Einführung in die Angewandte Bioinformatik: Algorithmen und Komplexität; Multiples Alignment 04.06.2009 und 18.06.2009

Prof. Dr. Sven Rahmann

Zwischenspiel: Algorithmik

Bisher „nebenbei“:
Vorstellung von
- Problemen (es muss etwas berechnet werden)
- Methoden (wie macht man das im Prinzip?) - bisher kaum diskutiert
- Programmen (wie macht man das konkret?)

Jetzt:
Hintergrundwissen dazu.

Problem – Algorithmus – Programm

Typisches Vorgehen in der Informatik:

1. Spezifikation eines **Problems**
2. „Erfinden“ eines **Algorithmus** (Methode), der das Problem löst
3. Umsetzung („Implementierung“) des Algorithmus als **Programm** in einer Programmiersprache (z.B. C, C++, Java, Perl, Python, R, ...)

Zu jeden Problem gibt es i.d.R. mehrere Algorithmen,
die unterschiedlich gut (schnell oder elegant oder einfach) sein können.

Zu jedem Algorithmus gibt es i.d.R. mehrere mögliche Implementierungen.

Problem – Algorithmus – Programm: Details

1. Spezifikation des **Problems** durch:
- Was ist gegeben?
- Was ist gesucht?
- Welche Bedingungen / Regeln sind einzuhalten?

2. „Erfinden“ eines **Algorithmus**, der das Problem löst.
- Algorithmus gibt endliche Abfolge von auszuführenden Schritten an.
- Beweis (math.), dass die angegebene Schrittfolge das Problem löst.
- Aussagen zur Dauer („Laufzeit“) bis zur Lösung.

3. Umsetzung („Implementierung“) des Algorithmus als **Programm**
in einer bestimmten formalen Sprache (z.B. C, C++, Java, Perl, Python, R, ...)

Zum Algorithmus-Begriff

Schwierig, den Begriff Algorithmus exakt zu formalisieren.
Berechenbarkeitsmodelle: Turing-Maschinen, Registermaschinen, Lambda-Kalkül...

Wichtig sind folgende Eigenschaften eines Rechenverfahrens:

1. mit einem endlichen Text eindeutig beschreibbar (Finitheit).
2. jeder Schritt ausführbar (Ausführbarkeit)
3. stets nur endlich viel Speicher (dynamische Finitheit)
4. nur endlich viele Schritte (Terminierung)
5. gleiches Ergebnis bei gleichen Voraussetzungen (Determiniertheit).
6. nächster Schritt ist stets eindeutig definiert (Determinismus).

Algorithmus leitet sich aus dem Namen
Abu Abdallah Muhammad ibn Musa al-Chwarizmi (أبو عبد الله محمد بن موسى الخوارزمي)
ab; vermutlich iranischer Mathematiker (* um 780; † zwischen 835 und 850).

Problem – Algorithmus – Programm: Beispiel ggT (größter gemeinsamer Teiler)

1. Spezifikation des **Problems:**
- Was ist gegeben? zwei positive ganze Zahlen x, y.
- Was ist gesucht? ihr größter gemeinsamer Teiler.
- Welche Bedingungen? Was ist (gemeinsamer) Teiler; was heißt größer?

2. „Erfinden“ eines **Algorithmus**, der das Problem löst:
- Sei m das Minimum von x und y
- Sei $g := 1$
- Für jede Zahl z von 2 bis m aufsteigend nacheinander:
  Wenn x/z und y/z beide ganzzahlig sind, setze $g := z$
- Gib g aus

Der Algorithmus ist „offensichtlich“ korrekt. (Wirklich? Warum?)
Aber er ist nicht sehr schnell: Man testet $m = \min(x,y)$ Zahlen.
Besser: Euklidischer Algorithmus (Mathe-Vorlesung).

Problem – Algorithmus – Programm: Beispiel ggT (größter gemeinsamer Teiler)

3a. Implementierung in Python:

```python
def ggt(x,y):
    """groesster gemeinsamer Teiler von x und y."""
    m = min(x,y)
    g = 1
    for z in range(2,m+1):
        if (x//z)*z == x and (y//z)*z == y:
            g = z
    return g
```

3b. Implementierung in R:

```r
ggt = function(x,y) {
  m = min(x,y)
  g = 1
  for (z in 2:m) {
    if (floor(x/z)*z==x && floor(y/z)*z==y) {
      g=z
    }
  }
  g
}
```

Problem – Algorithmus – Programm: Beispiel Türme von Hanoi

23.05.2009: Chemie-Doktorand Nino aus Mainz bei „Schlag den Raab“
Video unter http://www.schlag-den-raab.de/videos/videoplayer/64054/

Diskussion zu:

1. Problemspezifikation

2. Algorithmus

3. Implementierung (lassen wir weg)

Problem – Algorithmus – Programm: Beispiel globales paarweises Sequenzalignment

1. Spezifikation des Problems:

Gegeben:
- zwei Sequenzen über einem Alphabet (i.d.R. Nukleotide oder Aminosäuren),
- eine Scorematrix zu diesem Alphabet,
- Gapkosten (welcher Art? linear, affin, beliebig?)

