UNIVERSITÄT
LEIPZIG

Algorithmen und Datenstrukturen II

Vorlesung 8осемь

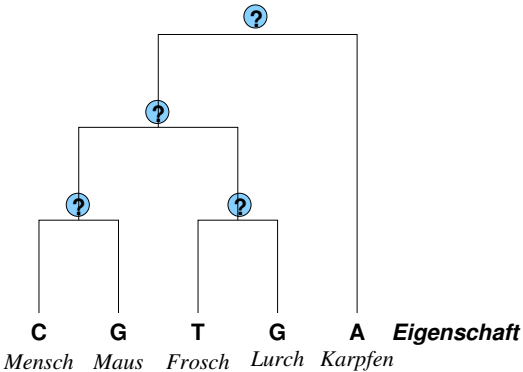
Leipzig, 21.05.2024
Peter F. Stadler & Thomas Gatter & Ronny Lorenz

Dynamische Programmierung: Bäume und Graphen

- In dieser VL-Einheit betrachten wir DP-Probleme auf Bäumen und Graphen an Beispielen.
- Im 1. Beispiel ist eine binärer Baum gegeben, der die Verwandtschaft einiger Spezies skizzieren soll.
- An den *Blättern* finden wir heute lebende Spezies (“extant”) und deren Eigenschaften. Es kann sich hierbei um Genome, Gene, oder andere Eigenschaften handeln. In unserem Beispiel ist jede Eigenschaft ein DNA-Buchstabe (`A,C,G,T`).
- Gegeben den phylogenetischen Baum wollen wir nun berechnen welche Eigenschaft die jeweiligen Vorfahren hatten. Mensch hat Eigenschaft `C`, Maus hat `G`. Dem inneren Elternknoten muss nun eine der vier Eigenschaften `A,C,G,T` zugewiesen werden.

Parsimony - Optimierung aus Bäumen I

Ein Problem aus der Bioinformatik



Können wir die Eigenschaften der Vorfahren (blaue Kreise/ Fragezeichen) rekonstruieren?

Ansatz: Minimiere die Anzahl der Änderungen der Eigenschaft entlang der Kanten des Baums.

Parsimony - Optimierung aus Bäumen II

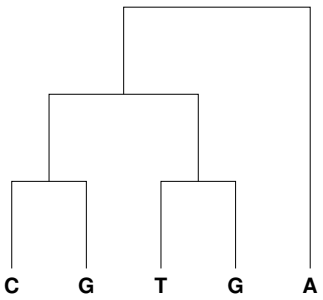
Eine *Änderung* liegt vor, wenn sich vom Elter zu Kind die Belegung ändert.

Beispiel

Setzen Sie den Elter von Mensch+Maus auf C, dann gibt es *keine* Änderung zu Mensch (C→C), allerdings sehr wohl eine Änderung zu Maus (C→G).

Wir wollen die beste Belegung mit möglichst wenig Brüchen effizienter als durch reines Durchtesten (hier: 4^4 Möglichkeiten) bestimmen!

Fitch Algorithm I



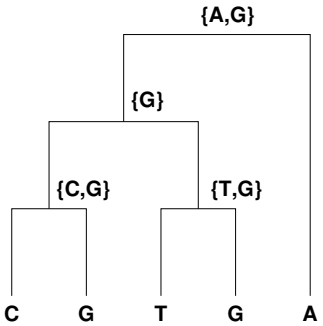
Vorwärts-Rekursion: Blätter belegt mit Daten.
$\rightarrow$ Postorder-Traversierung

Innerer Knoten w: nur die in den Nachfahren *u* und *v* belegten Eigenschaften sind plausibel.

