Dynamic Programming

Ferd van Odenhoven

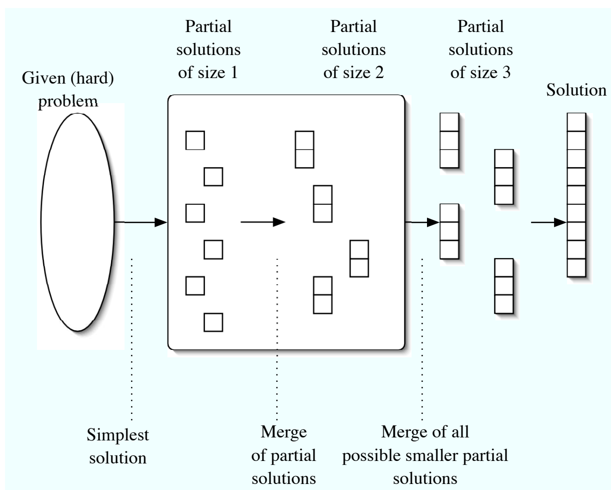
Fontys Hogeschool voor Techniek en Logistiek Venlo
Software Engineering

18 december 2013

Dynamisch Programmeren

- Optimaliseringsmethode voor het oplossen van een (moeilijke) problemen
- Moeilijk betekent hier NP, geen oplossing met complexiteit $O(N^k)$.
- Oplossingsstrategie: Combinatie van gelijksoortige (opgeloste) problemen
- Voorbeelden:
  - Traveling Salesman Problem
  - Bin-Packing Problem
- Voorgestelde algoritmen
  - Hill-Climbing
  - Heuristisch Zoeken
  - Dijkstra's Shortest Path Algorithm
  - A* Algorithm

Schema van dynamisch programmeren



Progammaschema van dynamisch programmeren

```
/**
 *  pseudo code!
 */

void DynamicProgramming (problem, solution) {
    t = 1;
    decompose into trivial problems;
    solve trivial problems;
    t++;
    while (t <= size of solution) {
        while (combination to size t possible) {
            combine partial solutions;
            find best practical solution;
            store combined (best) partial solution;
        }
        t++;
    }
}
```


Problemen bij dynamisch programmeren

- Welke problemen zijn op praktische wijze te ontleden?
- Hoe vinden we de kleinste eenvoudige (deel-) oplossing?
- Hoe bepalen we de complexiteit van deze algoritmen?
  1. De complexiteit is moeilijk te bepalen.
  2. De complexiteit hangt af van het corresponderende probleem.

Voorbeeld: Associativiteit bij matrixvermenigvuldiging

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \begin{pmatrix} c_{11} \\ c_{12} \\ c_{13} \end{pmatrix} \begin{pmatrix} d_{11} & d_{12} \end{pmatrix} \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix} \begin{pmatrix} f_{12} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix}$$

- De associatieve wet voor de matrixvermenigvuldiging bepaalt het aantal benodigde vermenigvuldigingen
- Links-associativiteit `(((((AB)C) D) E) F)` resulteert in 84 vermenigvuldigingen.
- Rechts-associativiteit `(A (B (C (D (EF)))))` resulteert in 69 vermenigvuldigingen.

