UNIVERSITÄT
LEIPZIG

Algorithmen und Datenstrukturen II

Vorlesung nén9

Leipzig, 28.05.2024
Peter F. Stadler & Thomas Gatter & Ronny Lorenz

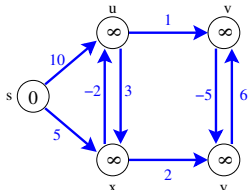
Bellmann-Ford

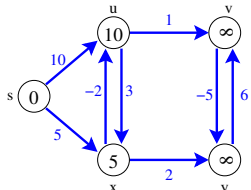
Bellmann-Ford erlaubt uns kürzeste Pfade von einem Startknoten s aus auch mit negativen Kantengewichten zu bestimmen (ohne negative Zyklen).

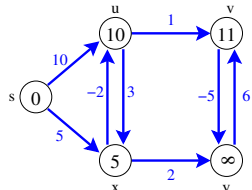
Bellmann-Ford-Algorithmus BF(G,s)

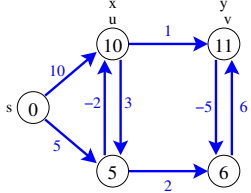
```
for alle Knoten v do
|    D[v] = ∞
D[s] = 0
 for i = 1; i ≤n-1; i++ do

|    for alle Paare (u,v) do
|    |    if D[u] + g((u,v)) < D[v] then
|    |    |    D[v] = D[u] + g((u,v))
```

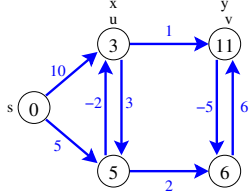
Bellmann-Ford Beispiel: (1. Durchgang)







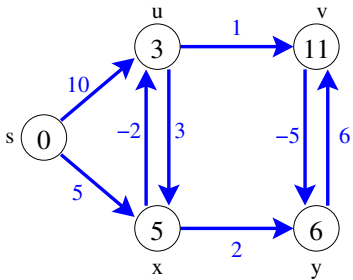




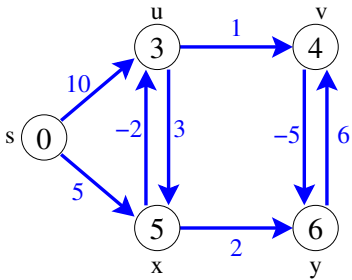
Schritte in denen nichts passiert sind nicht separat gezeichnet

Kanten werden hier in lexikographischer Ordnung durchlaufen:
$(s,u)(s,x)(u,v)(u,x)(v,y)(x,u)(x,y)(y,v)$

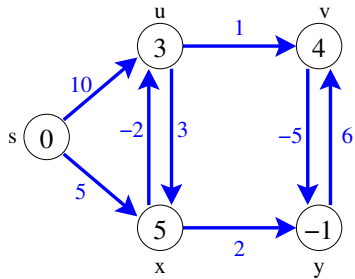
Bellmann-Ford Beispiel (2. Durchgang)



nach 1.Durchgang



(u, v)



(v, y)

Im 3. und 4. Durchgang passiert nichts mehr weiter

Durchlauf-Reihenfolge: $(s, u)(s, x)(u, v)(u, x)(v, y)(x, u)(x, y)(y, v)$

Anwendungsbeispiel: Edmonds-Karp Algorithm

Kurz zurück zu den Flüssen

- Edmonds-Karp Algorithm =
  vebesserter Ford-Fulkerson mit zusätzlicher Bedingung, dass immer ein **kürzester Erweiterungspfad** benutzt wird
- Löst Flussproblem in $O(|V||E|^2)$ für beliebige Kapazitäten.
- Für diese Anwendung haben alle Kanten die selbe Länge und es existiert eine einfachere Lösung des Shortest Path Problems:
  BFS ist ausreichend.

Warum geht das?

Warum Dynamische Programmierung nützlich ist

- An einer völlig hypothetischen Universität ist es üblich, dass in Klausuren viel mehr Aufgaben gestellt werden als in der vorgegebenen Zeit t^* gelöst werden können.
- Ein ebenfalls völlig hypothetischer Student möchte nun seinen Punktestand optimieren.

