

datastructuren en algoritmen

2011 10 06

college 10

## totnutoe

- **datastructuren: lineair en hierarchisch**  
stacks, queues, vectoren, lijsten, rijtjes, priority queues  
bomen, binaire bomen, binaire zoekbomen, AVL-bomen, heaps
- **sorteer-algoritmen**  
heap-sort, selection sort, insertion sort, merge sort, quick sort,  
quick select
- **hashing**
- **dynamic programming: geld wisselen, hout hakken,**  
**greedy algoritmen: geld wisselen, fractional knapsack**

## maximaal subarray: algoritme

**Algorithm** `maxSubarray(A, n)`:

**Input:** array  $A$  containing  $n$  integers

**Output:** maximum subarray sum

$B =$  new array of length  $n$

$B[0] = \max(A[0], 0)$

$max = B[0]$

**for**  $r = 1$  **to**  $n - 1$  **do**

$B[r] = \max(0, B[r - 1] + A[r])$

$max = \max(max, B[r])$

**done**

**return**  $max$

hier:  $\mathcal{O}(n)$

ook gezien: in  $\mathcal{O}(n^2)$  en in  $\mathcal{O}(n \log(n))$

# schema

- recap
- matrix vermenigvuldiging
- dynamic programming
- langste gemeenschappelijke deelrijtje
- knapsack 0/1
- task scheduling
- grafen
- kortste pad

# schema

- recap
- **matrix vermenigvuldiging**
- dynamic programming
- langste gemeenschappelijke deelrijtje
- knapsack 0/1
- task scheduling
- grafen
- kortste pad

# matrix vermenigvuldiging

input: matrix  $A$  met  $p$  rijen en  $q$  kolommen  
en matrix  $B$  met  $q$  rijen en  $r$  kolommen

**Algorithm** matrix – vermenigvuldiging( $A, B$ ):

**new** matrix  $C$  :  $p \times r$

**for**  $i := 1$  **to**  $p$  **do**

**for**  $j := 1$  **to**  $r$  **do**

$c_{ij} := 0$

**for**  $k := 1$  **to**  $q$  **do**

$c_{ij} := c_{ij} + a_{ik}b_{kj}$

tijdscomplexiteit wordt bepaald door  $pqr$

# matrix vermenigvuldiging

- matrix vermenigvuldiging is associatief dwz  $(AB)C = A(BC)$   
je mag de haakjes zetten zoals je wilt  
en dit bepaalt de complexiteit deels

## hoe zetten we de haakjes?

- het aantal elementaire stappen hangt af van hoe de haakjes staan

$$A = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad B = ( 1 \quad 1 \quad 1 ) \quad C = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

- $(AB)C$  levert  $6 + 6$  vermenigvuldigingsstappen,  
 $A(BC)$  levert  $2 + 3$  vermenigvuldigingsstappen
- hoe vinden we een beste haakjes-configuratie?

## matrix vermenigvuldiging: opsommen

- bekijk alle haakjes-mogelijkheden
- bereken per mogelijkheid het aantal operaties
- kies dan een beste haakjes-configuratie

### MAAR

- **duur:**  
aantal mogelijkheden is Catalan nummer van aantal matrices  
exponentieel, bijna  $4^n$

## matrix vermenigvuldigen: greedy

- kies steeds product met minimaal aantal operaties
- levert niet perse optimale oplossing

$$A = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad B = ( 1 \quad 1 ) \quad C = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

- dan  $AB$  kost 4 en  $BC$  kost 6
- toch is  $A(BC)$  met totaal 12 beter dan  $(AB)C$  met totaal 16