Bestimme die **Menge möglicher Belegungen** $S(x)$ für jeden inneren Knoten:

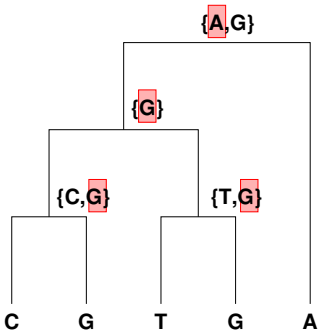
if $S(u) \cap S(v) \neq \emptyset$ then $S(w) = S(u) \cap S(v)$
keine Eigenschaftsänderung
else $S(w) = S(u) \cup S(v)$
*Änderung zwischen Elter und **einem** Kind*

Fitch Algorithm II



- Betrachten wir die beiden Blätter `C` und `G` ganz links. Die Möglichkeiten im Elter sind `{C,G}` während des Aufstieges. (Analog bei `T`, `G` in den beiden folgenden Blättern)
- Eine Ebene höher haben wir dann `{C,G}` links und rechts `{T,G}`. Hier gibt es nun min. ein gemeinsames Element, deshalb $\texttt{\{C,G\}} \cap \texttt{\{T,G\}} = \texttt{\{G\}}$.
- In der Wurzel schreiben wir nun wieder `{A,G}`, da `{G}` und `{A}` keine gemeinsamen Elemente haben.

Fitch Algorithm III



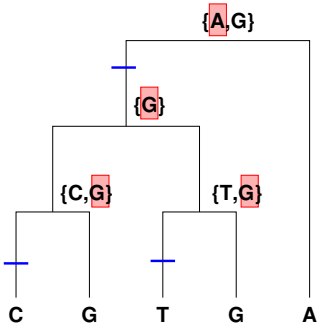
Bestimme eine **exakte Belegung** $s(x)$ für jeden inneren Knoten.

Rückwärts-Rekursion:
Wähle eine der Belegungen $s(w)$ der Wurzel.

Preorder-Traversierung:
Wenn die Wahl $s(v)$ im Elter v auch im Kind u vorhanden ist, $s(v) \in S(u)$ wähle diesen. $s(u) \leftarrow s(v)$
Wenn nicht, wähle $s(u)$ beliebig aus $S(u)$.

Fitch Algorithm IV

Abzählen der Änderungen: $s(v) \neq s(u)$ für jedes Elter/Kind-Paar



Übungsaufgabe: Modifizieren Sie den Algorithmus so, dass nur die Vorwärtsrekursion benötigt wird, um die Zahl der Änderungen zu berechnen.

Zwischenzusammenfassung: Wie funktioniert DP? I

- *Dynamische Programmierung* basiert auf dem Zusammenspiel zweier Prinzipien:
  - **Optimale Teilstruktur**
  - **Überlappende Teilprobleme**
- *Optimale Teilstruktur* liegt dann vor, wenn wir ein Problem in (i) kleinere Teilprobleme zerlegen können und (ii) die optimale Lösung des Problems sich mit Hilfe der optimalen Lösungen der Teilprobleme berechnen lässt. In diesem Fall können wir rekursiv die Lösung berechnen.

  Beispiel: Die n'te Fibonacci-Zahl $f(n) = f(n-1) + f(n-2)$ lässt sich rekursiv berechnen und endet mit $f(0) = 0; \quad f(1) = 1$.

Zwischenzusammenfassung: Wie funktioniert DP? II

– *Überlappende Teilprobleme* werden zusätzlich *memoisiert* (sic), also in einer Datenstruktur “für später” gespeichert. Überlappend bedeutet, dass jedes Teilproblem mehrmals gebraucht wird, wobei sich durch die Speicherung der “nochmalige” Rechenaufwand deutlich (oft $O(1)$) reduziert.

Beispiel

Berechnen der n’ten Fibonacci Zahl von klein nach gross. Wobei sie jedes $f(n)$ sofort speichern. Warum? Für $f(n)$ benutzen sie $f(n-2)$ gleich zweimal. Einmal direkt in der Funktion von $f(n)$ und einmal im Aufruf von $f(n-1)$. Je weiter wir absteigen in die Rekursion desto häufiger sehen wir Aufrufe der Form $f(k)$ wiederholt auftreten!