Warum Dynamische Programmierung nützlich ist

Welche Aufgaben sollten bearbeitet werden?

Für jede Aufgabe $i \in \{1, \ldots, n\}$ kennen wir die Bearbeitungszeit t_i und die dafür erreichbare Punktezahl p_i. Die Aufgabe besteht also darin, eine Teilmenge $S \subseteq \{1, \ldots, n\}$ zu finden, so dass

$$\sum_{i \in S} p_i \to \max \qquad \text{während} \qquad \sum_{i \in S} t_i \leq t^*$$

Üblicherweise lässt man stattdessen einen Dieb mit Rucksack, der ein Fassungsvermögen t^*, Gegenstände mit Wert p_i und Volumen t_i aus einem Tresor klauen ... weswegen das Problem Rucksackproblem und nicht Klausurproblem heißt.

Das 0/1-Rucksackproblem

0/1-Rucksackproblem

- Objekte $\{1, 2, \ldots, n\}$
- mit Volumina $t_1, t_2, \ldots, t_n$ und Werten $p_1, p_2, \ldots, p_n$
- Rucksack hat Gesamtvolumen c.

Gesucht: Ein $0/1$-Vektor $a_1, ..., a_n \in \{0, 1\}$ mit

$$\sum_{i=1}^{n} a_i t_i \leq c$$

Optimierungsproblem:
finde a, so dass der Gesamtwert $f(a)$ maximal wird.

$$f(a) = \sum_{i=1}^{n} a_i p_i$$

0/1 Rucksackproblem

Es sei $O_i \subseteq 1, 2, ..., i$ eine optimale Packung bei Kapazitätsgrenze t.

Dann gilt:

- Entweder $i \in O_i$, dann ist $O_i \setminus \{i\}$ eine optimale Packung aus $\{1, ..., i-1\}$ bei Kapazitätsgrenze $t - t_i$ oder
- $i \notin O_i$, dann ist O_i eine optimale Packung aus $\{1, ..., i-1\}$ bei Kapazitätsgrenze t

Algorithmus für das 0/1-Rucksackproblem

Es sei $F_{i,t}$ der maximale Wert beim Packen des Rucksacks mit i Objekten aus $\{1, 2, ..., i\}$ bei Kapazitätsgrenze t.

$F_{i,t}$ kann dann rekursiv wie folgt berechnet werden

Berechnung von $F_{i,t}$:

$$F_{i,t} = \begin{cases} 0 & \text{falls } i = 0 \\ F_{i-1,t} & \text{falls } i > 0 \text{ und } t < t_i \\ \max\{F_{i-1,t}, F_{i-1,t-t_i} + p_i\} & \text{sonst} \end{cases}$$

Beispieltabelle $F_{i,t}$

$c = 10$

i	1	2	3	4	5
p_i	8	2	7	8	3
t_i	3	1	4	5	2

		t									
		1	2	3	4	5	6	7	8	9	10
i	1	0	0	8	8	8	8	8	8	8	8
	2	2	2	8 (1)	10 (1,2)	10 (1,2)	10 (1,2)	10 (1,2)	10 (1,2)	10 (1,2)	10 (1,2)
	3	2 (2)	2 (2)	8 (1)	10 (1,2)	10 (1,2)	10 (1,2)	15 (1,3)	17 (1,2,3)	17 (1,2,3)	17 (1,2,3)
	4	2 (2)	2 (2)	8 (1)	10 (1,2)	10 (1,2)	10 (1,2)	15 (1,3)	17 (1,2,3)	18 (1,2,4)	18 (1,2,4)
	5	2 (2)	3 (5)	8 (1)	10 (1,2)	11 (1,5)	13 (1,2,5)	15 (1,3)	17 (1,2,3)	18 (1,3,5)	20 (1,2,3,5)

Mehrfach-Rucksackproblem

Nehmen wir an, dass all Gegenstände mehrfach vorhanden sind Jeder Gegenstand kann **mehrfach** (so oft wie gewünscht) eingepackt werden.

