Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 18 (25.6.2018)

Dynamische Programmierung II

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Fibonacci Zahlen

Definition der Fibonacci Zahlen $F_0, F_1, F_2, \ldots$:

$$F_0 = 0, \qquad F_1 = F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

Ziel: Berechne F_n

Rekursiver Algorithmus (basierend auf der rek. Definition):

```
fib(n):
    if n <= 1:
        f = n
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

Algorithmus mit Memoization

Memoization: Man merkt sich schon berechnete Werte
(auf einem Notizzettel = memo)

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    if n <= 1:
        f = n
    else:
        f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```

DP: Etwas Genauer …

DP ≈ Rekursion + Memoization

Memoize: *Speichere* Lösungen zu *Teilproblemen*, verwende die gespeicherten Lösungen, falls das gleicht Teilproblem wieder auftaucht.

- Bei den Fibonacci-Zahlen sind die Teilprobleme $F_1, F_2, F_3, \ldots$

Laufzeit = #Teilprobleme · Zeit pro Teilproblem

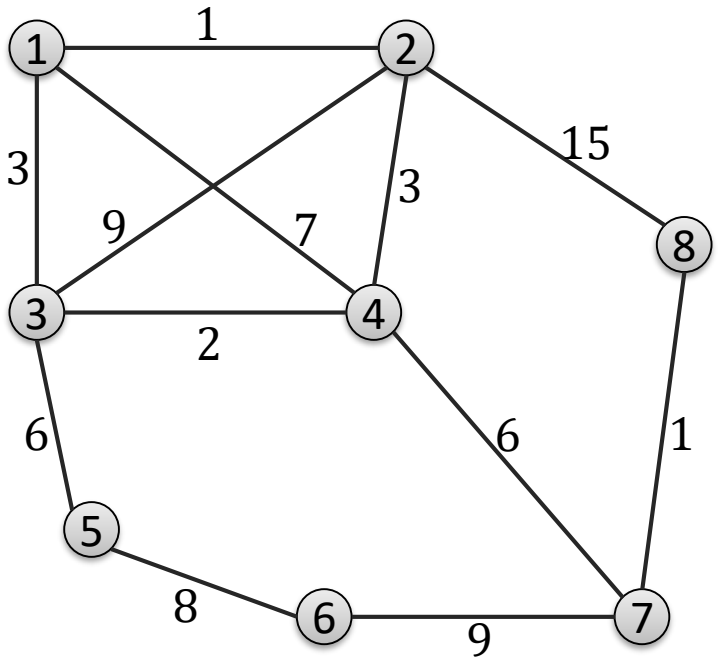
Fibonacci: Bottom-Up

```
fib(n):
    fn = [None] * (n + 1)
    for k in [0, 1, 2, …, n]:
        if k <= 1:
            f = k
        else:
            f = fn[k-1] + fn[k-2]
        fn[k] = f
    return fn[n]
```

Kürzeste Wege mit DP

Problem

- Gegeben: gewichteter Graph $G = (V, E, w)$, Startknoten $s \in V$
  - Wir bezeichnen Gewicht einer Kante (u, v) als $w(u, v)$
  - Annahme: $\forall e \in E: w(e)$, keine negativen Kreise
- Ziel: Finde kürzeste Pfade / Distanzen von s zu allen Knoten
  - Distanz von s zu v: $d_G(s, v)$ (Länge eines kürzesten Pfades)



Kürzeste Wege, rekursive Formulierung

- Wir interessieren uns nur für die Distanzen $d_G(s,v)$ (für alle v)

Rekursive Charakterisierung von $d_G(s,v)$?

Raten der Lösung

UNI FREIBURG

Kürzester Pfad von s nach v?

- Welches ist die letzte Kante des kürzesten Pfads?

- **Rate**, dass es die Kante (u, v) ist!
- Der Pfad ist ein kürzester Pfad von s nach u + Kante (u, v)

  Kosten: $d_G(s, u) + w(u, v)$

- Um die beste Möglichkeit zu finden, teste alle Möglichkeiten

- Zeit, um ein Teilproblem zu lösen:
  typischerweise dominiert durch die Anzahl Möglichkeiten um ein Teilproblem auf kleinere Teilprobleme zu reduzieren

Kürzeste Wege, rekursive Formulierung

Rekursive Charakterisierung von $d_G(s,v)$?

$$d_G(s,v) = \min_{(u,v)\in E}\{d_G(s,u) + w(u,v)\}, \qquad d_G(s,s) = 0$$

Kürzeste Wege in azyklischen Graphen