Gesucht:
Score eines optimalen globalen Alignments; optimales Alignment dazu.

Regeln:
- Alignment heißt: (siehe Definition von paarweisem Alignment)
- optimal heißt: höchstmöglicher Score
- Score eines Alignments berechnet sich als Summer über alle Spalten
- Score einer Spalte ergibt sich aus Scorematrix oder Gapkosten

Problem – Algorithmus – Programm: Beispiel globales paarweises Sequenzalignment

2a. Ein Algorithmus („naiv“):

- Für alle möglichen Alignments der Sequenzen:
  - Berechne Score des Alignments.
  - Wenn Score besser als bisher bester Score, merke Score und Alignment.
- Gib besten gemerkten Score und Alignment aus.

Problem dabei:
Zu zwei Sequenzen der Länge n gibt ca. 4^n mögliche Alignments.
(Die meisten haben einen niedrigen Score und sind irrelevant,
werden aber dabei trotzdem überprüft.)

Problem – Algorithmus – Programm: Beispiel globales paarweises Sequenzalignment

2b. Entwicklung eines besseren Algorithmus:

Beobachtung / Idee: Jedes Alignment endet auf eine von drei Arten:

- Match/Mismatch (zwei Symbole aligniert).
- Gap in der ersten Sequenz.
- Gap in der zweiten Sequenz.

```
...x    ...-    ...x
...y    ...y    ...-
```

Will man s, t optimal alignieren und kennt die drei optimalen Alignments von

- s ohne letztes Symbol mit t ohne letztes Symbol (Score S_0),
- s mit und t ohne letztes Symbol (Score S_1),
- s ohne letztes Symbol mit und t (Score S_2),

ergeben sich für den Alignment-Score von s, t die drei Möglichkeiten:

- S_0 + Score(letztes Symbol von s, letztes Symbol von t),
- S_1 – Gapkosten,
- S_2 – Gapkosten.

Behauptung: Optimales Alignment von s, t: beste dieser drei Möglichkeiten.

Problem – Algorithmus – Programm: Beispiel globales paarweises Sequenzalignment

Behauptung: Optimales Alignment von s, t: beste der drei Möglichkeiten.

Beweis:

(1) Die drei Möglichkeiten sind zulässige Alignments:
Das beste Alignment ist nicht schlechter als die beste der drei Möglichkeiten.

(2) Es kann kein besseres Alignment als die beste der drei Möglichkeiten geben.
Angenommen, doch: Entferne letzte Spalte.
Dann erhalten wir ein besseres Alignment der entsprechenden Präfixe.
Wir hatten aber deren Optimalität vorausgesetzt.
(Prinzip des Induktionsbeweises)

```
...x    ...-    ...x
...y    ...y    ...-
```

Definition dazu:

Ein **Präfix** einer Sequenz $s=s_1..s_n$ ist eine Sequenz $s_1..s_i$ mit $0 <= i <= n$.
Für $i=0$ spricht man vom **leeren Präfix**.

Problem – Algorithmus – Programm: Beispiel globales paarweises Sequenzalignment

Vorgehen

Gegeben: Sequenzen s, t; Scorematrix Score(.,.), Gapkosten g

Idee: Tabelliere
Opt(i,j) := optimaler Score der Präfixe $s_1..s_i$ und $t_1..t_j$ (für $i = 0 .. |s|$, $j = 0.. |t|$).

- Opt(i,0) = $-g\,i$ für $i >= 0$.
- Opt(0,j) = $-g\,j$ für $j >= 0$.

Wenn $i>0$ und $j>0$, nutzen wir das eben Festgestellte:

- Opt(i,j) = max { Opt(i-1, j-1) + Score(s_i, t_j), Opt(i-1, j) – g, Opt(i, j-1) – g }.

Dies ist (noch) kein Algorithmus, da wir nicht angegeben haben,
in welcher Reihenfolge die Einträge von Opt berechnet werden.
Wir wissen auch noch nicht, wie wir das Alignment erhalten (hier nur: Score).