Definiere

- `kosten[j]` ... Volumen von `j`
- `wert[j]` ... Wert von `j`
- Sei W_i der maximale Wert von Gegenständen mit Volumen i
- Wenn bei Volumen i der letzte eingepackte Gegenstand j ist, so ist der beste Wert $\texttt{wert}[j] + W_{i-\texttt{kosten}[j]}$
- Also:

$$W_i = \max_j \left(\texttt{wert}[j] + W_{i-\texttt{kosten}[j]}\right)$$

Mehrfach Rucksackproblem

- Effiziente Berechnung, indem die Operationen in einer zweckmäßigen Reihenfolge ausgeführt werden.
- **Variablen**
  - `i` Kapazität ($1 \leq$ `i` $\leq$ `M`)
  - `j` Gegenstand ($1 \leq$ `j` $\leq$ `N`)
- **Felder**
  - `best_wert[i]` ... bester Wert bei Kosten `i` ($\hat{=} W_i$)
  - `best_obj[i]` ... ausgewähltes `j` für besten Wert bei Kosten `i`

Mehrfach Rucksackproblem

Algorithmus

```
for j = 1 ; j <= N ; j ++ do
    for i = 1 ; i <= M ; i++ do
        if i >= kosten[j] then
            if best_wert[i] < best_wert[i - kosten[j]] + wert[j] then
                best_wert[i] = best_wert[i - kosten[j]] + wert[j];
                best_obj[i] = j;
```

Mehrfach-Rucksackproblem: Beispiel

Beispiel

Bezeichnung	A	B	C	D	E
Wert	4	5	10	11	13
Kosten	3	4	7	8	9

Mehrfach-Rucksackproblem: Beispiel

Lösung des Rucksack-Beispiels:

```
j = 1;       i 1 2 3 4 5 6  7  8  9 10 11 12 13 14 15 16 17
best_wert[i]   0 0 4 4 4 8  8  8 12 12 12 16 16 16 20 20 20
best_obj[i]    - - A A A A  A  A  A  A  A  A  A  A  A  A  A
j = 2
best_wert[i]   0 0 4 5 5 8  9 10 12 13 14 16 17 18 20 21 22
best_obj[i]    - - A B B A  B  B  A  B  B  A  B  B  A  B  B
j = 3
best_wert[i]   0 0 4 5 5 8 10 10 12 14 15 16 18 20 20 22 24
best_obj[i]    - - A B B A  C  B  A  C  C  A  C  C  A  C  C
j = 4
best_wert[i]   0 0 4 5 5 8 10 11 12 14 15 16 18 20 21 22 24
best_obj[i]    - - A B B A  C  D  A  C  C  A  C  C  D  C  C
j = 5
best_wert[i]   0 0 4 5 5 8 10 11 13 14 15 17 18 20 21 23 24
best_obj[i]    - - A B B A  C  D  E  C  C  E  C  C  D  E  C
```

Mehrfach-Rucksackproblem: Beispiel

- Das erste Zeilenpaar zeigt den maximalen Wert (den Inhalt der Felder `best_wert` und `best_obj`), wenn nur Elemente A benutzt werden.
- Das zweite Zeilenpaar zeigt den maximalen Wert, wenn nur Elemente A und B verwendet werden, usw.
- Der höchste Wert, der mit einem Rucksack der Größe 17 erreicht werden kann, ist 24.
- Im Verlaufe der Berechnung dieses Ergebnisses hat man auch viele Teilprobleme gelöst, z. B. ist der größte Wert, der mit einem Rucksack der Größe 16 erreicht werden kann, 22, wenn nur Elemente A, B und C verwendet werden.
- Der tatsächliche Inhalt des optimalen Rucksacks kann mit Hilfe des Feldes best_obj berechnet werden. Per Definition ist best_obj[M] in ihm enthalten, und der restliche Inhalt ist der gleiche wie im optimalen Rucksack der Größe M - kosten [best_obj [M]] usw.