# matrix vermenigvuldigen: dynamic programming

- **probleem:**  
geef beste haakjes voor  $A_0 \cdot \dots \cdot A_{n-1}$   
dimensie van  $A_i$  is  $d_i \times d_{i+1}$
- $N_{i,j}$ :  
beste haakjes voor  $A_i \cdot \dots \cdot A_j$
- **voorbeeld:**  
stel de beste haakjes  
 $(A_0 \cdot \dots \cdot A_i) \cdot (A_{i+1} \cdot \dots \cdot A_{n-1})$   
dan is het aantal operates  $N_{0,i} + N_{i+1,n-1} + 1$
- **definitie  $N_{i,j}$ :**

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i \cdot d_{k+1} \cdot d_{j+1}\}$$

- **uitleg dimensies:**  
 $A_i \cdot \dots \cdot A_k : d_i \times d_{k+1}$   
 $A_{k+1} \cdot \dots \cdot A_j : d_{k+1} \times d_{j+1}$

## matrix vermenigvuldigen: algoritme

eerst  $N_{i,i} = 0$ , dan  $N_{i,i+s-1}$

matrixChain( $d_0, d_1, \dots, d_{n-1}$ ):

for  $i = 0$  to  $n - 1$  do  $N[i, i] = 0$  done

for  $s = 2$  to  $n$  do

for  $i = 0$  to  $n - s$  do

$j = i + s - 1$

$N[i, j] = +\infty$

for  $k = i$  to  $j - 1$  do

$ops = N[i, k] + N[k + 1, j] + d_i \cdot d_{k+1} \cdot d_{j+1}$

$N[i, j] = \min(N[i, j], ops)$

done

done

done

## toepassing

- $d_0 = 2, d_1 = 1, d_2 = 2, d_3 = 4, d_4 = 3.$
- $A_0$  is  $2 \times 1, A_1$  is  $1 \times 2, A_2$  is  $2 \times 4, A_3$  is  $4 \times 3.$
- $N[0, 0] = 0, N[1, 1] = 0, N[2, 2] = 0, N[3, 3] = 0$
- $N[0, 1] = 4, N[1, 2] = 8, N[2, 3] = 24$
- $N[0, 2] = \min(0 + 8 + 8, 4 + 0 + 16) = 16,$   
 $N[1, 3] = \min(0 + 24 + 6, 8 + 0 + 12) = 20$
- $N[0, 4] = \min(0 + 20 + 6, 4 + 24 + 12, 16 + 0 + 24) = 26$
- optimaal:  $A_0 \cdot ((A_1 \cdot A_2) \cdot A_3).$

# schema

- recap
- matrix vermenigvuldiging
- **dynamic programming**
- langste gemeenschappelijke deelrijtje
- knapsack 0/1
- task scheduling
- grafen
- kortste pad

## dynamic programming: kenmerken

- geef structuur van een optimale oplossing
- geef recursief de waarde van een optimale oplossing
- bereken de waarde van een optimale oplossing
- construeer een optimale oplossing

het werkt niet altijd

geef in graaf kortste pad van  $u$  naar  $v$

dan: een deelpad van een kortste pad is zelf kortste

geef in graaf langste cykel-vrije pad van  $u$  naar  $v$

dan: een deelpad van een cykel-vrij langste pad is zelf niet perse langste

## memoization

- **vaak**: bottom-up zoals bij houthakken
- **alternatief**: top-down met memoization:  
we onthouden suboplossingen

# langste gemeenschappelijke deelrijtje

- **bijvoorbeeld in bioinformatica:**  
vergelijk twee DNA-rijtjes en bepaal hoe 'similar' ze zijn kan door langste gemeenschappelijke deelrijtje te bepalen
- **deelrijtje:** wordt verkregen door elementen weg te laten
- **voorbeeld:**  
[A, B, C, B, D, A, B] en [B, D, C, A, B, A]  
hebben langste gemeenschappelijke deelrijtje [B, C, B, A]

## algoritme: voorwerk

- **input:**  $X = [x_1, \dots, x_m]$  en  $Y = [y_1, \dots, y_n]$
- **brute-force:** bekijk elk deelrijtje van  $X$ , in  $\mathcal{O}(2^m)$ .
- **LCS = longest common subsequence**
- **notatie:**  $X_i = [x_1, \dots, x_i]$  en  $c[i, j]$  lengte van een LCS van  $X_i$  en  $Y_j$