Problem – Algorithmus – Programm: Beispiel globales paarweises Sequenzalignment

Algorithmus (Needleman-Wunsch)

Berechne Opt(0,0)=0
für $j = 1 .. |t|$:
    Berechne Opt(0,j) = $-g\,j$
für $i = 1 .. |s|$:
    Berechne Opt(i,0) = $-g\,i$
    für $j = 1 .. |t|$:
        Berechne Opt(i,j) = max {Opt(i-1, j-1) + Score(s_i, t_j), Opt(i-1, j) – g, Opt(i, j-1) – g }
Gib Opt(|s|,|t|) zurück # (optimaler Alignmentscore von s und t)

Wie kommt man vom optimalen Alignment-Score an das optimale Alignment?
Man muss zurückverfolgen, welche Alternativen bei max { } gewählt wurden.

Prinzip der „Dynamischen Programmierung“

Dynamische Programmierung (DP)
(hat nichts mit Programmierung eines Computers zu tun.)
(Richard Bellman: Dynamic Programming. Princeton University Press, 1957.)

Lösung des Gesamtproblems wird durch
„Zusammensetzen“ von Lösungen kleinerer gleichartiger Probleme berechnet.

Teillösungen werden in Tabelle zwischengespeichert,
so dass sie nicht ständig neu berechnet werden müssen.

Beispiel Needleman-Wunsch-Algorithmus:
Lösung des Problems für Sequenzen der Längen (m,n)
ergibt sich aus Lösung für Sequenzen der Längen $(m-1,n-1)$, $(m,n-1)$ und $(m-1,n)$.
Tabelle Opt speichert optimalen Score-Wert für jede Längenkombination (i,j).

Laufzeit von Algorithmen: O-Notation
Korrektheit von Algorithmen („exakte Algorithmen“)

Laufzeit des Needleman-Wunsch-Algorithmus

Es werden $(|s|+1) * (|t|+1)$ Werte berechnet.
Jede Berechnung braucht konstante Zeit (Maximierung über 3 Werte)
Insgesamt: $O(st)$ Zeit.

O-Notation: Angabe von Funktionen unter Vernachlässigung von
- konstanten Faktoren
- „kleineren“ (d.h. langsamer wachsenden) Termen

Definition [exakte Lösung in Zeit „groß-O von $f(n)$“]
Algorithmus **löst** ein gegebenes Problem der Größe n **exakt** in **Zeit $O(f(n))$**, wenn
- garantiert die korrekte Lösung berechnet wird („Korrektheit“)
- dies immer höchstens $c * f(n)$ Rechenschritte benötigt (c konstanter Faktor).

Beispiel zur O-Notation

Angenommen, es gibt zwei Sortieralgorithmen, die jeweils
- $n^2/2 - n \log n + n/3$ Vergleiche
- $3n \log n$ Vergleiche

machen, um n Zahlen zu sortieren.

Es ist $n^2/2 - n \log n + n/3$ in $O(n^2)$, oder auch in $O(n^{17})$.
Es ist $3n \log n$ in $O(n \log n)$, oder auch in $O(n^2)$, aber nicht in $O(n)$.

Für genügend große n ist der zweite Algorithmus immer schneller.
Für kleine n (bis wohin?) ist der erste Algorithmus schneller.
O-Notation betrachtet nur Verhalten für große n.

Heuristiken

Korrekte („exakte“) Algorithmen garantieren optimale Lösung.
Nicht immer existieren hinreichend schnelle korrekte Algorithmen!
Beispiel gleich: multiples Sequenzalignment

=> Verwendung von **Heuristiken** (gr. *heuriskein*, „(auf-)finden“, „entdecken“)
(Kunst, mit begrenztem Wissen und wenig Zeit zu guten Lösungen zu kommen.)

(Qualitäts-)Heuristik:
- garantierte (schnelle) Laufzeit.
- gefundene Lösung nicht notwendig optimal.

Laufzeit-Heuristik:
- löst das Problem in jedem Fall exakt.
- versucht, durch „Abkürzungen“ die Lösung möglichst schnell zu finden.

Beispiel zu Heuristiken: Smith-Waterman vs. BLAST

Der **Smith-Waterman-Algorithmus** ('water' bei EMBOSS)
löst Problem des lokalen Sequenz-Alignments von s,t **exakt in $O(|s|\,|t|)$** Zeit.
(Funktionsweise: DP ähnlich Needleman-Wunsch)
Ist t eine große Datenbank -> mehrere Minuten bis Stunden; zu langsam.

BLAST ist eine **Heuristik** für dasselbe Problem.
Bei unähnlichen Sequenzen wird i.d.R. nicht das optimale Alignment gefunden.
Laufzeit nur etwa $O(|s| + |t|)$. In der Praxis sehr gut.