Rucksackproblem: Eigenschaften

- Rucksack-Problems mit DP: O($N \cdot M$) Zeit
- $\Rightarrow$ Rucksack-Problem leicht, wenn M klein ist;
  sonst kann die Laufzeit sehr groß werden.
- **Problem:** die Effizienz dieses Verfahrens hängt davon ab, wieviele *verschiedene Werte* die Kosten annehmen können.
  Z.B. kritisch, wenn Kosten keine kleinen ganzen Zahlen sind.
  (Man kann das Verfahren auf reell-wertige Kosten erweitern, bekommt aber dann schnell dieses Problem.)
- Grundsätzlich gilt bei diesem Algorithmus, dass optimale Entscheidungen nicht geändert werden müssen, nachdem sie einmal getroffen wurden.
  ($\hat{=}$ *Bellmansches Optimalitätsprinzip*)
- *Zur Erinnerung:* DP funktioniert, immer wenn (dem Algorithmus eine Problemzerlegung zugrunde gelegt wird, bei der) dieses Prinzip gilt.

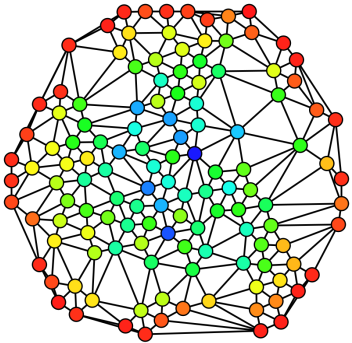
DP ist nicht nur eine Optimierungsmethode

Dynamische Programmierung kann auch zum Abzählen genutzt werden.

– Betweenness Zentralität eines Knoten in einem ungerichteten Graphen

$$C_B(v) = \sum_{s \neq t \neq v} \sigma_{st}(v)/\sigma_{st}$$

wobei σ_{st} die Zahl der kürzesten Pfade zwischen s und t und $\sigma_{st}(v)$ die Zahl der kürzesten Pfade zwischen s und t ist, die durch v laufen.

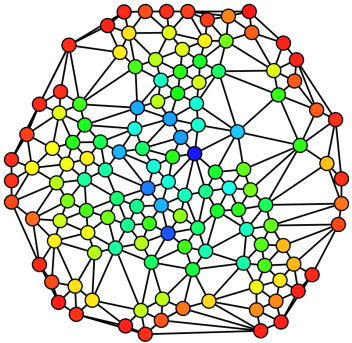


Farbe gibt $C_B(v)$ von klein bis gross

DP ist nicht nur eine Optimierungsmethode

Dynamische Programmierung kann auch zum Abzählen genutzt werden.

– Wir nehmen an, dass wir bereits die Distanzmatrix des gegebenen Graphen G berechnet haben, also $d(x, y)$, für alle $x, y \in V$ bekannt ist.
($\rightarrow$ Dijkstra oder Bellman-Ford Algorithmus)



Farbe gibt $C_B(v)$ von klein bis gross

Betweenness Zentralität

- **Beobachtung:** v liegt auf dem kürzesten Pfad von x nach y genau dann wenn $d(x, y) = d(x, v) + d(v, y)$.
- Fixiere eine Startpunkt s und setze $\sigma_{ss} = 1$
  Ordne die $x \in V$ nach Entfernung von s ... $d(s, x)$ kennen wir ja.
- Betrachte einen Knoten x. Ein Nachbar u von x liegt auf einem kürzesten Pfad von x nach v wenn $d(s, u) + g(u, x) = d(s, x)$.

Betweenness Zentralität

– Also gilt für die Zahl der kürzesten Pfade:

$$\sigma_{s,x} = \sum_{\substack{u \in N(x) \\ d(s,u)+g(u,x)=d(s,x)}} \sigma_{s,u}$$

... denn kürzeste Pfade, die durch verschiedene Nachbarn laufen, sind jedenfalls verschieden. Deswegen können die Anzahlen σ_{su} einfach addiert werden.

