Elfde college algoritmiek

2 mei 2014

Dijkstra en Branch & Bound

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.

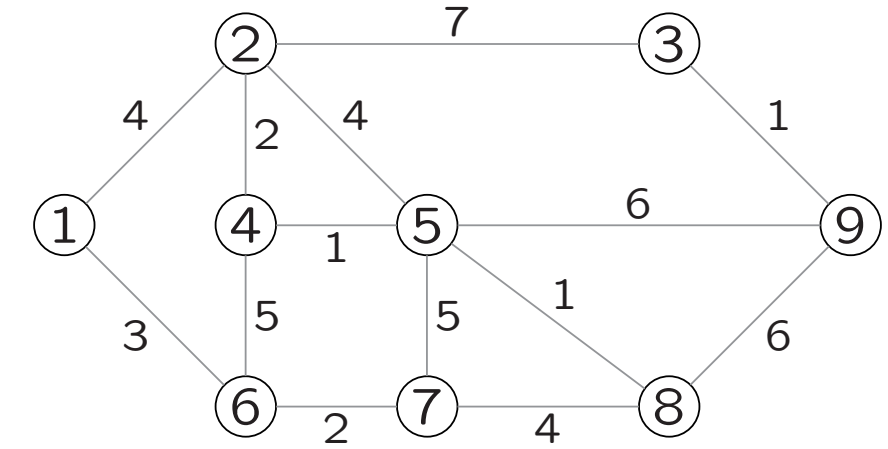
Merk op dat al deze kortste paden vanuit s samen een boomstructuur vormen.

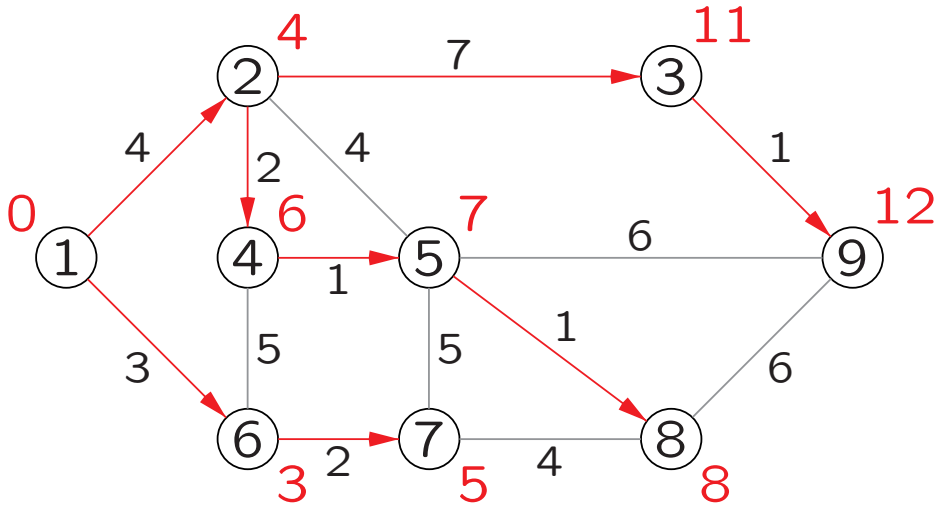
Oplossing: het algoritme van Dijkstra is een *gretig* algoritme, dat de kortste paden van s naar elk van de andere knopen vindt in volgorde van hun lengte. In elke stap wordt een knoop (en daarmee het pad van s naar die knoop) gekozen waarvoor het tot nu toe bekende pad vanaf s zo kort mogelijk is.

Dijkstra: Edsger W. Dijkstra (1930-2002)

// 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

Voorbeeld


1
2
3
4
5
6
7
8
9
7
4
4
1
2
6
1
3
5
5
1
6
2
4


0
4
11
6
7
12
3
5
8
1
2
3
4
5
6
7
8
9
7
4
4
1
2
6
1
3
5
5
1
6
2
4

De achtereenvolgende stappen van het algoritme van Dijkstra voor de voorbeeldgraaf:

1	2	3	4	5	6	7	8	9	Actie
0	∞	∞	∞	∞	∞	∞	∞	∞	Begin met 1
–	4	∞	∞	∞	3	∞	∞	∞	Kies 6, vanaf 1
–	4	∞	8	∞	–	5	∞	∞	Kies 2, vanaf 1
–	–	11	6	8	–	5	∞	∞	Kies 7, via 6
–	–	11	6	8	–	–	9	∞	Kies 4, via 2
–	–	11	–	7	–	–	9	∞	Kies 5, via 4
–	–	11	–	–	–	–	8	13	Kies 8, via 5
–	–	11	–	–	–	–	–	13	Kies 3, via 2
–	–	–	–	–	–	–	–	12	Kies 9, via 3

Een rij in de tabel komt overeen met het array `pad` in de pseudo-code op de volgende slide.

