Absolute Koordinaten machen keinen Sinn, da wir es bei der Wiedererkennung mit transformierten Objekten zu tun haben. Abhilfe schaffen hier relative Koordinaten, deren Bezugssystem eine *Basis* ist, die aus einer genügend großen Anzahl von Features extrahiert wird (z.B. 3 Punkte im \mathbb{R}^3).

Preprocessing

Abbildung 16 enthält den Pseudocode für das Preprocessing. Die Hashtabelle H enthält „Kollisionslisten“.

Für jedes Modell A :

1. Extrahiere die Features von A , seien dies $A = (A_1, \dots, A_n)$.
2. Für jede Basis α von Features in A :
Für alle übrigen Features $A_i \notin \alpha, i \in [n]$:
 - (a) Stelle das Feature A_i in Koordinaten c bezüglich des durch α definierten Koordinatensystems dar.
 - (b) Quantisiere c in einen diskreten Wert \tilde{c} .
Trage die Information (A, α) in die Hashzelle $H(\tilde{c})$ ein.

Abbildung 16: Preprocessing für das geometrische Hashing

Um das relative Koordinatensystem für eine Basis (x, y, z) zu bestimmen, kann man wie folgt vorgehen. Der Ursprung wird in den Schwerpunkt $(x + y + z)/3$ gelegt. Die ersten zwei Basisvektoren sind $b_1 := \frac{y-x}{|y-x|}$ und $b_2 := \frac{z-x - (z-x, b_1)b_1}{|z-x - (z-x, b_1)b_1|}$. Der dritte Basisvektor b_3 steht senkrecht auf b_1, b_2 , und zwar so dass $\det(b_1, b_2, b_3) = 1$.

Bei der Quantisierung ist ein trade-off zu berücksichtigen. Je feiner man die diskreten Stufen wählt, um so genauer ist die gefundene Anfangsüberlagerung zwischen Szene und erkanntem Objekt, und die Selektivität steigt. Wenn man gröber quantisiert, wird die Anfangsüberlagerung ungenauer, dafür erkennt man mehr Treffer. Erwünscht ist außerdem eine gleichmäßige Füllung der Hashtabelle. Daher wird man in der Abbildung $c \mapsto \tilde{c}$ eventuell etwas mehr tun als nur zu runden, z.B. den mittleren Bereich vergrößern, wenn sich dort die Einträge häufen.

Die Laufzeit des Preprocessings ist $O(mn^{d+1})$ für m Modelle mit je n Features, wobei jede Basis aus d Features besteht.

Erkennung

Abbildung 17 enthält den Pseudocode für die Erkennungsphase (recognition).

Wenn in Schritt 2.(b) ein Eintrag (A, α) aus der Hashzelle $H(\tilde{c})$ eine Stimme erhält, so muss es ein Feature A_j aus dem Modell A geben, welches bezüglich α ähnliche relative Koordinaten hat wie das Feature B_i bezüglich β . Wenn also ein Paar (A, α) eine hohe Anzahl von Stimmen erhält, so erhalten wir aus den Zuordnungen $A_j \leftrightarrow B_i$ ein „gutes“ Matching, und im Fall der starren Bewegungen haben wir gesehen, wie man für ein gegebenes Matching die optimale Transformation T berechnen kann. Nichts hindert uns, das Matching noch weiter (heuristisch) nachzuoptimieren, d.h. Kanten hinzuzufügen oder herauszunehmen.

Die Laufzeit von Schritt 2 der Erkennungsphase ist $O(hn^{d+1})$, wobei h die durchschnittliche Füllungsichte der betrachteten Hashzellen ist. In günstigen Fällen ist $h = O(1)$, jedoch kann schlimmstenfalls $h = mn^{d+1}$ sein. Hinzu kommt noch die Laufzeit für die Bearbeitung der gefundenen Treffer.

Modifikationen des Hashings

- *Weighted Voting*. Das Gewicht einer Stimme ist umgekehrt zur Anzahl der Einträge der betrachteten Hashzelle, oder sogar 0, falls die Hashzelle zu voll ist. Die Intuition dahinter ist,

Gegeben: Szene B .

