Tiende college algoritmiek

14 april 2011

Gretige algoritmen

- Greed = hebzucht
- Voor oplossen van optimalisatieproblemen
- Oplossing wordt stap voor stap opgebouwd
- In elke stap wordt een gretige keuze gemaakt waarmee de huidige deeloplossing wordt uitgebreid
- Dat wil zeggen: een (locale) keuze die op dat moment de beste lijkt (de grootste directe winst oplevert)
- De vraag is of dat leidt tot een globaal optimale oplossing

De oplossing wordt dus opgebouwd via een serie achtereenvolgende gretige keuzes. Deze keuzes

- zijn consistent met de restricties van het probleem
- zijn locaal optimaal, d.w.z. de best uitziende keuze in die stap
- zijn onherroepelijk: keuzes kunnen niet meer worden teruggedraaid

Een gretig algoritme ziet er dus ruwweg zo uit:

> **while** nog niet alle stappen zijn gedaan **do**
>
> doe een keuze die in eerste instantie de grootste winst lijkt op te leveren
>
> **od**

Uitbreiden van deeloplossingen moet uiteraard wel steeds in overeenstemming met de geldende restricties.

Soms leveren gretige algoritme een optimale oplossing, en soms/vaak niet. In dat geval is de gretige strategie een heuristiek, die bijvoorbeeld leidt tot een goede, maar meestal niet optimale oplossing. Of: de gretige strategie leidt vaak, maar niet altijd, tot een optimale oplossing.

Gegeven onbeperkt veel munten van $d_1, d_2, \ldots d_m$ eurocent, en een te betalen bedrag van n ($n \geq 0$) eurocent. Alle d_i zijn > 0 en verschillend.

Gevraagd: het minimale aantal munten dat nodig is om het bedrag van n eurocent te betalen.

Voorbeeld:

type munt	waarde
1	1
2	4
3	6

te betalen bedrag: 8

Vier manieren om te betalen: 6 + 1 + 1; 4 + 4; 4 + 1 + 1 + 1 + 1; 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1. Dus het gevraagde minimale aantal is: 2 (twee munten van 4 cent).

Laat $\texttt{munt}[i][j]$ het minimale aantal munten zijn dat nodig is om een bedrag van j eurocent te betalen, wanneer alleen munten van $d_1, d_2, \ldots, d_i$ $(i \geq 1)$ worden gebruikt. We zoeken dus $\texttt{munt}[m][n]$.

Dan geldt (want d_i wordt wel of niet gebruikt):

$$\texttt{munt}[i][j] = \begin{cases} \min\{\texttt{munt}[i-1][j], 1 + \texttt{munt}[i][j-d_i]\} & \text{als } i > 1, j \geq d_i \\ \texttt{munt}[i][j] = \texttt{munt}[i-1][j] & \text{als } i > 1, 0 < j < d_i \\ \infty & \text{als } i = 1, 0 < j < d_1 \\ 1 + \texttt{munt}[1][j-d_1] & \text{als } i = 1, j \geq d_1 \\ 0 & \text{als } i \geq 1, j = 0 \end{cases}$$

Laat $\texttt{munt}[j]$ het minimale aantal munten zijn dat nodig is om een bedrag van j eurocent te betalen. We zoeken dus $\texttt{munt}[n]$.

Dan geldt:

$$\texttt{munt}[j] = \begin{cases} 0 & \text{als } j = 0 \\ \infty & \text{als } j > 0 \text{ en } j < d_1, \ldots, j < d_m \\ \min_{d_i \leq j} \{1 + \texttt{munt}[j - d_i]\} & \text{anders} \end{cases}$$

Als $d_1, \ldots, d_m$ oplopend gesorteerd zijn, kun je gemakkelijk nagaan welke d_i's kleiner-gelijk j zijn.

Een gretige strategie (recursief geformuleerd):

betaal n met $d_1, \ldots d_m$ (voor het gemak oplopend gesorteerd)::

if ($n = 0$) klaar;

else geef de grootste munt $d_i \leq n$ (restrictie);

// dan is het nog te betalen bedrag zo klein mogelijk

// en dus heb je zo weinig mogelijk munten

// nodig (hoop je)

betaal $n - d_i$ met $d_1, \ldots d_i$.