Traveling Salesman Problem (TSP) I

- **Gegeben:** n Städte, paarweise Distanzen:
  d_{ij} Entfernung von Stadt i nach Stadt j.
- **Gesucht:** Rundreise mit minimaler Länge durch alle Städte,
  also Permutation $\pi : (1, ..., n) \to (1, ..., n)$, für die $c(\pi)$ minimal ist.

$$c(\pi) = \left[\sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} \right] + d_{\pi(n)\pi(1)}$$

- *NP-hart* (d.h. exponentiell, ausser P=NP)
- *Naiver Algorithmus:* Alle $(n-1)! = 1 \times 2 \cdots \times (n-1)$ Reihenfolgen betrachten.

Traveling Salesman Problem (TSP) II

Dynamische Programmierung für TSP

- Sei $g(i, S)$ Länge des kürzesten Weges von Stadt i über jede Stadt in der Menge S (jeweils genau *ein* Besuch) nach Stadt 1.
- *Lösung des TSP also:* $g(1, \{2, ..., n\})$.

  Anmerkung: Die Start-Stadt i (hier: 1) kann beliebig gewählt werden, da Rundreise gesucht wird. $i \notin S$
- Es gilt:

$$g(i, S) = \begin{cases} d_{i1} & \text{falls } S = \emptyset \\ \min_{j \in S} \left[d_{ij} + g(j, S \setminus \{j\}) \right] & \text{sonst} \end{cases}$$

- $g(i, \emptyset) = d_{i1}$ entspricht dem “Rückweg” von i zum Start 1.

Traveling Salesman Problem (TSP) III

Algorithmus TSP

for $i = 2 ; i \leq n; i$++ **do**
  g[i,$\emptyset$] = d[i,1]

for $k = 1 ; k \leq n\text{-}2 ; k$++ **do**
  for $S, |S| = k, 1 \notin S$ **do**
    for $i \in \{2, \ldots, n\} \setminus S$ **do**
      Berechne g[i,S] gemäß Formel

Berechne g[1,{ 2,...,n }] gemäß Formel

Komplexität:
Tabellengröße $\times$ Aufwand je Tabelleneintrag

Größe: $< n2^n$
(Anzahl der i's mal Anzahl betrachtete Teilmengen)

Tabelleneintrag: Suche nach Minimum unter j aus S: $O(n)$

Insgesamt: $O(n^2 \times 2^n)$, deutlich besser als $(n-1)!$.

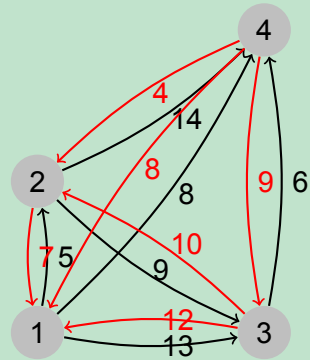
Dynamische Programmierung für TSP - Beispiel I

Beispiel I

Es seien **n = 4** und

D =

	1	2	3	4
1	-	5	13	8
2	7	-	9	14
3	12	10	-	6
4	8	4	9	-



Dynamische Programmierung für TSP - Beispiel II

Beispiel II

$|S| = 0$:

$g(2, \emptyset) = 7, g(3, \emptyset) = 12, g(4, \emptyset) = 8$

$|S| = 1$:

$g(2, \{3\}) = d_{23} + g(3, \emptyset) = 9 + 12 = 21$
$g(2, \{4\}) = d_{24} + g(4, \emptyset) = 14 + 8 = 22$
$g(3, \{2\}) = d_{32} + g(2, \emptyset) = 10 + 7 = 17$
$g(3, \{4\}) = d_{34} + g(4, \emptyset) = 6 + 8 = 14$
$g(4, \{2\}) = d_{42} + g(2, \emptyset) = 4 + 7 = 11$
$g(4, \{3\}) = d_{43} + g(3, \emptyset) = 9 + 12 = 21$