Für Datenbanksuche wird ausschließlich BLAST verwendet.
Für Alignment einzelner Sequenzen wird Smith-Waterman verwendet.

Problemkomplexität; Komplexitätsklasse P

Komplexität eines Problems :=
Laufzeit des schnellsten exakten Algorithmus, der das Problem löst.

Beispiel: Maximum von n Zahlen: offensichtlich $O(n)$ Zeit („Linearzeit“).
Schneller kann es nicht gehen (jede Zahl muss betrachtet werden).
Problemkomplexität ist $O(n)$.
Beispiel: Sortieren von n Zahlen: z.B. in $O(n^2)$ Zeit, aber auch in $O(n \log n)$ Zeit.
Man lernt in der Informatik, dass es nicht schneller gehen kann.
Problemkomplexität ist $O(n \log n)$.

Komplexitätsklasse P („in polynomieller Zeit lösbar“)
Gesamtheit aller Probleme, für die gilt:
Es gibt einen Algorithmus, der das Problem in einer Zeit exakt löst,
die durch ein **Polynom** in der Eingabegröße beschränkt ist.
z.B. Maximum, Sortieren, Sequenzalignment.

Komplexitätsklasse NP

Es gibt Probleme,
- für die kein Polynomialzeit-Algorithmus zur Lösung bekannt ist,
- für die aber in Polynomialzeit überprüft weden kann,
  ob eine „vom Himmel gefallene“
  oder durch ein (hypothetisches) allwissendes Orakel gegebene
  Lösung das zugehörige Entscheidungsproblem löst.

=> **Klasse NP**
- „nichtdeterministisch in Polynomialzeit lösbar“
- durch glückliches Raten in Polynomialzeit lösbar
- in Polynomialzeit überprüfbar

Jedes Problem in P ist auch in NP.

Beispiel: Traveling Salesperson ist in NP

Traveling Salesperson (TSP, Handlungsreisender)
Gegeben: *n* Städte und paarweise Entfernungen.
Gesucht: Rundreise durch alle Städte mit der kürzesten Gesamtentfernung.

(*n*-1)! = 1 * 2 * 3 * ... * (*n*-1) Möglichkeiten, eine Tour zu planen.
Kein Algorithmus bekannt, der in Polynomialzeit garantiert die kürzeste Tour findet.

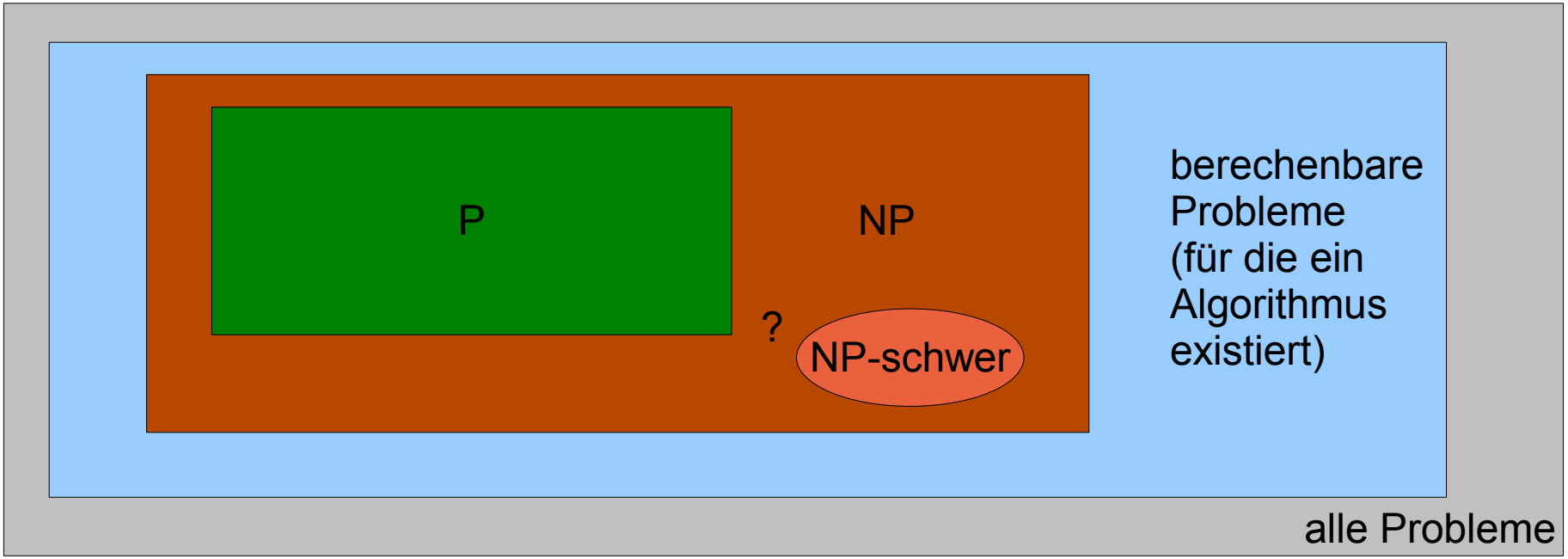
Entscheidungsproblem: Existiert eine Rundreise mit Gesamtentfernung <= *k* ?
Orakel: Liefert eine solche Rundreise, wenn sie existiert, sonst irgendeine.
Wir: prüfen, ob die Gesamtkosten der Reise tatsächlich <= *k* sind.
Prüfung besteht aus Addition von *n* Entfernungen; das geht in O(*n*) Zeit (in P).