Bovenstaand algoritme is erg eenvoudig en snel, en levert voor het geval van de gebruikelijke euro-munten (munten van 1, 2, 5, 10, 20, 50, 100, 200 eurocent) het optimale antwoord. Dit is echter niet het geval voor de muntwaarden uit het voorbeeld. (Bovendien gaat dit algoritme ervan uit dat het bedrag te betalen is. Anders is nog een kleine aanpassing nodig.)

- Optimale oplossing
  - Muntenprobleem voor de gebruikelijke euromunten
  - Sommige planningsproblemen
  - Kortste paden in een graaf (Dijkstra)
  - Minimale opspannende boom (Prim, Kruskal; zie Datastructuren)
  - ....
- Benadering
  - Handelsreizigersprobleem
  - Knapzakprobleem
  - ....

Zie ook Levitin, Exercise 9.1.3.

Gegeven n jobs, genummerd 1 t/m n, die allemaal na elkaar door één processor moeten worden uitgevoerd. Van elke job i is de executietijd t_i gegeven. De jobs moeten zo na elkaar worden gepland dat de totale hoeveelheid tijd in het systeem van alle jobs samen wordt geminimaliseerd. De tijd die job i in het systeem doorbrengt is de wachttijd + de executietijd t_i.

We willen dus

$$T = \sum_{i=1}^{n} \texttt{(tijd die job i in het systeem doorbrengt)}$$

minimaliseren.

De waarde van T hangt af van de volgorde waarin de jobs door de processor worden uitgevoerd.

Voorbeeld: $n = 3$; Jobs: $1, 2, 3$ met $t_1 = 5, t_2 = 10, t_3 = 3$.

volgorde	T
1 2 3	$5 + (5 + 10) + (5 + 10 + 3) = 38$
1 3 2	$5 + (5 + 3) + (5 + 3 + 10) = 31$
2 1 3	$10 + (10 + 5) + (10 + 5 + 3) = 43$
2 3 1	$10 + (10 + 3) + (10 + 3 + 5) = 41$
3 1 2	$3 + (3 + 5) + (3 + 5 + 10) = 29$
3 2 1	$3 + (3 + 10) + (3 + 10 + 5) = 34$

Idee voor een gretige oplossing: kies in elke stap de job met de kleinste executietijd van de nog resterende jobs. Door die keuze houd je de wachttijd voor de overige jobs op dat moment (tot de volgende job aan de beurt is) zo klein mogelijk.

Voor het voorbeeld levert deze gretige strategie de optimale oplossing.

Observatie.
Stel dat de jobs in de volgorde $i_1, i_2, \ldots, i_n$ worden uitgevoerd. Dan is

$$T = t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \ldots + (t_{i_1} + t_{i_2} + \ldots + t_{i_n})$$

$$= n * t_{i_1} + (n-1) * t_{i_2} + \ldots + 2 * t_{i_{n-1}} + t_{i_n}$$

Algoritme

Sorteer de jobs in oplopende volgorde van hun executietijd;
// Dit is de optimale volgorde om de jobs uit te voeren

Complexiteit

Sorteren kan in $O(n \lg n)$ stappen, dus dit algoritme ook.

Correctheid

Dit algoritme levert altijd een optimale oplossing. Dit moet wel expliciet bewezen worden.
Hiertoe laten we het volgende zien. Stel dat de volgorde van uitvoering zo is dat er twee jobs $a := i_k$ en $b := i_{k+1}$ zijn met $t_a > t_b$ (dus b wordt direct na a uitgevoerd maar heeft kortere executietijd). Dan krijg je een betere oplossing door de volgorde van deze twee jobs om te keren.

Activiteiten-selectie probleem

Gegeven een verzameling $A = \{1, 2, \ldots, n\}$ met activiteiten die allemaal gebruik willen maken van een of andere "resource" (bijvoorbeeld een collegezaal). Deze resource kan maar door één activiteit tegelijk gebruikt worden. Activiteit i vindt plaats gedurende het tijdsinterval $[b_i, e_i)$. De begin- en eindtijden b_i en e_i zijn voor alle activiteiten bekend.