1. Extrahiere die Features von B , seien dies $B = (B_1, \dots, B_n)$.
2. Wähle eine Basis β von Features in B :
Für alle übrigen Features $B_i \notin \beta, i \in [n]$:
 - (a) Stelle das Feature B_i in Koordinaten c bezüglich des durch β definierten Koordinatensystems dar.
 - (b) Quantisiere c in einen diskreten Wert \tilde{c} . Jeder Eintrag $(A, \alpha) \in H(\tilde{c})$ erhält eine Stimme.
 - (c) Zähle die Stimmen aus für alle Paare von Modellen und Basen (Histogramm). Paare mit genügend vielen Stimmen entsprechen potentiellen Treffern.
 - (d) Für jeden potentiellen Treffer (A, α) :
Bestimme die optimale Transformation T für die gematchten Features von A und B und bewerte die gefundene Lösung.
3. Falls keine gute Lösung gefunden wurde, wiederhole Schritt 2.

Abbildung 17: Erkennungsphase des geometrischen Hashings

dass beim Auszählen die Stimmen vollen Hashzellen weniger „Information“ beitragen als solche aus fast leeren, aber viel Zeit kosten.

- **KD-Bäume.** Wenn das Rauschen in den Daten stark richtungsabhängig ist (wie dies beispielsweise bei 3D-Daten, die aus Stereobildern gewonnen wurden, der Fall ist), ist es sinnvoll (und möglich), die Fehlerschranken bis in die relativen Koordinaten zu propagieren. Dann wählt man die Quantisierung $c \mapsto \tilde{c}$ etwas feiner und wertet in der Erkennungsphase zusätzlich auch Stimmen aus benachbarten Hashzellen mit aus. Alternativ kann man aber auch ganz auf die Quantisierung verzichten und die Hashtabelle durch einen **KD-Baum** ersetzen. Diese Datenstruktur unterstützt „orthogonal range queries“, d.h. Abfragen nach allen Punkten in einem gegebenen kartesischen Produkt von Intervallen (z.B. achsenparalleler Quader im \mathbb{R}^3).

Verbesserung der gefundenen Anfangsüberlagerung mittels bipartitem Matching Wenn das geometrische Hashing einen Treffer findet, so gibt es eine Basis α in A und eine Basis β in B , so dass das zugehörige Matching

$$M := \{ (A_i, B_j) \mid \tilde{c}(A_i, \alpha) = \tilde{c}(B_j, \beta) \}$$

„viele“ Kante enthält. Mit dem Algorithmus zur optimalen Überlagerung bei gegebenem Matching erhält man daraus eine optimale starre Bewegung T für das Matching M . Es kann aber sein, dass uns dabei aufgrund des Hashings einige Zuordnungen entgangen sind.

Zur Nachoptimierung des Matchings kann man den bipartiten Graphen mit Knotenmengen A und B und den Kanten $E = \{ (A_i, B_j) \mid |A_i - T(B_j)| \leq \delta \}$ betrachten. Der Parameter δ ist geeignet zu wählen, z.B. 4 Å. Jede Kante erhält ein Gewicht, das proportional zum Abstand ihrer Endpunkte ist. Nun wird ein größtes Matching M' bestimmt, welches minimales Gewicht hat. (Dies ist in polynomieller Zeit möglich, z.B. $O(\sqrt{nm}^2)$ für bipartite Graphen mit n Knoten und m Kanten, vgl. GA1.) Anschließend wird $M \leftarrow M'$ gesetzt und der Vorgang iteriert, bis Konvergenz eingetreten oder eine vorgegebene Maximalanzahl von Iterationen erreicht ist. Der Parameter δ dient vor allem zur Kontrolle der Laufzeit, beeinflusst aber auch die Größe des Matchings. Es gibt aber auch Matching-Algorithmen, die inkrementell schwerste Matchings mit 1, 2, ... Kanten bestimmen.

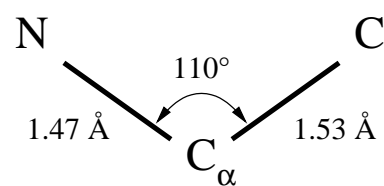


Abbildung 18: Jedes C_α -Atom definiert zusammen mit den benachbarten C- und N-Atomen eine Basis.

Verwendung von C_α -Atomen als Basen Problematisch bei Geometric Hashing Methoden ist oft der Zeitaufwand für die Enumeration aller Basen. In Proteinen gibt es in der Hauptkette die C_α -Atome. Die benachbarten N- und C-Atome sind immer in der gleichen Weise angeordnet (siehe Abbildung 18), und zusammen mit dem C_α -Atom ergibt sich ein relatives Koordinatensystem. Die Laufzeit reduziert sich damit auf $O(mn^2)$ für das Preprocessing. (Sie war $O(mn^4)$ bei Verwendung aller Tripel.)