Also: TSP ist in NP.
Auch in P? Unbekannt!

P=NP – Frage und NP-schwere Probleme

P ist Teilklasse von NP. Gibt es Probleme, die in NP, aber nicht in P sind?
Große ungelöste Frage der theoretischen Informatik („Jahrtausendproblem“).

Manche NP-Probleme (z.B. TSP) sind mindestens so schwer wie alle NP-Probleme.
=> „NP-vollständig“ (Entscheidungsprobl.) bzw. „NP-schwer“ (Optimierungsprobl.).
Wenn man für eins davon einen polynomiellen Algorithmus findet, ist P = NP.



Zurück zu Alignments

Multiple Alignments
und ihre Berechnung

Bisher: Paarweise Alignments

Optimales Alignment:

- Alignment mit höchstem Score unter allen Alignments
- arbeitet die Ähnlichkeiten zwischen zwei Sequenzen heraus
- Spaltentypen: Match/Mismatch, Insertion, Deletion mit Scores bzw. Kosten

Wichtig für:

- quantitative Bestimmung von Sequenzähnlichkeit
- Übertragen von Informationen (Struktur/Funktion) zwischen ähnlichen Seq.

Aber:

Proteinfamilien oder Domänen bestehen aus mehr als 2 Sequenzen

Erinnerung: Proteindomänen und Proteinfamilien (Pfam)

Domänen

sind wiederkehrende modulare Bausteine von Proteinen.
Durch verschiedene Kombinationen von Domänen
entstehen Proteine mit unterschiedlichen Eigenschaften.
Ziel: alle existierenden Domänen katalogisieren, analysieren

Proteinfamilien

Eine Domäne oder eine bestimmte Kombination von Domänen
kann eine bestimmte Familie von Proteinen charakterisieren.

Datenbanken zu Domänen

Pfam:	http://pfam.sanger.ac.uk/	protein families
SMART:	http://smart.embl-heidelberg.de/	simple modular architecture research tool

Modellierung von Proteindomänen

Wie kann man eine Domäne beschreiben?