Voorbeeld: associativiteit bij matrixvermenigvuldiging

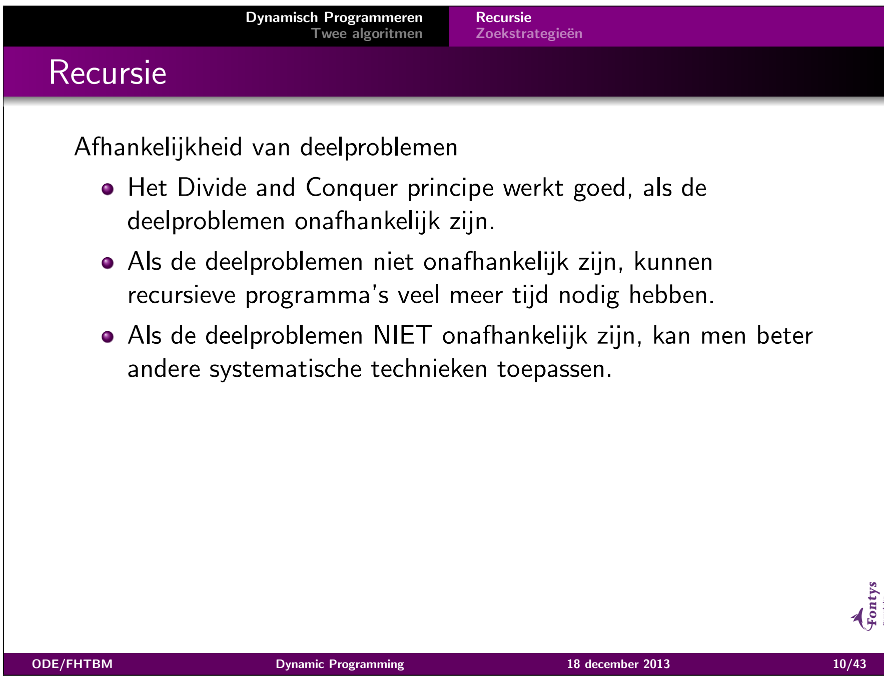
- Vraag: welke haakjes volgorde is optimaal, d.w.z. heeft het minimale aantal vermenigvuldigingen?
- Om all mogelijkheden (“uitputtend zoeken“) is niet praktisch voor een groot aantal matrices n omdat $\frac{1}{n+1}\binom{2n}{n} \sim O(4^n)$ verschillende haakjesparen mogelijk zijn.