Definitie: activiteiten i en j heten compatibel als $[b_i, e_i)$ en $[b_j, e_j)$ niet overlappen, dus als $e_i \leq b_j$ of $e_j \leq b_i$.

Opdracht: vind een zo groot mogelijke deelverzameling van paarsgewijs compatibele activiteiten uit A.

Gretig algoritme: in elke stap

- wordt een activiteit i gekozen die compatibel is met de reeds gekozen activiteiten en die op dat moment het beste lijkt (gretige keuze)

Mogelijke gretige keuzes voor het kiezen van activiteit i:

1. selecteer steeds de activiteit die het eerst begint
2. selecteer steeds de activiteit die het kortste duurt
3. selecteer steeds de activiteit die overlapt met zo min mogelijk andere activiteiten
4. selecteer steeds de activiteit die het eerst eindigt

Welke gretige keuzes voor het kiezen van activiteit i leiden altijd tot een optimale oplossing?

1. selecteer de activiteit die het eerst begint: werkt niet
2. selecteer de activiteit die het kortste duurt: werkt niet
3. selecteer de activiteit die overlapt met zo min mogelijk andere activiteiten: werkt niet
4. selecteer de activiteit die het eerst eindigt: werkt wel

```
// neem aan dat A oplopend gesorteerd is op eindtijd e_i,
// anders eerst even sorteren: O(n lg n)
A' := {1};
j := 1;
// de laatst aan A' toegevoegde activiteit
for i := 2 to n do
// loop activiteiten af in volgorde van eindtijd
    if i is compatibel met A' then (*)
       A' := A' ⋃ {i}; j := i;
    fi
od
// A' bevat nu een optimale paarsgewijs
// compatibele deelverzameling van A
```

(*) Merk op: i is compatibel met A' als hij compatibel is met de laatst toegevoegde activiteit.

Dus `(*)` wordt: `if` $b_i \geq e_j$ `then`

Correctheid: moet bewezen worden
Complexiteit: $O(n)$ als A reeds gesorteerd is

i	b_i	e_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

Optimale oplossing: $\{1, 4, 8, 11\}$

De **correctheid** van het gretige algoritme volgt uit de volgende twee observaties:

1. Er bestaat een optimale oplossing die met activiteit 1 begint (die met de kleinste eindtijd dus).

2. Stel A' is een optimale oplossing van het oorspronkelijke probleem, dus met activiteitenverzameling A, die 1 bevat. Dan is $B' = A' \setminus \{1\}$ een optimale oplossing van het probleem met activiteitenverzameling $B = \{i \in A : b_i \geq e_1\}$.

Gegeven een graaf G met gewichten op de takken, en een beginknoop s. We nemen aan dat alle gewichten ≥ 0 zijn.

Gevraagd: voor elke willekeurige knoop v in de graaf (de lengte van) het/een kortste pad van s naar v.

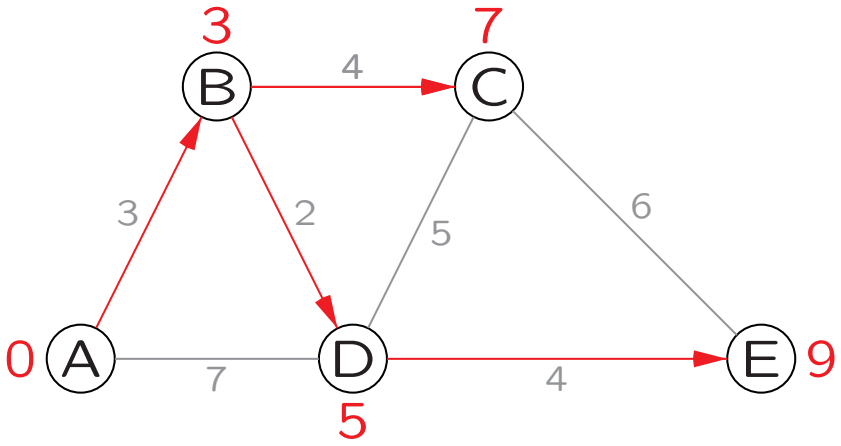
Merk op dat al deze kortste paden vanuit s samen een boomstructuur vormen.