// 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 kortste afstand aan. Moet bewezen worden.

- Voor de andere knopen w geldt na elke ronde (= doorgang door de while):

  (#) $\text{pad}[w] = \min_{u \in U}\{\text{pad}[u] + \text{gewicht}(u, w)\}^*$

  Dit volgt direct uit het algoritme.

*Dit betekent dat pad[w] voor deze knopen $w \notin U$ de lengte van een kortste pad van s naar w aangeeft via uitsluitend knopen van U.

- De volgende *dichtstbijzijnde* knoop v^* wordt gekozen uit de knopen uit $V \setminus U$ die direct grenzen aan U. Nadat deze gekozen is worden de labels aangepast, zodat (#) ook geldt voor de nieuwe U.
- Het is niet zo moeilijk dit algoritme aan te passen zodat ook de kortste paden zelf worden berekend. Sla direct na het aanpassen van het label van knoop v de nieuwe kandidaattak (v^*, v) op:

if pad$[v^*]$ + gewicht(v^*, v) < pad$[v]$ **then**
    pad$[v]$:= pad$[v^*]$ + gewicht(v^*, v);
    nieuwe kandidaattak voor v: (v^*, v);
fi

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

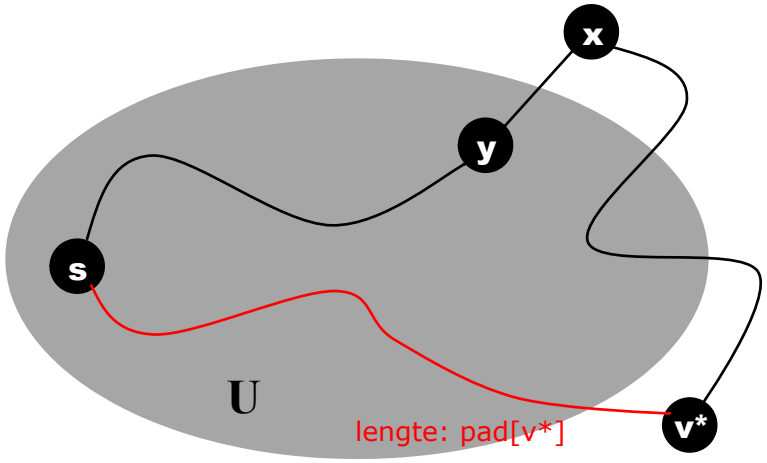
- U bevat alle knopen waarvan de definitieve kortste afstand vanaf s reeds bepaald is. Voor elke $v \in U$ geeft het label pad[v] die kortste afstand aan.

Om dit te bewijzen moet je laten zien dat:

- wanneer v^* wordt toegevoegd aan U, pad[v^*] inderdaad de lengte van het kortste pad van s naar v^* bevat

Essentie van het bewijs

Bekijk, behalve het pad ter lengte pad[v^*] van s naar v^* via U, een willekeurig ander pad van s naar v^*. Stel dat x de eerste knoop op dat pad is buiten U, en y de laatste knoop daarvóór. (Deze x kán gelijk zijn aan v^*.) Zij d(s, y, x, v^*) de lengte van dat pad.



Dan geldt: d$(s, y, x, v^*) \geq$ d$(s, y, x) \geq$ pad$[x] \geq$ pad$[v^*]$ omdat respectievelijk alle gewichten van de takken ≥ 0 zijn, (#) geldt voor x en v^* gekozen was in deze ronde als "minimale" knoop. QED

Backtracking

- bouwt een oplossing component voor component op
- kijkt tijdens de stap-voor-stap constructie of de deeloplossing die bekeken wordt nog aan de gestelde restricties/eisen voldoet (kan voldoen)
- zo niet, breidt dan de deeloplossing niet verder uit en herziet de laatste keuze (backtrack!)
- houdt (alleen) het pad corresponderend met de (deel)oplossing die 'nu' wordt uitgebreid bij
- toepasbaar op allerlei problemen waarbij oplossingen stap voor stap gegenereerd kunnen worden
- werking te beschrijven m.b.v. een state space tree

Optimalisatieprobleem: er wordt een oplossing gezocht met minimale of maximale .....