## algorithme

**Algorithm** LCS( $X, Y$ ):

**new** array  $C[0 \dots m, 0 \dots n]$

**for**  $i := 0$  **to**  $m$  **do**

$C[i, 0] := 0$

**for**  $j := 0$  **to**  $n$  **do**

$C[0, j] := 0$

**for**  $i := 1$  **to**  $m$  **do**

**for**  $j := 1$  **to**  $n$  **do**

**if**  $x_i = y_j$  **then**

$C[i, j] := C[i - 1, j - 1] + 1$

**else**

$C[i, j] := \max(C[i, j - 1], C[i - 1, j])$

**return**  $C[m, n]$

# schema

- recap
- matrix vermenigvuldiging
- dynamic programming
- langste gemeenschappelijke deelrijtje
- knapsack 0/1
- task scheduling
- grafen
- kortste pad

## knapsack 0/1: voorbeeld

- 7 kilo goudstuk
- 5 kilo goudstuk
- 4 kilo goudstuk
- rugzak voor 10 kilo! wat nu?
- de greedy aanpak levert 7 dus niet optimaal

# knapsack 0/1: probleem

- **gegeven:**  
verzameling  $S$  met  $n$  items  
elk item  $i$  heeft gewicht  $w_i$  en benefit  $b_i$   
maximaal totaalgewicht  $W$
- **doel:**  
kies items  $T \subseteq S$  zodat  
 $\sum_{i \in T} b_i$  maximaal  
onder constraint  $\sum_{i \in T} w_i \leq W$
- **naieve aanpak:**  
bekijk alle  $2^n$  deelverzamelingen van  $S$  (hmm)
- **greedy werkt niet (ook niet met gewogen waarden)**

## knapsack 0/1: idee algoritme

- $S_k$  bevat elementen  $1, \dots, k$
- $B[k, w]$  is beste selectie uit  $S_k$  met totaalgewicht  $w$
- hoe vind je  $B[k, w]$ ?

als  $w_k > w$ : item  $k$  kan er niet bij  
dus  $B[k, w] = B[k - 1, w]$

als  $w_k \leq w$ : item  $i$  kan (maar hoeft niet) erbij  
dan  $B[k, w] = \max\{B[k - 1, w], B[k - 1, w - w_k] + b_k\}$

## knapsack 0/1: idee algoritme

- voor elke  $k = 0, 1, \dots, n$  bekijken we  $S_k$
- voor elke  $S_k$  bekijken we  $w = 0, 1, \dots, W$
- in  $\mathcal{O}(nW)$   
stel  $W = 2^n$  dan niet zo gunstig
- een pseudo-polynomiaal algoritme-idee
- wordt onwaarschijnlijk geacht dat iemand een polynomiaal algoritme vindt

## knapsack 0/1: voorbeeld toepassing

- item 1 met  $w_1 = 3$  en  $b_1 = 9$
- item 2 met  $w_2 = 2$  en  $b_2 = 5$
- item 3 met  $w_3 = 2$  en  $b_3 = 5$
- maximaal totaalgewicht  $W = 4$

we runnen het algoritme:

- $B[0,0] = 0$ ,  $B[0,1] = 0$ ,  $B[0,2] = 0$ ,  $B[0,3] = 0$ ,  $B[0,4] = 0$
- $B[1,0] = 0$ ,  $B[1,1] = 0$ ,  $B[1,2] = 0$ ,  $B[1,3] = 9$ ,  $B[1,4] = 9$
- $B[2,0] = 0$ ,  $B[2,1] = 0$ ,  $B[2,2] = 5$ ,  $B[2,3] = 9$ ,  $B[2,4] = 9$
- $B[3,0] = 0$ ,  $B[3,1] = 0$ ,  $B[3,2] = 5$ ,  $B[3,3] = 9$ ,  $B[3,4] = 10$