Oplossing: het algoritme van Dijkstra is een gretig algoritme, dat de kortste paden van s naar elk van de andere knopen vindt in volgorde van hun lengte. In elke stap wordt een knoop (en daarmee het pad van s naar die knoop) gekozen die het dichtst bij de reeds gekozen knopen ligt.

Dijkstra: Edsger W. Dijkstra (1930-2002)

Edsger Wybe Dijkstra (Rotterdam, 11 mei 1930 — Nuenen, 6 augustus 2002) was een Nederlandse wiskundige en informaticus die veel voor de informatica heeft gedaan, met name op het gebied van gestructureerd programmeren. In 1972 werd hij onderscheiden met de Turing Award.

Voorbeeld kortste-paden-boom



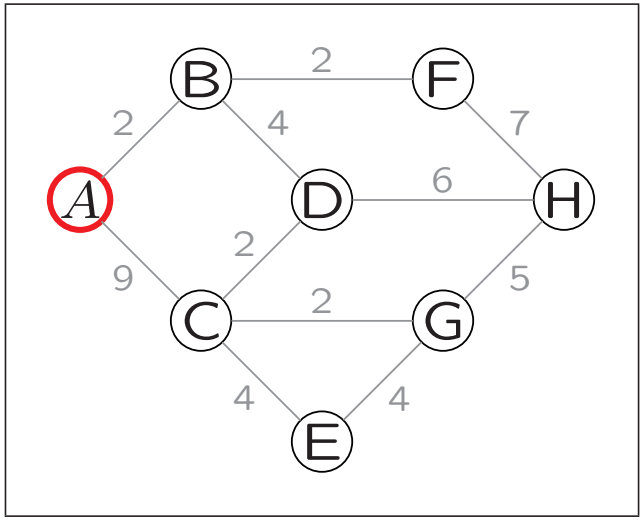
De kortste paden vanuit A zijn:

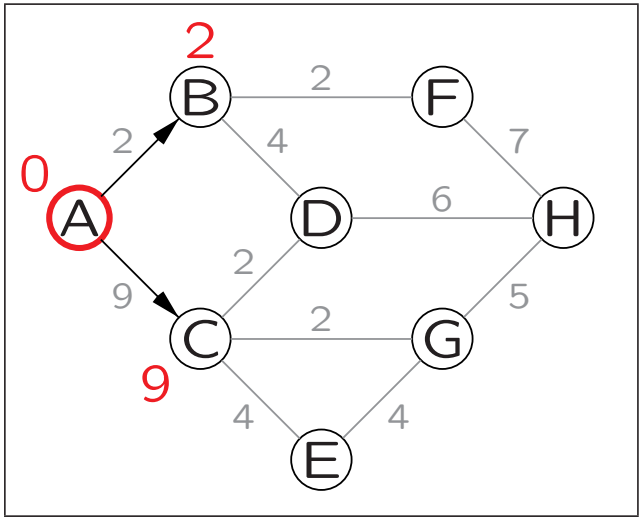
A $\longrightarrow$ B: lengte 3
A $\longrightarrow$ B $\longrightarrow$ D: lengte 5
A $\longrightarrow$ B $\longrightarrow$ C: lengte 7
A $\longrightarrow$ B $\longrightarrow$ D $\longrightarrow$ E: lengte 9