Terminologie:

- De functie/waarde die ge-optimaliseerd moet worden heet wel de objectfunctie (bijvoorbeeld de lengte van een Hamiltonkring)

- Een oplossing die voldoet aan de restricties van het probleem heet feasible (toelaatbaar)

- Een optimale oplossing is een/de toelaatbare oplossing met de beste waarde van de objectfunctie

Backtracking en optimalisatieproblemen

- Bij een minimalisatieprobleem kun je bijv. ook stoppen met uitbreiden wanneer je deeloplossing al een grotere waarde van de te optimaliseren functie heeft dan de huidige beste oplossing (die wordt dus bijgehouden/opgeslagen) en alleen maar nog groter kan worden. (Iets dergelijks bij een maximalisatieprobleem.)
- Backtracking is het efficiëntst wanneer *een* oplossing gezocht wordt, dus voor optimalisatieproblemen is backtracking niet bij voorbaat de meest geschikte oplossingsmethode.
- Als je snel een (bijna) optimale oplossing vindt, kunnen verdere deeloplossingen eerder worden afgekapt en ben je dus sneller klaar. Branch & bound doet dat beter dan backtracking.

Branch & bound

- is alleen toepasbaar op optimalisatieproblemen
- genereert oplossingen stap voor stap en houdt de tot dusver gevonden beste oplossing bij
- gebruikt voor elke deeloplossing (= knoop in de state space tree) een of andere ondergrens (minimalisatie-probleem) resp. bovengrens (maximalisatieprobleem) op de waarde van de objectfunctie die je zou krijgen bij verdere uitbreiding van die deeloplossing*

*Dit is dus een afschatting voor de waarde die je krijgt als je de deeloplossing verder zou uitbreiden. Hierna noemen we dit wel de te verwachten waarde van de objectfunctie.

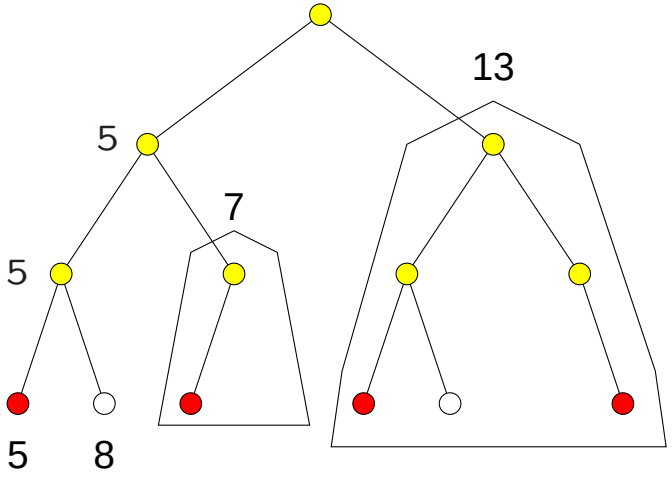
Het doel van zo'n ondergrens/bovengrens is:

- van deeloplossingen te kunnen bepalen dat ze niet verder bekeken hoeven te worden: snoeien*

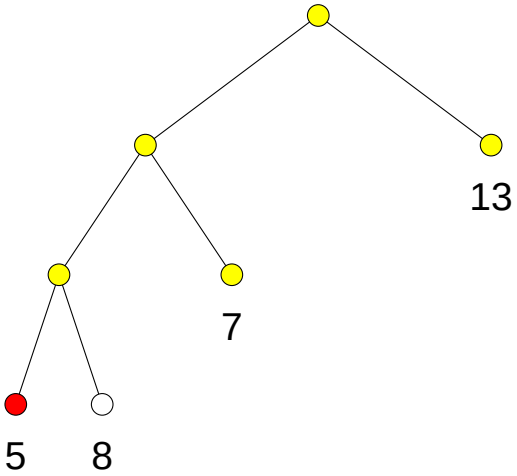
- de zoekvolgorde in de zoekruimte (state space tree), dus de volgorde waarin knopen worden gegenereerd en verder bekeken, te leiden†