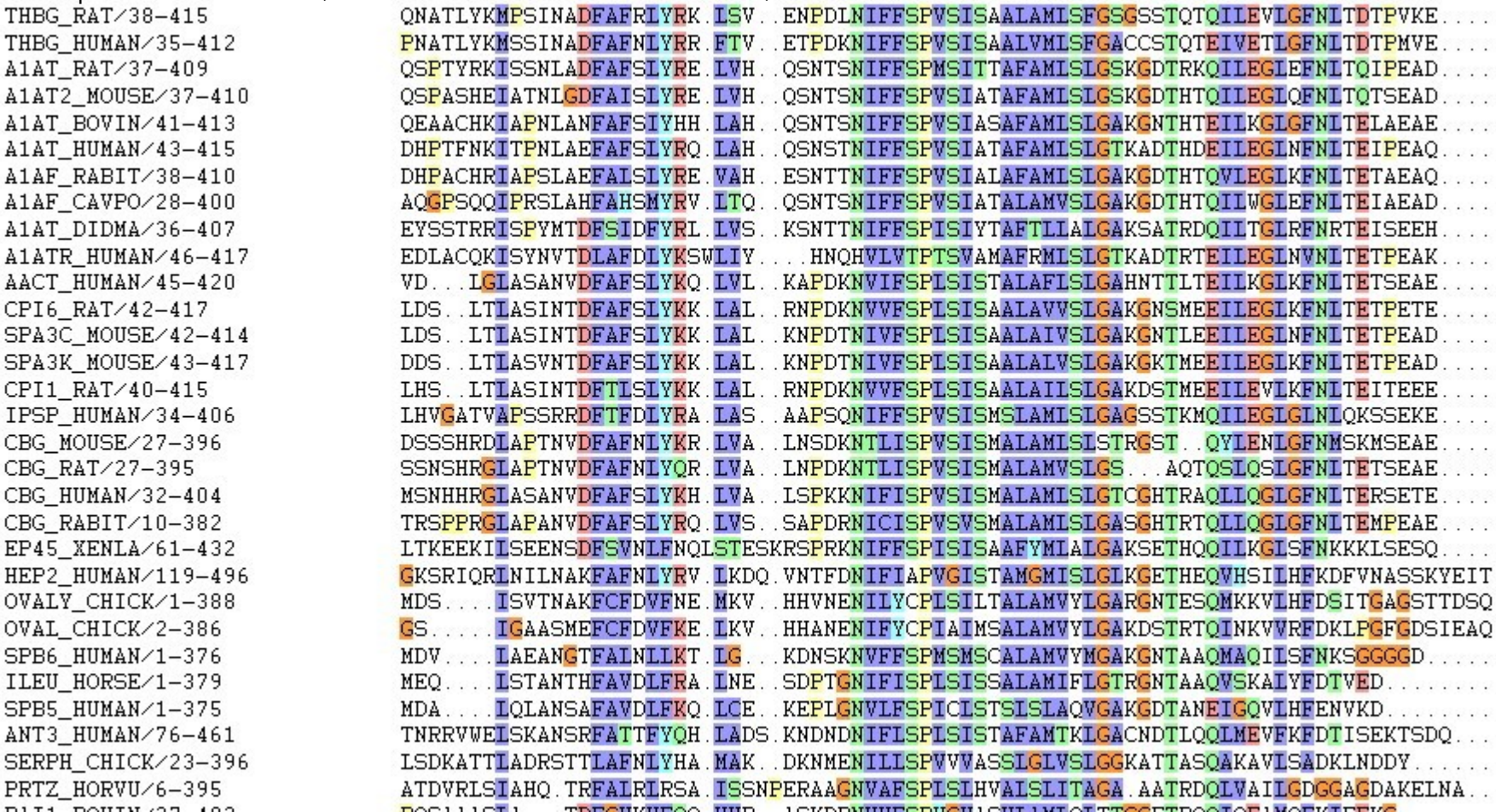
- Aminosäuresequenz (Konsensus + Variationsmöglichkeiten)
  Sequenz angeben, evtl. mehrere Symbole pro Position
  (nicht sehr nützlich wg. Variationen)

- statistisches Modell (Hidden-Markov Model, HMM)

- **multiples Alignment** aus bekannten Beispiel-Sequenzen

Beschreibung durch Multiples Alignment

Serpin-Domäne (Serin Protease Inhibitor)



Multiple Alignments

Motivation

- Möchte alle Mitglieder einer Proteinfamilie auf einmal betrachten, Gemeinsamkeiten / Unterschiede **auf einen Blick** sehen.
- Homologe (evolutionär sich entsprechende) Positionen besser sichtbar.
- Drei optimale paarweise Alignments von drei Sequenzen evtl. inkonsistent: *a* mit *b* aligniert, *b* mit *c*, aber *a* nicht mit *c*.
  In multiplem Alignment unmöglich: Alle Sequenzen gleichzeitig aligniert!

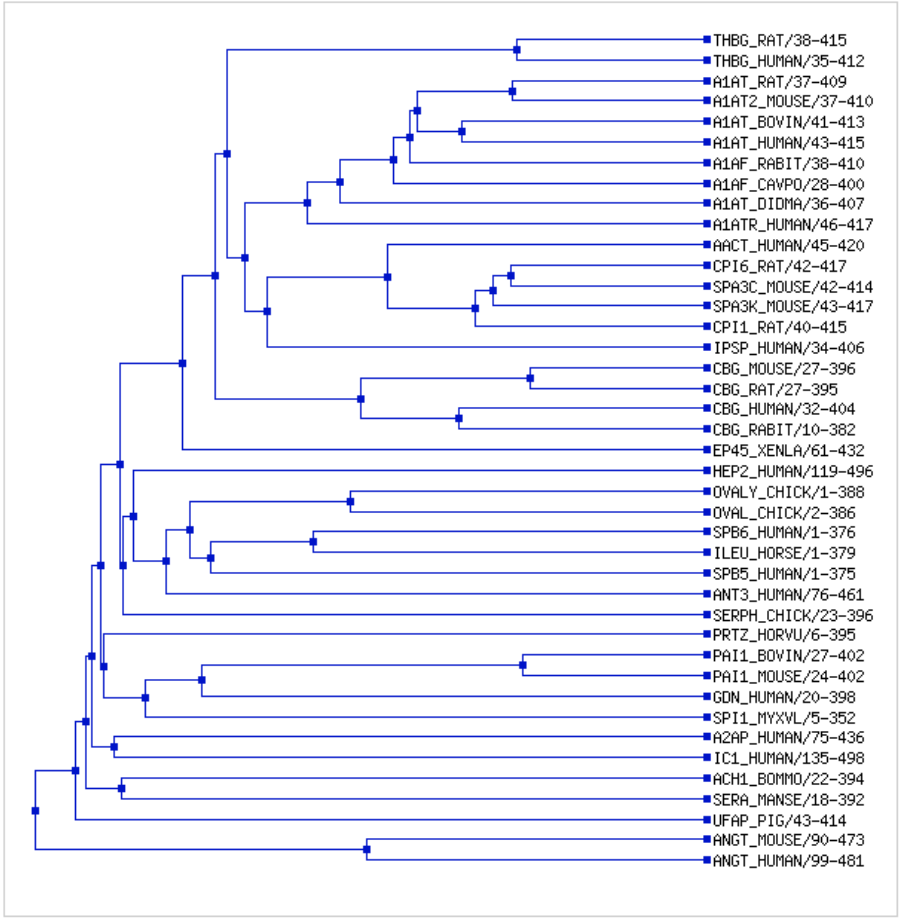
Multiples Alignment :=
Alignment von mindestens 3 Sequenzen.
Jede Zeile entspricht (durch Weglassen der Gaps) einer der Sequenzen.
Bei *k* Sequenzen kann jede Spalte 0 bis *k*-1 Gap-Zeichen enthalten.