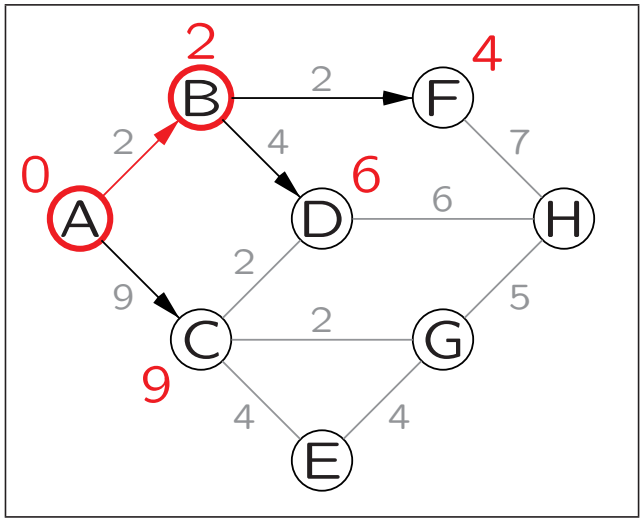
// invoer: samenhangende gewogen graaf $G = (V, E)$ en startknoop s
// uitvoer: array dat de lengtes van de kortste paden vanuit s bevat;
// na afloop is pad[v] = de lengte van een kortste pad van s naar v
for $v \in V$ **do**
    pad[v] := ∞; **od**
pad[s] := 0;
$U := \emptyset$;
while ($U \neq V$) **do**
    vind knoop $v^* \in V \setminus U$ met pad[v^*] minimaal;
    $U := U \cup \{v^*\}$;
    for alle knopen v aangrenzend aan v^* **do**
        if pad[v^*]+ gewicht(v^*, v) $<$ pad[v] **then**
            pad[v] := pad[v^*]+ gewicht(v^*, v);
        fi
    od
od

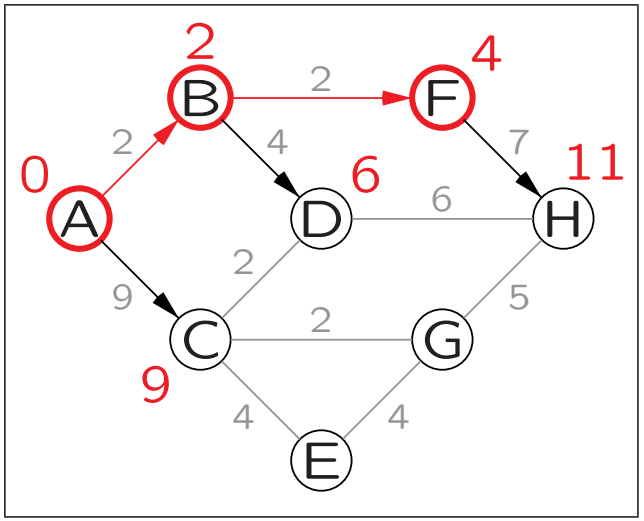
- In het algoritme bevat U steeds alle knopen waarvan de definitieve kortste afstand vanaf s reeds bepaald is. Voor deze knopen geeft het label pad[v] al de definitieve afstand aan. Moet bewezen worden.
- Voor de andere knopen w geeft pad[w] steeds de kortste afstand vanuit s naar w via uitsluitend knopen uit U. Moet bewezen worden.
- De volgende dichtstbijzijnde knoop wordt gekozen uit de knopen uit $V \setminus U$ die direct grenzen aan U. Nadat deze gekozen is worden de labels aangepast.
- In elke stap wordt zo voor elke knoop uit $V \setminus U$ de kortste afstand vanaf s via de reeds afgehandelde knopen uit U (en bijbehorende takken) bepaald.
- Het is niet zo moeilijk dit algoritme aan te passen zodat ook de kortste paden zelf worden berekend.