*dit zou bij backtracking ook kunnen

†maar dit niet of nauwelijks



state space tree (voorbeeld)



gesnoeide boom

Witte knopen corresponderen met toelaatbare oplossingen; rode knopen met niet-toelaatbare oplossingen; de waarden bij de bladeren geven de waarde van de objectfunctie van de bijbehorende oplossing; de waarden bij de andere knopen geven een ondergrens op de te verwachten waarde van de objectfunctie; bij de knoop met grens 13 kan meteen gesnoeid worden, die met 7 wordt nog bekeken, maar heeft geen toelaatbare uitbreiding

Assignmentprobleem (toewijzingsprobleem)

Gegeven n personen en n taken (jobs). Persoon i kan taak j doen voor `kosten`$[i][j]$ euro.
Gevraagd: de/een toewijzing van de personen aan de jobs (één persoon per job en één job per persoon) met minimale kosten.

Voorbeeld:

	W	X	Y	Z
Alice	9	2	7	8
Bob	6	4	3	7
Carol	5	8	1	8
David	7	6	9	4

	W	X	Y	Z
Alice	9	2	7	8
Bob	6	4	3	7
Carol	5	8	1	8
David	7	6	9	4

Een ondergrens voor de kosten van een (optimale) oplossing:

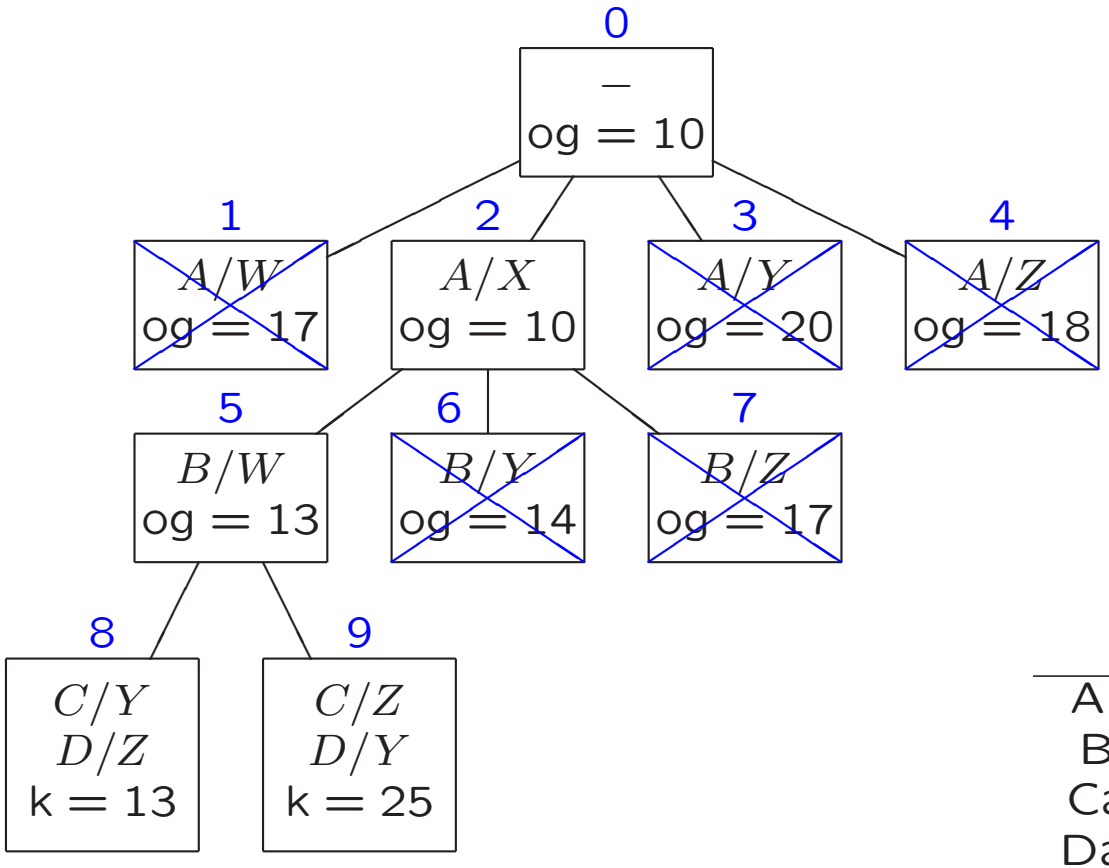
(a) neem uit elke rij de kleinste waarde en tel die bij elkaar op: $2 + 3 + 1 + 4 = 10$