Es erweist sich als vorteilhaft, in der Erkennungsphase grundsätzlich über alle C_α -Atome als Basen zu enumerieren. Die gefundenen Transformationen werden geclustert. Aus den Clustern werden „Mittelwerte“ berechnet.

Eine weitere Effizienzsteigerung ist möglich, wenn man berücksichtigt, dass die C_α -Atome auf der Hauptkette in linearer Folge angeordnet sind. Dadurch kann man viele potentielle Matchings im voraus ausschließen.

Generell kann man sagen, dass die gegenwärtigen Programmwerkzeuge jeweils einen Kompromiss zwischen Allgemeinheit und Effizienz darstellen, wobei eine Fülle von weiteren, größtenteils heuristischen und problemspezifischen Überlegungen mit einbezogen werden.

4 Phylogenetische Bäume

Ultrametrik, ...

Ende VL 2003-01-27

Konstruktion in $O(n^2)$, ...

Ende VL 2003-02-03

Literatur

[Aho] Alfred V. Aho: *Algorithms for Finding Patterns in Strings*, Chapter 5 in: *Handbook of Theoretical Computer Science*, ed. by J. van Leeuwen, Elsevier Science Publishers, Amsterdam, 1990. [2.1.4]

[Aku] Tatsuya Akutsu: *Protein Structure Alignment Using Dynamic Programming and Iterative Improvement*, *IEICE Transactions on Information and Systems*, E79D(12):1629–1636, 1996. [3.3]

[Fis] Gerd Fischer: *Lineare Algebra*, Vieweg Verlag, Braunschweig, 1986. [3.1]

[Gus] Dan Gusfield: *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*, Cambridge University Press, Cambridge, 1997. [2, 2.1.3]

[Heu] Volker Heun et al.: *Algorithmische Bioinformatik I/II*, Kapitel 7 aus dem Vorlesungsskript, Lehrstuhl Effiziente Algorithmen, Informatik TU München, Ver-

sion vom 17.12.2002. [4] <http://www.mayr.informatik.tu-muenchen.de/>

[KMUW] D. Kandel, Y. Matias, R. Unger, P. Winkler: *Shuffling Biological Sequences*, *Discrete Applied Mathematics* 71, p. 171–185, 1996. [2.2]
Siehe auch die Fußnote 3, Seite 10 in Abschnitt 2.2.2 dieses Skripts.

[PA] X. Pennec, N. Ayache: *A geometric algorithm to find small but highly similar 3D substructures in proteins*, *Bioinformatics*, 14(4), 516–522, 1998. [3.3]

[RW] I. Rigoutsos, H.J. Wolfson: *Geometric Hashing: An Overview*, *IEEE Computational Science and Engineering*, 4(4), 10–21, 1997. [3.3]
<http://www.math.tau.ac.il/~wolfson/on-line-articles.html>

[Sch] Georg Schnittger: *Algorithmen der Bioinformatik*, Skript zur VL im WS 2000/2001, vorläufige Fassung 14. Februar 2001. [3.1]

[Ste] Graham A. Stephen: *String Searching Algorithms*, (Lecture Notes Series on Computing, Vol. 3), World Scientific Publishing, Singapore, 1994. [2.1.1, 2.1.2]

[Ume] Shinji Umeyama: *Least-Squares Estimation of Transformation Parameters Between Two Point Patterns*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 4, April 1991. [3.1]
Siehe auch die Anmerkungen auf Seite 17 in Abschnitt 3.1 dieses Skripts.

In [] sind jeweils die Kapitel des Skripts angegeben.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Daten: Womit haben wir es zu tun? .	1
1.2	Aufgaben: Was ist zu tun?	1
1.3	Methoden: Wie tun wir es?	1
2	Strings	1
2.1	String Matching	2
2.1.1	Brute Force	2

2.1.2	Knuth-Morris-Pratt	2
2.1.3	BLAST	4
2.1.4	Aho-Corasick: Multiple String Matching	5
2.2	Sequenzen mischen	6
2.2.1	Vertauschungsalgorithmus zum Mischen von Sequenzen	7
2.2.2	Exaktes Mischen von Sequen- zen mittels Eulertouren . . .	7
3	Proteinstruktur	12
3.1	Optimale Überlagerung bei gegeb- nem Matching	13
3.2	Optimale Überlagerung durch Enu- meration	18
3.3	Geometrisches Hashing	18
4	Phylogenetische Bäume	21

.....

Hinweise und Korrekturen bitte per email an
groepl@informatik.hu-berlin.de !