Wann sind multiple Alignments sinnvoll?

Globales multiples Alignment nur sinnvoll, wenn
- Sequenzen global ähnlich,
- alle Sequenzen evolutionär verwandt

Lokales multiples Alignment sinnvoll:
- hinreichend lange gemeinsame ähnliche Teilsequenz aller Sequenzen (Domäne?)
- diese Bereiche evolutionär verwandt

Verwandtschaftsbeziehungen werden durch **phylogenetischen Baum** beschrieben.
Beispiel: Serpine in Pfam



Berechnung multipler Alignments

Ziel:
Biologisch / evolutionär korrektes multiples Alignment
(evolutionär voneinander abstammende Aminosäuren,
oder solche mit gemeinsamem Vorfahren, stehen untereinander)

Formulierung als Optimierungsproblem:
Definiere (wie schon auf paarweisen Alignments) eine Score-Funktion.
Finde das multiple Alignment, das den Score maximiert.

Problematisch:
Es gibt vermutlich kein Scoring-Verfahren,
das stets das evolutionär korrekte Alignment
zu dem mit dem höchsten Score macht (Beispiel).

Scoring von multiplen Alignments

Sum-of-pairs Score:
Multiples Alignment aus k Sequenzen enthält ~ $k^2/2$ paarweise Alignments.
Summe aller paarweisen Scores ergibt den Score des multiplen Alignments.

Tree score:
Annahme: Zwischen Sequenzen bestehen evolutionäre Verwandtschaften, gegeben durch phylogenetischen Baum.
Summiere paarweise Scores von im Baum benachbarten Sequenzen.

gewichteter Sum-of-pairs Score
wie Sum-of-pairs Score, aber jedes Paar erhält individuelles Gewicht

3 Optimierungsprobleme beim multiplen Alignment

Sum-of-pairs Problem

Gegeben *k* Sequenzen, Scorematrix, Gapkosten.
Finde multiples Alignment, das (gewichteten) sum-of-pairs Score maximiert.

Tree Alignment Problem

Gegeben zusätzlich ein Baum mit den *k* Sequenzen an den Blättern.
Finde Belegung der inneren Knoten mit Sequenzen (gemeinsame Vorfahren)
und multiples Alignment, das den Tree Score maximiert.

Verallgemeinertes Tree Alignment Problem

Gegeben *k* Sequenzen, Scorematrix, Gapkosten (kein Baum!),
finde Baumtopologie, Belegung der inneren Knoten mit Sequenzen
und multiplea Alignment, das den Tree Score maximiert

Komplexität des multiplen Alignment - Problems

Für alle drei Varianten des Problems
- Sum-of-pairs Problem
- Tree Alignment Problem
- Verallgemeinertes Tree Alignment Problem

gilt:

Es gibt exakte Algorithmen,
aber Zeitbedarf exponentiell in der Anzahl der Sequenzen k.
Nur praktikabel für 3 – 7 Sequenzen.
Multiples Alignment ist **NP-schwer**!

Verwendung von Heuristiken evtl. „nicht so schlimm“, denn:
- Score-maximales Alignment ungleich biologisch korrektes Alignment.