(b) neem uit elke kolom de kleinste waarde en tel die bij elkaar op: $5 + 2 + 1 + 4 = 12$

We genereren oplossingen door de personen een voor een aan een job te koppelen, en gebruiken daarbij de ondergrens volgens suggestie (a).
De optimale oplossing heeft totale kosten 13:
Alice job X; Bob job W; Carol job Y; David job Z

Voor een willekeurige knoop/deeloplossing berekenen we de ondergrens op de te verwachten waarde van de object-functie door uit elke verdere rij de kleinste waarde van de nog beschikbare jobs te nemen en deze op te tellen bij de waarde van de deeloplossing. Voor de deeloplossing(en) die Alice aan job W koppelt zal die ondergrens bijvoorbeeld $9+3+1+4 = 17$ zijn. (Analoog voor kolommen: kies uit elke verdere kolom de kleinste waarde van de nog beschikbare personen(*).)

De volgorde waarin de knopen van de state space tree worden uitgebreid laten we afhangen van de berekende ondergrens. We kiezen de knoop met de beste (=laagste) ondergrens als eerste: dit lijkt de meest veelbelovende knoop. Deze strategie heet wel de best-fit-first branch-and-bound.



	W	X	Y	Z
Alice	9	2	7	8
Bob	6	4	3	7
Carol	5	8	1	8
David	7	6	9	4

Opgave: los het probleem op met de ondergrenzen berekend volgens (*).

Branch & bound

- is alleen toepasbaar op optimalisatieproblemen
- genereert oplossingen stap voor stap en houdt de tot dusver gevonden beste oplossing bij
- gebruikt voor elke deeloplossing (= knoop in de state space tree) een of andere ondergrens (minimalisatie-probleem) resp. bovengrens (maximalisatieprobleem) op de waarde van de objectfunctie die je zou krijgen bij verdere uitbreiding van die deeloplossing*

*te verwachten waarde van de objectfunctie.

Het doel van zo'n ondergrens/bovengrens is:

- van deeloplossingen te kunnen bepalen dat ze niet verder bekeken hoeven te worden: snoeien*

- de zoekvolgorde in de zoekruimte (state space tree), dus de volgorde waarin knopen worden gegenereerd en verder bekeken, te leiden†

*dit zou bij backtracking ook kunnen

†maar dit niet of nauwelijks

Een branch and bound algoritme breidt een knoop (deeloplossing) niet verder uit als

- de waarde van de ondergrens (bovengrens) bij die knoop niet beter is dan de waarde van de tot dusver gevonden beste oplossing: als ondergrens $\geq$ tot dusver gevonden minimale waarde, dan snoeien
- de deeloplossing niet meer voldoet aan de restricties (of niet meer uit te breiden is tot een toelaatbare oplossing)
- er nog maar één volledige, toelaatbare oplossing mogelijk is bij de deeloplossing (i.h.b. als de deeloplossing zo'n volledige oplossing *is*); de waarde van deze ene oplossing wordt met de beste oplossing tot nu toe vergeleken; update zonodig de beste oplossing

De volgorde waarin de knopen (deeloplossingen) worden uitgebreid hangt direct af van de berekende grenzen:

- er worden meerdere deeloplossingen tegelijk bijgehouden, dit in tegenstelling tot backtracking
- in elke stap wordt een van al deze deeloplossingen gekozen, en daarvan worden alle 1-staps-uitbreidingen (kinderen in de state space tree) bekeken en geëvalueerd (d.w.z. ondergrens/bovengrens bepaald)
- zinloze uitbreidingen worden meteen verworpen
- meestal wordt de deeloplossing gekozen die het meest veelbelovend lijkt: best-fit-first branch-and-bound
- bij minimalisatieproblemen (resp. maximalisatieproblemen) kiezen we de knoop met de laagste ondergrens (resp. hoogste bovengrens) als eerste

Los het toewijzingsprobleem op voor onderstaand voorbeeld met behulp van

1. backtracking (snoeien op de kosten van deeloplossingen (*))
2. branch and bound

en vergelijk de hoeveelheid snoeiwerk bij beide methoden, alsmede de volgorde waarin de knopen van de state space tree worden bekeken.