Dynamische Programmierung für TSP - Beispiel (3)

Beispiel III

$|S| = 2$:

$g(2,\{3,4\}) = \min\{d_{23} + g(3,\{4\}), d_{24} + g(4,\{3\})\} = min\{9+14, 14+21\} = 23$
$g(3,\{2,4\}) = \min\{d_{32} + g(2,\{4\}), d_{34} + g(4,\{2\})\} = min\{10+22, 6+11\} = 17$
$g(4,\{2,3\}) = \min\{d_{42} + g(2,\{3\}), d_{43} + g(3,\{2\})\} = min\{4+21, 9+17\} = 25$

$|S| = 3$:

$g(1,\{2,3,4\}) =$
$\min\{d_{12} + g(2,\{3,4\}), d_{13} + g(3,\{2,4\}), d_{14} + g(4,\{2,3)\}\} =$
$\min\{5+23, 13+17, 8+25\} = 28$

Dynamische Programmierung für TSP - Beispiel (4)

Beispiel IV

Die optimale Rundreise mit 1 als Startpunkt lautet also:
$\pi = (1, 2, 3, 4)$ mit der Länge 28

Bei jeder Berechnung $g(i, S)$ kann das j gespeichert werden, das als Minimum angenommen wird.
Die Rundreise π lässt sich dann wie folgt konstruieren:

- starte mit 1
- berechne j aus $g(1, S_0)$
- berechne π rekursiv weiter

KÜRZESTE WEGE

Kürzeste Wege

Definition

Für **kantenmarkierter (gewichteter) Graph** $G = (V, E, g)$

- Weg/Pfad P der Länge n: $(v_0, v_1), (v_1, v_2), , ..., (v_{n-1}, v_n)$
- Gewicht (Länge) des Weges/Pfads $w(P) = \sum_{i=1}^{n} g((v_{i-1}, v_i))$
- Distanz $d(u, v)$: Gewicht des kürzesten Pfades von u nach v

Varianten:

- nicht-negative Gewichte und negative und positive Gewichte
- Bestimmung der kürzesten Wege **a)** zwischen allen Knotenpaaren, **b)** von einem Knoten u aus oder **c)** zwischen zwei Knoten u und v.

Kürzeste Wege

Bemerkungen

- kürzeste Wege sind nicht immer eindeutig
- kürzeste Wege müssen nicht existieren, z.B. wenn:
  - kein Weg existiert oder
  - ein Zyklus mit negativem Gewicht existiert
- Graphen werden als gerichtet betrachted. Der ungerichtete Fall wird (wie gehabt) durch Kanten in beide Richtungen mit gleichen Gewichten modelliert.

Zur Erinnerung: Warshall's Transitive Hülle

Warshall Algorithmus (transitive Hülle)

```
boolean[][] A = {...} ;      // Adjazenzmatrix, 1-basiert
for (int i=1; i<=A.length; i++) do
    A[i][i] = 1 ;
for (int j=1; j<=A.length; j++) do
    for (int i=1; i<=A.length; i++) do
        if (A[i][j] == 1) then
            for (int k=1; k<=A.length; k++) do
                if (A[j][k] == 1) then
                    A[i][k] = 1 ;
```

Warshall's Algorithmus für Distanzen

Warshall-Algorithmus lässt sich einfach modifizieren, um kürzeste Wege zwischen allen Knotenpaaren zu berechnen

Warshall Algorithmus (Distanzen)

```
for alle Paare i,j do
    // ∞ wenn (i,j) ∉ E, 0 wenn i==j
    A[i][j] = g((i,j))
for j = 1; j ≤n; j++ do
    for i = 1; i ≤n; i++ do
        for k = 1; k ≤n; k++ do
            A[i][k] = min (A[i][k], A[i][j] + A[j][k])
```