# schema

- recap
- matrix vermenigvuldiging
- dynamic programming
- langste gemeenschappelijke deelrijtje
- knapsack 0/1
- **task scheduling**
- grafen
- kortste pad

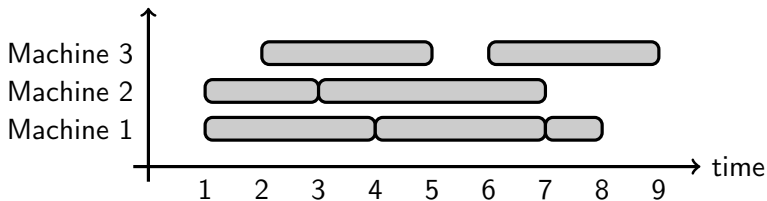
## task scheduling: voorbeeld

doe alle taken op zo min mogelijk machines

taken:

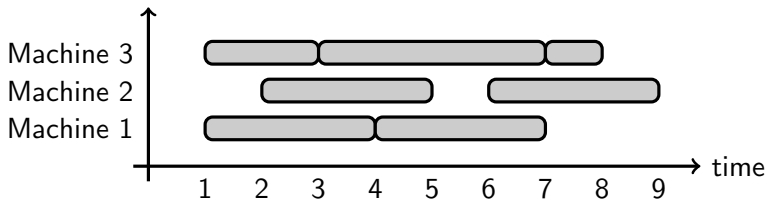
$[1, 4]$ ,  $[1, 3]$ ,  $[2, 5]$ ,  $[3, 7]$ ,  $[4, 7]$ ,  $[6, 9]$ ,  $[7, 8]$

indeling:



# task scheduling

- gegeven: verzameling  $T$  van taken met begin- en eind-tijd
- schedule alle taken op zo min mogelijk machines



## task scheduling: algoritme

- greedy choice:  
kies steeds taak met kleinste begin-tijd
- neem alleen nieuwe machine erbij als echt nodig

`taskSchedule( $T, \vec{s}, \vec{f}$ ):`

`$m = 0$     number of machines`

`sort  $T$  such that elements are ascending w.r.t.  $s_i$`

**while**  `$\neg T.isEmpty()$`  **do**

`$i = T.remove(T.first())$`

**if** a machine  `$j \leq m$`  has time for task  `$i$`  **then**

`schedule  $i$  on machine  $j$`

**else**

`$m = m + 1$`

`schedule  $i$  on machine  $m$`

**done**

## task scheduling: correctheid algoritme

- stel algoritme gebruikt  $m$  machines,
- er is een schedule met minder dan  $m$  machines
- bekijk toekenning van machine  $m$  door algoritme
- laat geen ruimte voor alternatieven

# scheduling tijd

- $n$  klanten bij postkantoor, elk met geholpen-tijd  $t_i$
- doel: minimaliseer aanwezig-tijd van alle klanten samen
- hoeveel verschillende volgordes zijn er?
- geef greedy algoritme dat een beste volgorde geeft

# scheduling met deadlines

elke taak  $i$  heeft:

- een deadline  $d_i$
- een benefit  $b_i$
- duur 1

vraag: geef een uitvoerbare verzameling taken met maximale totale benefit

# schema

- recap
- matrix vermenigvuldiging
- dynamic programming
- langste gemeenschappelijke deelrijtje
- knapsack 0/1
- task scheduling
- grafen
- kortste pad

## gewogen grafen: definitie

- $G = (V, E)$

$G$ : de graaf

$V$ : verzameling knopen (vertices)

$E$ : verzameling paren van knopen (kanten, edges)  
(kan met of zonder richting)

- $w : E \rightarrow \mathbb{R}^+$

elke kant  $e$  heeft een positief gewicht  $w(e)$

## gebruik van grafen

- circuits
- networks
- google maps
- ...x

# implementatie van een graaf

Vertex stores:

- element

Edge stores:

- element
- start-point and end-point

## Edge List Structure

Graph is stored as list of vertexes and list of edges.