	W	X	Y	Z
Alice	4	7	3	5
Bob	6	2	9	1
Carol	3	9	5	3
David	1	1	1	8

(*) overigens kun je bij backtracking ook ondergrenzen berekenen zoals bij B&B, en die gebruiken om te snoeien; B&B vindt een optimale oplossing echter i.h.a. eerder

Gegeven n objecten, met gewicht $w_1, \ldots, w_n$ en waarde $v_1, \ldots, v_n$, en een knapzak met capaciteit W.
Gevraagd: de meest waardevolle deelverzameling der objecten die in de knapzak past (dus met totaalgewicht $\leq W$).

Voorbeeld:

item	w	v	v/w
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

Maximaal gewicht $W = 10$

item	w	v	v/w
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

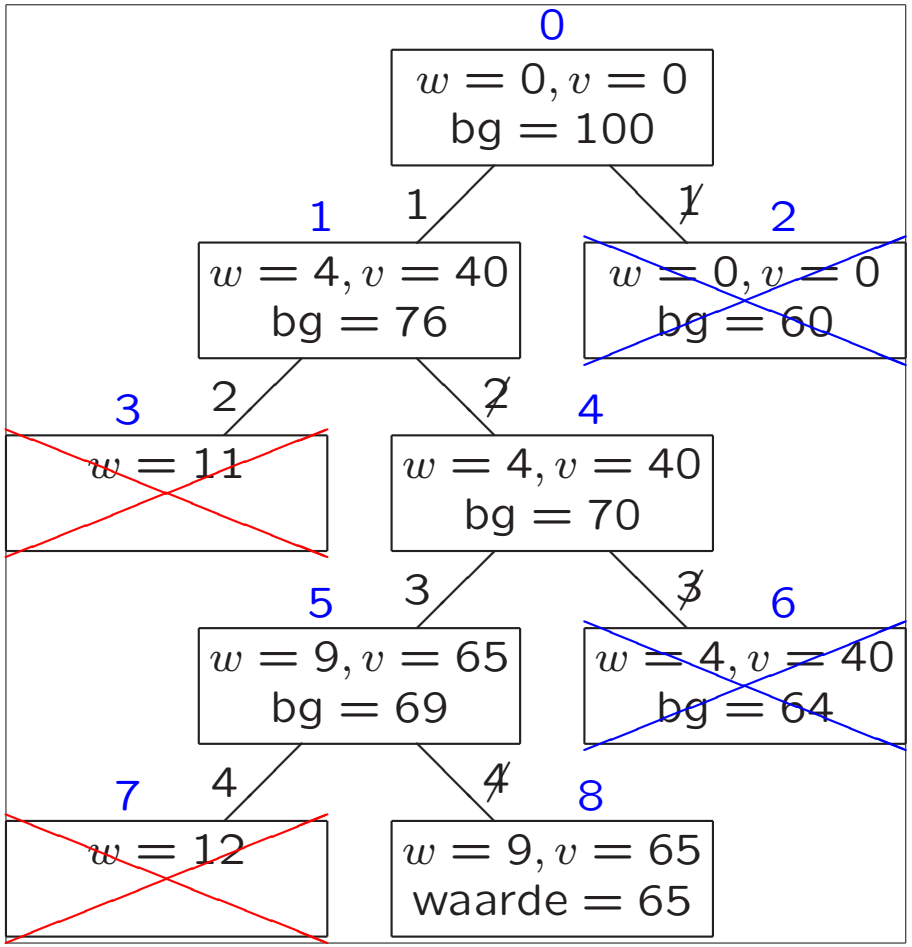
Opbouwen van de oplossing: in de i-de stap wordt object i wel of niet gekozen

Een bovengrens voor de kosten van een optimale oplossing:

- Bij aanvang: $W * (v_1/w_1) = 100$;
- Na de i-de stap: $v + (W - w) * (v_{i+1}/w_{i+1})$, met v de totaalwaarde van de reeds gekozen objecten en w het totaalgewicht daarvan.

De optimale oplossing heeft gewicht 9 en waarde 65: $\{1, 3\}$.

B&B state space tree



item	w	v	v/w
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

Zie ook exercise 12.2.5, Levitin.

- **Deadline programmeeropdracht 3:** 12 mei 2014
- **Lezen/leren bij dit college:** Paragrafen 9.3 en 12.2, sheets
- **Volgende colleges:**
  vrijdag 9 mei 2014 en vrijdag 16 mei 2014
- **Volgende werkcolleges:**
  donderdag 8 mei 2014 en
  donderdag 15 mei 2014 in zaal B2
- **Opgaven:**
  http://www.liacs.nl/home/graaf/ALGO/