Annahme: kein Zyklus mit negativem Gewicht vorhanden

Komplexität: $O(n^3), n = |V|$

Dijkstra-Algorithmus I

Bestimmung der von einem Knoten ausgehenden kürzesten Wege

- **Gegeben:** Kanten-bewerteter Graph $G = (V, E, g)$ mit $g : E \to \mathbb{R}_0^+$ (Kantengewichte)
- Startknoten s
  zu jedem Knoten u wird die Distanz zu Startknoten s in $D[u]$ geführt
- Q sei Prioritäts-Warteschlange (sortierte Liste)
  Priorität = Distanzwert
- Funktion $succ(u)$ liefert die Menge der direkten Nachfolger von u
- Verallgemeinerung der Breitensuche (gewichtete Entfernung)
- funktioniert nur bei nicht-negativen Gewichten
- Optimierung gemäß Greedy-Prinzip

Dijkstra-Algorithmus II

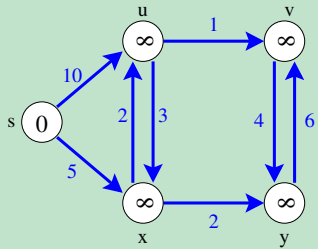
Algorithmus

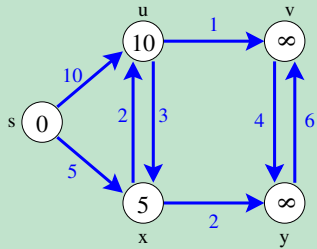
for *alle Knoten v* **do**
    D[v] = ∞

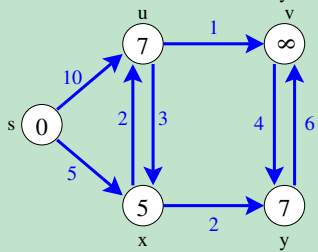
D[s] = 0
PriorityQueue Q = V
while *Q nicht leer* **do**

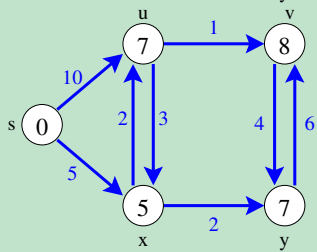
    v = nächster Knoten aus Q mit kleinstem Abstand/Gewicht in D
    Entferne v aus Q
    for *jeden Nachbarn u in succ(v) und u in Q* **do**
        if *D[v] + g((v,u)) < D[u]* **then**
            D[u] = D[v] + g((v,u))

Beispiel - Dijkstra









Dijkstra-Algorithmus: Korrektheit

Korrektheitsbeweis

- nach i Schleifendurchgängen sind die Längen von i Knoten, die am nächsten an s liegen, korrekt berechnet und diese Knoten sind aus Q entfernt.
- **Induktionsanfang:** s wird gewählt, $D(s) = 0$
- **Induktionsschritt:** Nimm an, v wird aus Q genommen. Der kürzeste Pfad zu v gehe über direkten Vorgänger v' von v. Da v' näher an s liegt, ist v' nach Induktionsvoraussetzung mit richtiger Länge $D(v')$ bereits entfernt. Da der kürzeste Weg zu v die Länge $D(v') + g((v', v))$ hat und dieser Wert bei Entfernen von v' bereits v zugewiesen wurde, wird v mit der richtigen Länge entfernt.
- erfordert nicht-negative Kantengewichte (wachsende Weglänge durch hinzugenommene Kanten)

□

Dijkstra-Algorithmus: Eigenschaften

Komplexität $O(n^2), n = |V|$

- n-maliges Durchlaufen der äußeren Schleife liefert Faktor $O(n)$
- innere Schleife: Auffinden des Minimums begrenzt durch $O(n)$, ebenso das Aufsuchen der Nachbarn von v
- Pfade bilden aufspannenden Baum (der die Wegstrecken von s aus gesehen minimiert)
- Bestimmung des kürzesten Weges zwischen u und v:
  Spezialfall für Dijkstra-Algorithmus mit Start-Knoten u (Beendigung sobald v aus Q entfernt wird)