```
memo = {}
distance(v):
    if v in memo: return memo[v]
    d = ∞
    if s == v:
        d = 0
    else:
        for (u,v) in E:
```

(gehe durch alle eingehenden Kanten von v)

```
        d = min(d, distance(u) + w(u,v))
    memo[v] = d
    return d
```

Kürzeste Wege in allgemeinen Graphen

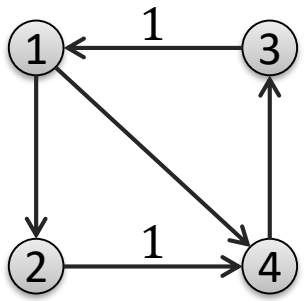
UNI FREIBURG

Idee: Führe zusätzliche Teilprobleme ein, um die zyklischen Abhängigkeiten zu vermeiden

Teilprobleme $d_G^{(k)}(s,v)$

- Länge des kürzesten Pfades bestehend aus höchstens k Kanten

Abhängigkeitsgraph ("Auffalten" von G)



Rekursive Formulierung in allg. Graphen

Teilprobleme $d_G^{(k)}(s,v)$

- Länge des kürzesten Pfades bestehend aus höchstens k Kanten

Rekursive Definition:

$$d_G^{(k)}(s,v) = \min_{(u,v)\in E}\left\{d_G^{(k-1)}(s,u) + w(u,v)\right\}$$
$$d_G^{(k)}(s,s) = 0, \qquad (\forall k \geq 0)$$
$$d_G^{(0)}(s,v) = \infty, \qquad (\forall v \neq s)$$

DP ≈ Rekursion + Memoization + Raten

Kürzeste Wege in allgemeinen Graphen

UNI FREIBURG

```
memo = {}
distance(k, v):
    if (k, v) in memo: return memo[(k, v)]
    d = ∞
    if s == v:
        d = 0
    elif k > 0:
        for (u,v) in E:
                                    (gehe durch alle eingehenden Kanten von v)
            d = min(d, distance(k-1, u) + w(u,v))
    memo[(k, v)] = d
    return d

distance(v):
    return distance(n-1, v)
```

Kürzeste Wege mit DP: Laufzeit

UNI FREIBURG

Laufzeit bei DP typischerweise:

$$\#\text{Teilprobleme} \cdot \text{Zeit pro Teilproblem}$$

- Zeit pro Teilproblem: rekursive Aufrufe kosten 1 Zeiteinheit
  - Durch die Memoization wird jedes Teilproblem nur einmal aufgerufen
  - Rekursive Kosten sind daher durch 1. Faktor abgedeckt
- Zeit pro Teilproblem: typischerweise #rek. Möglichkeiten

Kürzeste Wege:

- #Teilprobleme:
- Zeit pro Teilproblem:

Kürzeste Wege: Bottom-Up

- Normalerweise werden dynamische Programme bottom-up aufgeschrieben
  - ist oft effizienter (keine Rekursion, keine Hashtabelle)
  - ist oft eine natürliche Formulierung des Algorithmus
- Bottom-Up DP Algorithmus
  - Benötigt Reihenfolge in welcher die Teilprobleme berechnet werden können (topologische Sortierung des Abhängigkeitsgraphen)
  - Da man sowieso sicherstellen muss, dass keine zyklischen Abhängigkeiten bestehen, ist diese topologische Sortierung oft sehr einfach zu erhalten
- Reihenfolge beim kürzeste Wege Problem
  - Sortiere $d_G^{(k)}(s,v)$ aufsteigend nach k
  - Für gleiche k-Werte gibt es keine Abhängigkeiten

Kürzeste Wege: Bottom-Up