Heuristiken für multiples Alignment

Center-star-Methode:

- Wähle Sequenz (mit geringster evolutionärer Abstandssumme zu den anderen).
- Aligniere jede andere Sequenz paarweise daran.
- Setze k-1 paarweise Alignments zu einem multiplen Alignment zusammen.
- Nachteil: Es werden nur k-1 der möglichen paarweisen Alignments betrachtet.

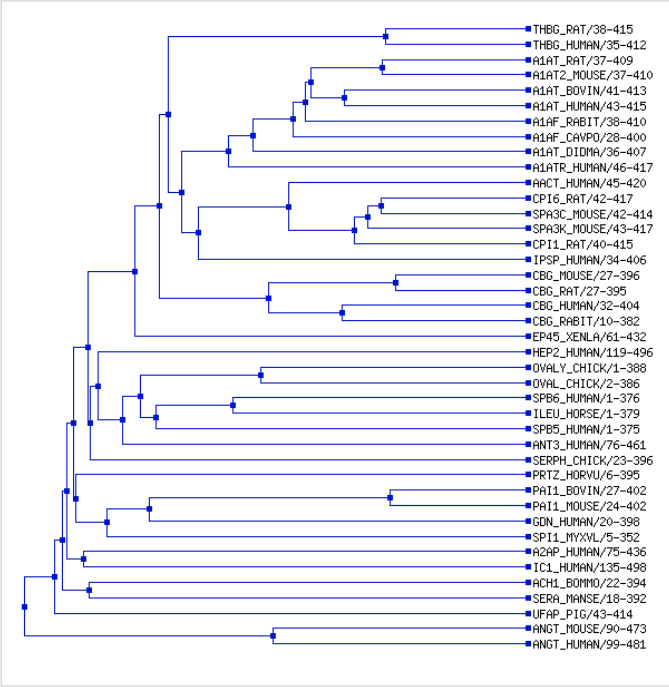
Divide-and-conquer Alignment:

- Suche etwa in der Mitte jeder Sequenz gut konservierte Stellen,
- an der man relativ sicher ist, dass alle diese Stellen eine Alignmentspalte bilden.
- Teile Problem in linkes und rechtes Problem auf, verfahre dort genauso.
- Nachteil: funktioniert nicht gut, wenn es keine solchen Stellen gibt.

Heuristiken für multiples Alignment

Progressives Alignment:

- Beginne mit zwei nah verwandten Sequenzen
- Aligniere in jedem Schritt eine weitere Sequenz optimal paarweise an bestehendes multiples Alignment.
- Reihenfolge durch evolutionäre Distanzen der Sequenzen bestimmt (Baum).
- Nachteil: falsche frühe Entscheidungen können nicht rückgängig gemacht werden.
- Wichtigstes Beispiel: **Clustal**



Clustal (http://www.clustal.org)

- schnelle Heuristik (und Software-Paket) für multiple Alignments

Webserver zum Berechnen von Alignments:
- am EBI: http://www.ebi.ac.uk/Tools/clustalw2/index.html
- am SIB: http://www.ch.embnet.org/index.html

Idee und Verfahren
- Berechne eine Folge von paarweisen Alignments
- Nimm dazu einen Baum zur Hilfe („guide tree“)
  (Art phylogenetischer Baum, aber eher ein Hilfsmittel zur Berechnung).
- Bereits existierende Teil-Alignments werden dabei nicht mehr verändert.

1. Berechne Distanzen zwischen allen Sequenzpaaren
2. Berechne aus den Distanzwerten einen Baum (mehr dazu später)
3. Aligniere Sequenzen und existierende Alignments
   in der Reihenfolge, die der Baum vorgibt (bottom-up).

Abhängigkeit von Baum und Alignment

Baum („guide tree“) bildet Verwandtschaftsverhältnisse der Sequenzen ab.
Wichtiges Hilfsmittel beim Berechnen des multiplen Alignments.
Woher bekommen?

Man schätzt evolutionäre Distanzen (z.B. in PAM) zwischen Sequenzen.
Dazu braucht man aber das Alignment.

Henne-Ei-Problem:

- Berechnung des Baums benötigt Alignment.
- Berechnung des Alignments benötigt Baum.

Abhängigkeit von Baum und Alignment

Lösung des Henne-Ei-Problems:

Beginne mit (groben) Schätzungen für Distanzen,
z.B. aus paarweisen Alignments (unterschätzen wahren Distanzen).
Berechne ersten Baum.
Erstelle erstes multiples Alignment.

Schätze daraus neue Distanzen, neuen Baum, neues Alignment.

Iteriere so lange, bis sich nichts mehr ändert,
oder eine Maximalzahl an Iterations-Schritten gemacht wurde.