## Adjacency List Structure

Graph is stored as list of vertexes. Each vertex stores:

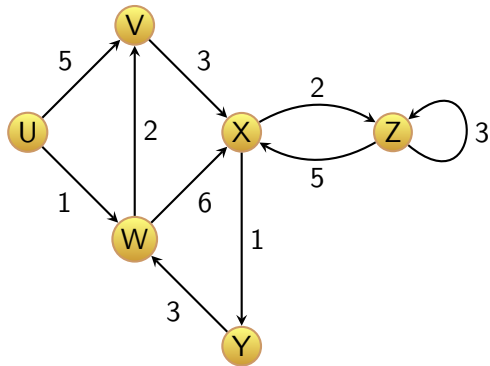
- element
- list of (outgoing and incoming) edges

# implementatie van een graaf

graaf met  $n$  knopen en  $m$  kanten

	Edge List	Adjacency List
<code>outgoingEdges(<math>v</math>)</code>	$\mathcal{O}(m)$	$\mathcal{O}(\text{deg } v)$
<code>areAdjacent(<math>v, w</math>)</code>	$\mathcal{O}(m)$	$\mathcal{O}(\min(\text{deg } v, \text{deg } w))$
<code>insertVertex(<math>o</math>)</code>	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<code>insertEdge(<math>v, w, o</math>)</code>	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<code>removeVertex(<math>v</math>)</code>	$\mathcal{O}(m)$	$\mathcal{O}(\text{deg } v)$
<code>insertEdge(<math>e</math>)</code>	$\mathcal{O}(1)$	$\mathcal{O}(1)$

# gewogen grafen



# schema

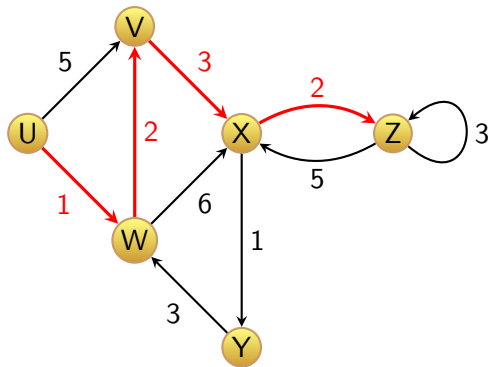
- recap
- matrix vermenigvuldiging
- dynamic programming
- langste gemeenschappelijke deelrijtje
- knapsack 0/1
- task scheduling
- grafen
- kortste pad

## kortste pad

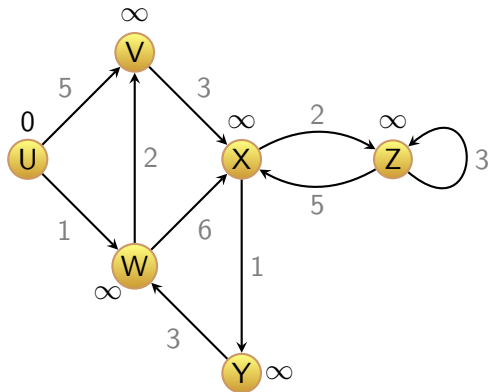
- **input:**  
graaf  $G = (V, E)$  met  $w : E \rightarrow \mathbb{R}^+$ , en knoop  $v \in V$
- **afstand:**  
 $d(v, u)$  is het kleinste totaalgewicht van een pad van  $v$  naar  $u$   
lengte kortste pad
- **output:**  
 $d(v, u)$  voor alle  $u \in V$