Vergleichen Sie das z.B. mit dem Dijkstra Algorithmus

Betweenness Zentralität

– Sind die Anzahlen der kürzesten Pfade $\sigma_{x,y}$ erst einmal für alle Knoten-Paare bekannt, können wir auch die Zahl der kürzesten Pfade **durch** einen Knoten einfach berechnen:

$$\sigma_{s,t}(v) = \begin{cases} \sigma_{s,v}\sigma_{v,t} & \text{falls } d(s,t) = d(s,v) + d(v,t) \\ 0 & \text{sonst} \end{cases}$$

– Hier nutzen wir, dass jeder kürzeste $s - v$ Pfad mit jedem kürzesten $v - t$ Pfad kombiniert werden kann. Die Anzahl der Pfade multipliziert sich daher.

Hirschberg's Algorithm for String Editing I

String-editing I

Bekannt: Globales Alignment / String Editing via Dynamischer Programmierung

- Gegeben Strings X, Y mit $|X| = n$ und $|Y| = m$
- DP Tabelle der Größe nm
- Laufzeit: nm: Insertion, Deletion, Substitution je Zelle
- Speicherplatz: nm
- Backtracing um eines der (co-)optimalen Alignments auszugeben $\min(n, m)$

Hirschberg's Algorithm for String Editing II

String-editing II

Hinweis: Falls *kein*(!) Backtracing nötig: Speicherplatz $2\min(n, m)$, da jeweils zwei Zeilen/Spalten ausreichen. Für backtracing braucht man für jede Zelle die Vorgänger. Das erfordert die komplette DP Tabelle!

Überlegung: Wie kann man einen Backtrace bekommen, ohne die komplette DP Tabelle behalten zu müssen? *Hirschberg-Algorithmus*!

Vorüberlegungen zum Hirschberg

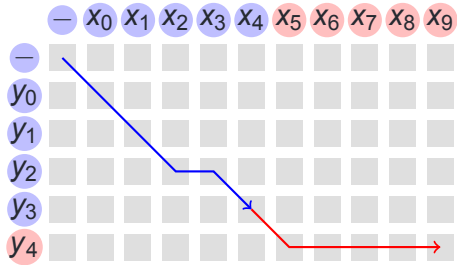
Überlegung: Sei $n = |X| \gg m = |Y|$. Sei $F(\cdot, \cdot)$ unser DP-Algorithmus. $F(X, Y)$ braucht nm Platz falls Backtracing gewollt, $2m$ Platz, falls nur die optimalen Kosten berechnet werden sollen.

Gesucht: Falls es einen Algorithmus $F'(\cdot, \cdot)$ *gäbe* mit $F(X, Y) \equiv F'(X_{0\ldots k}, Y) \oplus F'(X_{k+1\ldots n-1}, Y)$ dann könnte man F' rekursiv aufrufen und würde nie mehr als cm Platz brauchen mit c einer Konstante "in der Nähe" von 2. Hierbei setzt $\oplus$ zwei optimale Teilalignments so zusammen das wir die optimalen Scores erhalten und auch deren Backtrace.

Hinweis: F' lässt sich so nicht finden, aber eine Erweiterung von F', der *Hirschberg-Algorithmus*, liefert was wir suchen!

1. Schritt zum Hirschberg

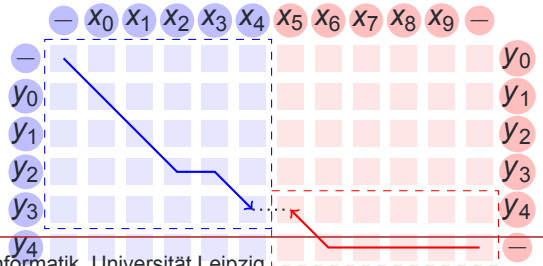
Ziel: Erweitere F', so dass wir aus $X_{0\ldots k}$ und $X_{k+1\ldots n-1}$ getrennt deren optimale Alignments an Y berechnen können und diese zusammen das optimale Alignment von X and Y liefern.



Problem: X_{k+1} braucht das optimale Alignment an X_k, welches wir im “rechten” Teilschritt nicht haben (sonst müssten wir die ganze Matrix speichern). Hier: Match (x_5, y_4) kommt von (x_4, y_3).