```
dist = {}
for k in range(n):
    for v in V:
        d = ∞
        if v == s:
            d = 0
        elif k > 0:
            for (u,v) in E:
                                (gehe durch alle eingehenden Kanten von v)
                d = min(d, dist[(k-1, u)] + w(u,v))
        dist[(k, v)] = d
```

5 Schritte zur DP Lösung

5 Schritte	Analyse
1) Teilprobleme definieren	#Teilprobleme zählen
2) Raten (Teil der Lösung)	#Möglichkeiten zählen
3) Rekursionsformel aufstellen	Zeit pro Teilproblem
4) Rekursion + Memoization oder Bottom-Up DP Tabelle aufbauen	Zeit = Zeit pro Teilproblem · #Teilprobleme
5) Löse ursprüngliches Problem	Benötigt evtl. zusätzliche Zeit

5 Schritte zur DP Lösung

5 Schritte	Fibonacci-Zahl F_n
1) Teilprobleme definieren	#Teilprobleme $= n$
2) Raten (Teil der Lösung)	nichts zu raten, #Möglichkeiten $= 1$
3) Rekursionsformel aufstellen	Zeit pro Teilproblem $= O(1)$
4) Rekursion + Memoization oder Bottom-Up DP Tabelle aufbauen	Zeit $=$ Zeit pro Teilproblem $\cdot$ #Teilprobleme $= O(1) \cdot n = O(n)$
5) Löse ursprüngliches Problem	Lösung ist Teilproblem F_n, Zeit $O(1)$

5 Schritte	Single Source Shortest Paths (Bellman-Ford)
1) Teilprobleme definieren	#Teilprobleme $= n \cdot (n-1)$ (alle $d_G^{(k)}(s,v)$)
2) Raten (Teil der Lösung)	$d_G^{(k)}(s,v)$: Kante zu v, #Mögl.: $1 +$ Eingangsgrad von v
3) Rekursionsformel aufstellen	Zeit pro Teilproblem $= \Theta\big(1 + \text{in\_degree}(v)\big)$
4) Rekursion + Memoization oder Bottom-Up DP Tabelle aufbauen	Zeit $= \sum_{\text{Teilprobleme}}$ Zeit pro Teilproblem $= \sum_{v \in V} \Theta\big(1 + \text{in\_degree}(v)\big) = \Theta(\lvert V\rvert \cdot \lvert E\rvert)$
5) Löse ursprüngliches Problem	Alle $d_G^{(n-1)}(s,v)$, Zeit $O(\lvert V\rvert)$

Berechnen der Lösung

UNI FREIBURG

Rekursive Berechnung der Optimierungsfunktion

- Alle Möglichkeiten werden (rekursive) durchprobiert
- Die beste (min/max) wird ausgewählt

Berechnen der Lösung

- Der rekursive Aufruf der Optimierungsfunktion gibt nur den optimalen Funktionswert zurück (z.B. Länge des kürzesten Pfades)
- Um die rekursive berechnete Lösung zu erhalten, muss man sich merken, welche der Möglichkeiten in jedem Schritt den optimalen Wert ergeben hat
- Wenn man DP mit der Hashfunktion macht, kann man sich das z.B. durch eine zweite/erweiterte Hashtabelle merken
- Bottom-Up: Man speichert in jede Zelle der Tabelle nicht nur den besten Wert, sondern auch wie man ihn erhalten hat

Lösung berechnen: Parent Pointers

UNI FREIBURG

Allgemeines DP

```
memo = {}
parent = {}
DP(x1, x2, …, xk):
    if (x1, x2, …, xk) in memo:
        return memo[(x1, x2, …, xk)]
    if (x1, x2, …, xk) in Basis
        value = …
    else:
        value = min/max des Wertes von DP(x1, x2, …, xk)
                über Vorgängerknoten (y1, y2, …, yk) im
                Abhängigkeitsgraphen (hängt von x1, …, xk,
                y1, …, yk, sowie DP(y1, y2, …, yk) ab
    memo[(x1, x2, …, xk)] = value
    parent[(x1, x2, …, xk)] = (y1, y2, …, yk)-Tupel, welches
                              das min/max erreicht hat
    return value
```

Blocksatz berechnen

Gegeben: Text T (Liste mit Wörtern)
Zeilenlänge W (#Zeichen pro Zeile)

Ziel: Text T mit Zeilenlänge W im Blocksatz ausgeben

- Man muss in Zeilen unterteilen und mit Leerzeichen auffüllen, damit jede Zeile (mit mind. 2 Wörtern) am Schluss mit einem Wort anfängt und aufhört und genau W Buchstaben hat.

Einfache Lösung: Greedy Algorithmus

- Text: bla, bla, bla, ha, bla, bla, sehrlangeswort, Zeilenlänge: 15

Greedy:

```
bla bla bla ha
bla         bla
sehrlangeswort
```

besser:

```
bla   bla  bla
bla   ha   bla
sehrlangeswort
```

Blocksatz, Qualität einer Lösung

- Wenn möglich sollten alle Wortabstände ähnlich gross sein, sehr grosse Abstände sollten wenn möglich vermieden warden

- Für jede Zeile definieren wir die *badness* als Mass, wie schlecht die Abstände sind (als Kosten der Zeile), z. B.
  - chars: #Buchstaben aller Wörter der Zeile
  - $words$: #Wörter der Zeile

$$\mathbf{badness} = \left(W - \mathbf{chars} - (\mathbf{words} - 1)\right)^3$$

- Die Gesamtkosten einer Aufteilung in Zeilen ist dann die Summe der *badness*-Werte aller Zeilen

- Diese Kosten sollen minimiert warden!

Blocksatz: Teilprobleme, Rekursion

- Wie kann man die Gesamtkosten rekursiv minimieren?

Blocksatz: Teilprobleme, Rekursion

UNI FREIBURG

Teilprobleme:

- $\text{blocksatz}(i)$: Kosten, wenn man mit dem i-ten Wort beginnt
  - Wort i ist am Anfang einer Zeile
  - Nur die Zeilen ab Wort i werden betrachtet

Rekursion:

$$\textbf{blocksatz}(n) = 0$$
$$\textbf{blocksatz}(i) = \min_{j>i} \textbf{badness}(i,j) + \textbf{blocksatz}(j)$$

Blocksatz: Dynamisches Programm

```
memo = {}; nextline = {}
blocksatz(i):
    if i in memo: return memo[i]
    if i >= n: cost = 0
    else:
        b = badness(i, i + 1)
        cost = ∞
        j = i + 1
        min_idx = i + 1
        while j <= n and b < ∞:
            c = b + blocksatz(j)
            if c < cost:
                cost = c; min_idx = j
            j += 1
            b = badness(i, j)
    memo[i] = cost; nextline[i] = min_idx
    return cost
```

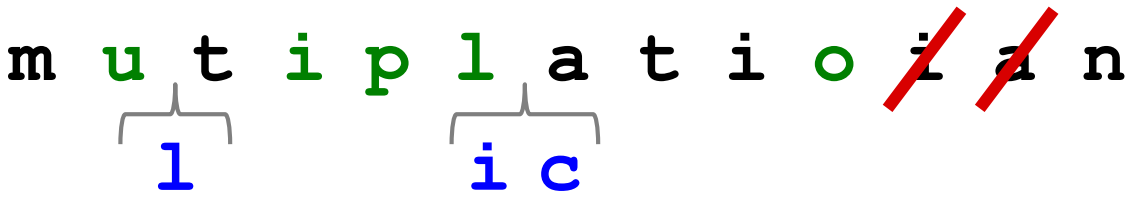
Editierdistanz

- For two given strings A and B, efficiently compute the

  edit distance $D(A,B)$ (# edit operations to transform A into B)

  as well as a minimum sequence of edit operations that transform A into B.

- **Example:** mathematician → multiplication:



Editierdistanz

Given: Two strings $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$

Goal: Determine the minimum number $D(A, B)$ of edit operations required to transform A into B

Edit operations:

a) **Replace** a character from string A by a character from B

b) **Delete** a character from string A

c) **Insert** a character from string B into A

```
m a - t h e m - - a t i c i a n
m u l t i p l i c a t i o - - n
```

Editierdistanz - Kostenmodell

- Cost for **replacing** character a by b: $\boldsymbol{c(a,b) \geq 0}$

- Capture insert, delete by allowing $a = \varepsilon$ or $b = \varepsilon$:
  - Cost for **deleting** character a: $\boldsymbol{c(a,\varepsilon)}$
  - Cost for **inserting** character b: $\boldsymbol{c(\varepsilon,b)}$

- **Triangle inequality**:

$$c(a,c) \leq c(a,b) + c(b,c)$$

→ each character is changed at most once!

- **Unit cost model**: $c(a,b) = \begin{cases} 1, & \text{if } a \neq b \\ 0, & \text{if } a = b \end{cases}$

Rekursive Struktur

- Optimal "alignment" of strings (unit cost model)

  `bbcadfagikccm` and `abbagflrgikacc`:

```
- b b c a g f a - g i k - c c m
a b b - a d f l r g i k a c c -
```

- Consists of optimal "alignments" of sub-strings, e.g.:

```
-bbcagfa      -gik-ccm
abb-adfl      rgikacc-
```

- Edit distance between $A_{1,m} = a_1 \dots a_m$ and $B_{1,n} = b_1 \dots b_n$:

$$D(A,B) = \min_{k,\ell}\{D(A_{1,k}, B_{1,\ell}) + D(A_{k+1,m}, B_{\ell+1,n})\}$$