## kortste pad: voorbeeld

van  $U$  naar  $Z$  met gewicht 8



## Dijkstra's kortste pad algoritme voorbeeld



# Dijkstra's kortste pad algoritme

- **input:**

$$G = (V, E)$$

$$w : E \rightarrow \mathbb{R}^+$$

$$v \in V$$

- **output:**

$$D[u] = d(v, u) \text{ voor alle } u \in V$$

- **init:**

$$D[v] = 0$$

$$D[u] = \infty \text{ voor alle } u \neq v$$

doe alles in priority queue  $Q$

- **while:**

$Q \neq \emptyset$  do

neem en verwijder  $u \in Q$  met  $D[u]$  minimaal

update: voor alle burens  $z$  van  $u$  doe

als  $D[u] + w(u, z) < D[z]$  dan  $D[z] = D[u] + w(u, z)$

- **return:**

$D[u]$  voor alle  $u \in V$

# Dijkstra's Algorithm

DijkstraDistances( $G, s$ ):

$Q$  = new heap-based priority queue

(set all distances to  $\infty$  except for  $s$  with distance 0)

**for each**  $v \in G.vertices()$  **do**

**if**  $v == s$  **then**  $v.setDistance(0)$  **else**  $v.setDistance(\infty)$

$h = Q.insert(v.getDistance(), v)$  (returns heap node)

$v.setHeapNode(h)$

**while**  $\neg Q.isEmpty()$  **do**

$v = Q.removeMin()$

    (update the distance of all neighbours)

**for each**  $e \in v.outgoingEdges()$  **do**

$w = e.endPoint()$

$d = v.getDistance() + e.getWeight()$

**if**  $d < w.getDistance()$  **then**

$w.setDistance(d)$

$Q.replaceKey(w.getHeapNode(), d)$

**done**

## kortste pad implementatie

- priority queue  $Q$  als heap
- graaf  $G$  met adjacency list  
lijst met knopen,  
per knoop inkomende en uitgaande kanten  
daardoor: geef alle buren van  $u$  in  $\mathcal{O}(\text{deg}(u))$

## kortste pad: complexiteit

- $n$  knopen en  $m$  kanten
- **init:**  
 $n$  keer in  $\log(n)$  dus in  $\mathcal{O}(n \log n)$
- **while:**  
 $n$  keer verwijder kleinste in  $\log n$  dus in  $\mathcal{O}(n \log n)$

geef buren in  $\mathcal{O}(\deg(u))$  en update  $z$  in  $\mathcal{O}(\log n)$   
dus in  $\sum \deg(u_i) \log n$

NB:  $\sum \deg(u_i) = 2m$  (Thm 6.6)

dus while in  $\mathcal{O}((n + m) \log n)$

samenhangend dus  $m \geq n - 1$  dus  $\mathcal{O}(m \log n)$

## kortste pad: correctheid

**B:** als  $u$  aan cloud wordt toegevoegd dan  $D[u] = d(v, u)$

**bewijs:**

Stel niet. Bekijk de eerste  $u$  met  $D[u] > d(v, u)$ . Dan: er is een kortste pad  $P : v \rightarrow^* y \rightarrow z \rightarrow^* u$  met  $|P| = d(v, u)$ , en  $z$  de eerste knoop niet in de cloud.

We gaan een tegenspraak afleiden. We gebruiken:

$$D[z] \leq D[y] + w(y, z) = d(v, y) + w(y, z) = (!)d(v, z).$$

$$\begin{aligned} D[u] &\leq D[z] && \text{anders was } z \text{ gekozen} \\ &= d(v, z) && \text{zie boven} \\ &\leq d(v, u) \\ &< D[u] \end{aligned}$$

Tegenspraak. Dus aanname 'stel niet' onwaar. Dus  $B$  waar.

# schema

- recap
- matrix vermenigvuldiging
- dynamic programming
- langste gemeenschappelijke deelrijtje
- knapsack 0/1
- task scheduling
- grafen
- kortste pad
- **materiaal**

## materiaal

- boek 5.3, 5.1
- boek 6.2
- boek 7.1.1
- zie ook boek 9,4 voor LCS

## extra materiaal

- voor integer vermenigvuldiging