Lösung zur Zerlegung

Lösung: Gegeben sei die Funktion $\texttt{rev}(\cdot)$ welche einen String umkehrt:
$\texttt{rev}(x_0 x_1 \dots x_{n-2} x_{n-1}) == x_{n-1} x_{n-2} \dots x_1 x_0$.
Berechne $F_{\text{left}} = F(X_{0\dots k}, Y)$ und
$F_{\text{right}} = F(\texttt{rev}(X_{k+1\dots n-1}), \texttt{rev}(Y)) = F(X_{n-1\dots k+1}, Y_{m-1\dots 0})$.
$F_{\text{left}} \oplus F_{\text{right}}$ können nun zusammen gefügt werden um die optimale Lösung zu finden.
Dabei wird zusätzlich ein Index l gefunden in $Y_{0\dots l\dots m-1}$.
Hinweis: Das optimale Alignment geht durch die Zelle (k, l).



Zum Schnitt zwischen x_4, x_5

Das Alignment wird getrennt für den **blauen** und **roten** Teil gerechnet, wobei in jedem Teil nur das Optimum berechnet wird, kein Backtrace.

An der Schnittstelle zwischen rot und blau wird von oben nach unten die Summe der nebeneinander liegenden Paare berechnet:
$\{(l, F_{\text{left}}(4, l) + F_{\text{right}}(5, l))|l \in \{0, 4\}\}$.

Finde das l das die Summe minimiert (gepunktete Linie im Bild).
Zerlege den blauen und roten Bereich und rufe jeweils rekursiv den Hirschberg-Algorithmus auf diesen Bereichen auf (gestrichelte Kästen).

Zum Schnitt zwischen x_4, x_5

Überlegung: Jeder Aufruf von F (Hirschberg) benötigt $2m$ viele Spalten. An der Verbindung der Spalten von rot und blau werden die jeweiligen Endspalten benötigt. Es wird jeweils *nicht* die ganze Tabelle gebraucht, nur die Spalten. Dies gilt damit rekursiv genau so!

Backtracing: Bei jeder Rückkehr aus einem $F(\cdot, \cdot)$-Aufruf wird der jeweilige Backtrace zurückgegeben.

Im Folgenden wird eine Hilfsfunktion $S_L(X, Y)$ bzw. $S_R(X, Y)$ genutzt, welche die rechteste blaue, bzw. linkeste rote Spalte zurück gibt. S berechnet normal die Edit-Distanz, wobei jeweils nur zwei Spalten im Speicher gehalten werden. $edit(X, Y)$ berechnet die Edit-distanz inklusive Backtracing.

Hirschberg: Pseudocode

Hirschbergalgorithmus

```
F(X,Y):
    if |X|<=1 || |Y| <= 1 then
        return edit(X,Y)
    else
        k = |X| / 2
        left = S_L(X[0…k],Y)
        right = S_R(rev(X[k+1…|X|-1]),rev(Y))
        l = argmin (left+rev(right))
        return F(X[0…k],Y[0…l]) + F(X[k+1…|X|-1],Y[l+1 …|Y|-1])
```

Wir nutzen hier aus, dass wir implizit Paare aus `(score,alignment)` haben für die gilt, dass scores addiert und alignments konkateniert werden.

Berechnung von Distanzen

Ein wichtiger Punkt ist zu überlegen welche Art von Alignment-Berechnungen wird mittels des Hirschberg-Algorithmus durchführen können.

- globale Alignments, lineare Kosten, sowohl mit Maximierung als auch Minimierung sind einfach.
- lokale Alignments können bei beliebigen (k, l) starten und beliebigen (s, t) aufhören. In lokalen Alignments wird die komplette DP-Tabelle gescanned um den Stopp-punkt des besten lokalen Alignments zu finden. *Überlegen Sie: Ist das kompatibel mit Hirschberg?*
- Affine Kosten der Art $\alpha + d\beta$ mit d in/dels in Folge erfordern trickreiche Kombination der Tabellen. Bei der Berechnung in den “roten” Bereichen auf gedrehten (`rev`) Eingaben werden die affine Kosten α nicht am Beginn eines in/del Bereiches, sondern am Ende addiert.