Voorbeeld -1-

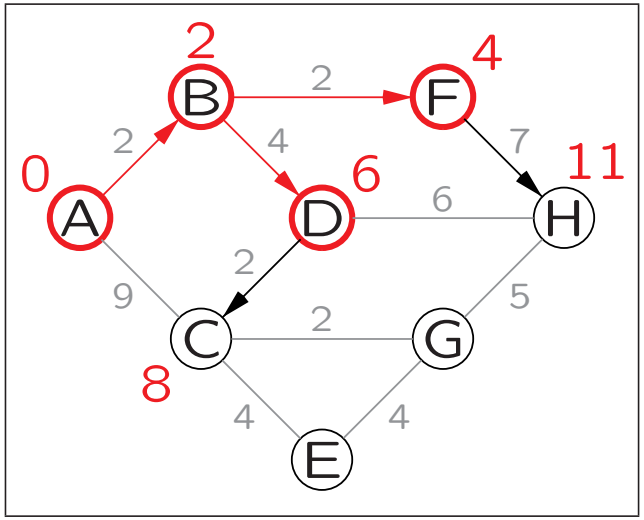

A
B
C
D
E
F
G
H
2
2
4
7
6
2
9
2
5
4
4

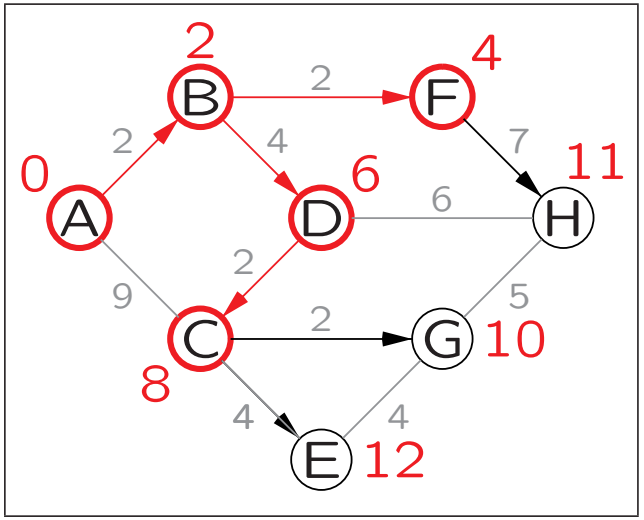

0
2
9
A
B
C
D
E
F
G
H
2
2
4
7
6
2
2
5
4
4

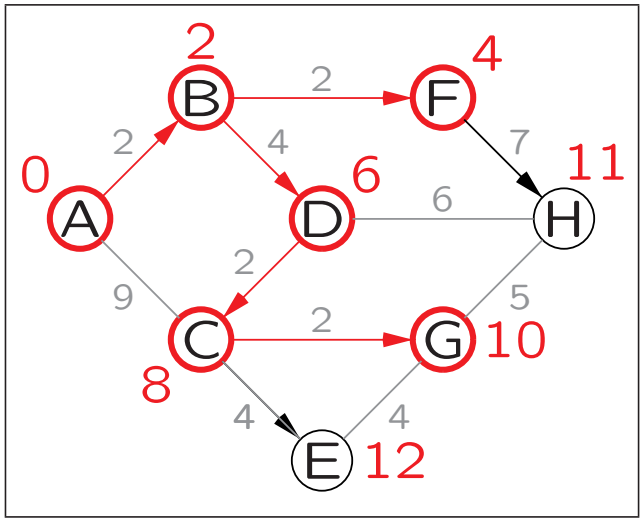

0
2
9
6
4
A
B
C
D
E
F
G
H
2
2
4
7
6
2
2
5
4
4

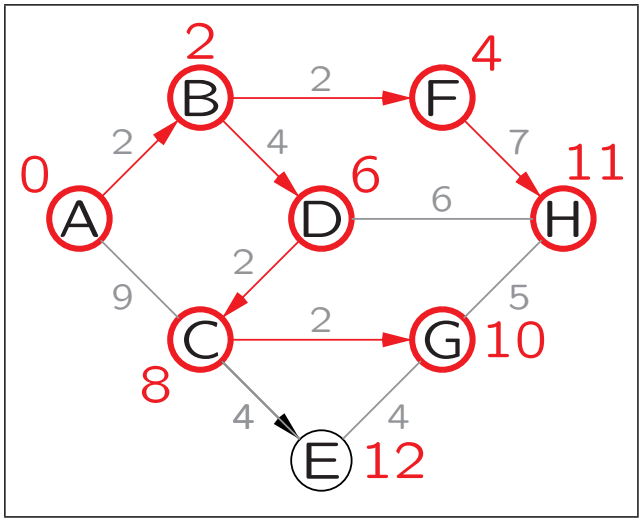

0
2
9
6
4
11
A
B
C
D
E
F
G
H
2
2
4
7
6
2
2
5
4
4

Voorbeeld -2-

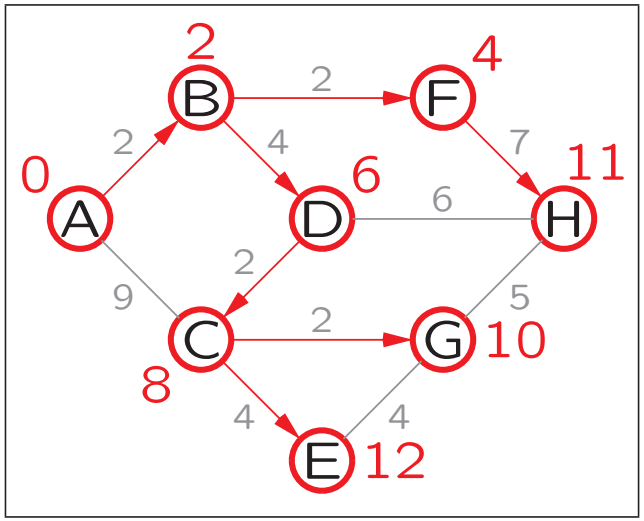

A
B
C
D
E
F
G
H
0
2
8
6
4
11
2
2
9
4
6
2
7
2
5
4
4


A
B
C
D
E
F
G
H
0
2
8
6
4
11
10
12
2
2
9
4
6
2
7
2
5
4
4

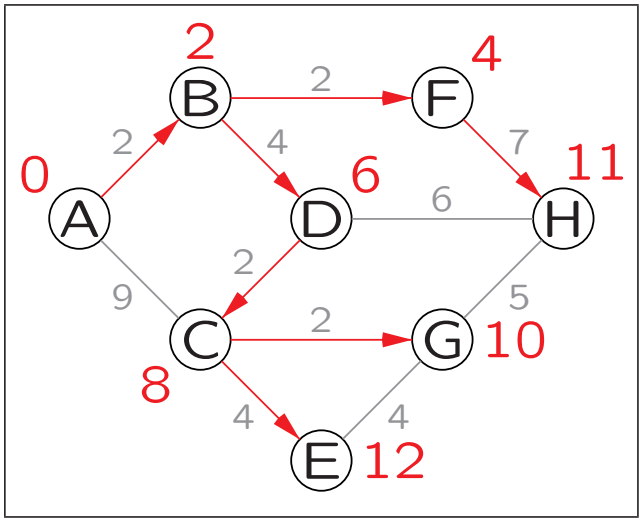

A
B
C
D
E
F
G
H
0
2
8
6
4
11
10
12
2
2
9
4
6
2
7
2
5
4
4


A
B
C
D
E
F
G
H
0
2
8
6
4
11
10
12
2
2
9
4
6
2
7
2
5
4
4

Voorbeeld -3-



Het algoritme is klaar:
alle knopen gehad



Alle kortste paden vanuit
A met hun lengtes

Na elke ronde (dus ook na de laatste, wanneer $U = V$) van het algoritme van Dijkstra geldt:

1. U bevat alle knopen waarvan de definitieve kortste afstand vanaf s reeds bepaald is. Voor elke $v \in U$ geeft het label pad[v] die kortste afstand aan.
2. Voor de andere knopen w ($w \notin U$) is pad[w] de kortste afstand vanuit s naar w via uitsluitend knopen uit U.

Om 1. en 2. te bewijzen moet je laten zien dat:

a. wanneer v^* wordt toegevoegd aan U, pad[v^*] inderdaad de lengte van het kortste pad van s naar v^* bevat
b. na elke update van de labels na toevoeging van v^* de pad[w] nog steeds de kortste afstand via (de nieuwe) U aangeven, voor alle $w \notin U$

- **Lezen bij dit college:**

  Inleiding hoofdstuk 9; paragraaf 9.3; sheets

- **Werkcollege:**

  donderdag 14 april 2011, 13:45–15:30: vervallen!
  donderdag 21 april 2011, alsnog gewoon werkcollege
  en niet werken aan derde programmeeropdracht

- **Opgaven:**

  zie http://www.liacs.nl/home/rvvliet/algoritmiek

- **Volgend college:** donderdag 21 april 2011