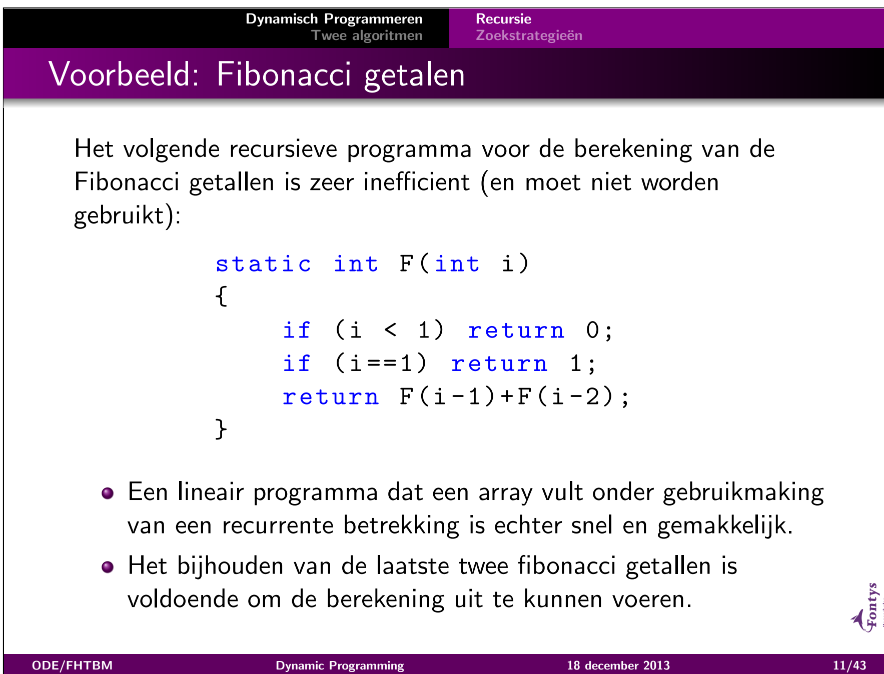
Optimale haakjesplaatsing

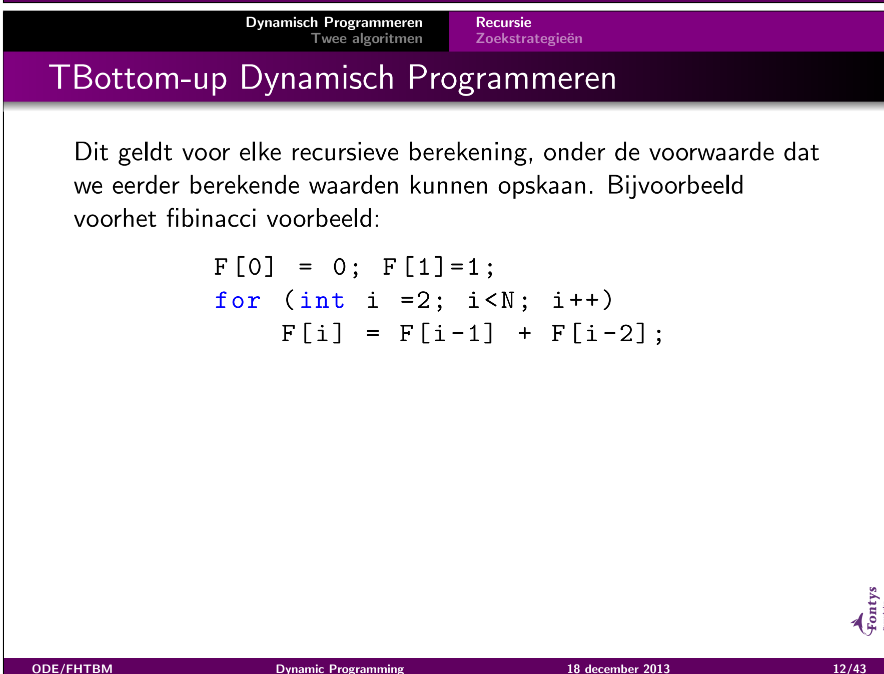
- Laat: $A_1, A_2, \ldots, A_n$ een eindige reeks matrices zijn en $r_0, r_1, r_2 \ldots, r_n$ de zogenaamde aansluitgetallen, die de dimensie van de matrices beschrijven:
- A_i is een $\left(r_{i-1} \times r_i\right)$-matrix. Een optimale haakjesvolgorde voor het produkt $A_1 A_2 \ldots A_n$ wordt gegeven door paarsgewijze tweedeling van de vorm $A_1 A_2 \ldots A_{k^*}$ en $A_{k^*+1} \ldots A_n$ $(1 \leq k^* < n)$.
- Opdat het gehele produkt optimaal 'omhaakt' is, moeten beide delen ook optimaal 'omhaakt' zijn.
- Zodoende bestaat de optimale oplossing van het totale probleem uit de optimale oplossing van de deelproblemen.
- Elke optimale oplossing van een deelprobleem is weer deel van de optimale oplossing van het totale probleem (optimale onderstruktuur).

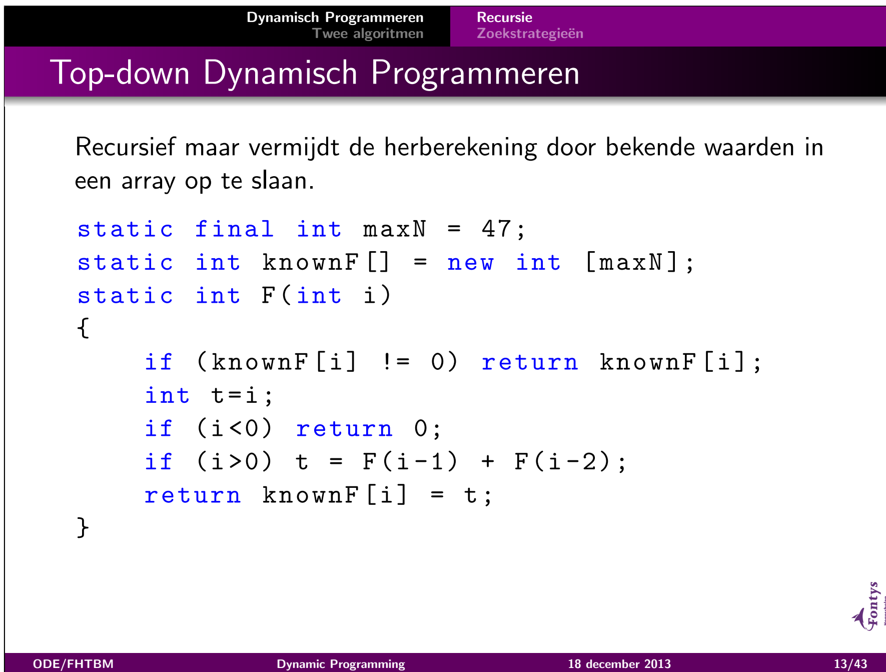
Voorbeelden van bekende en toepasbare algoritmen

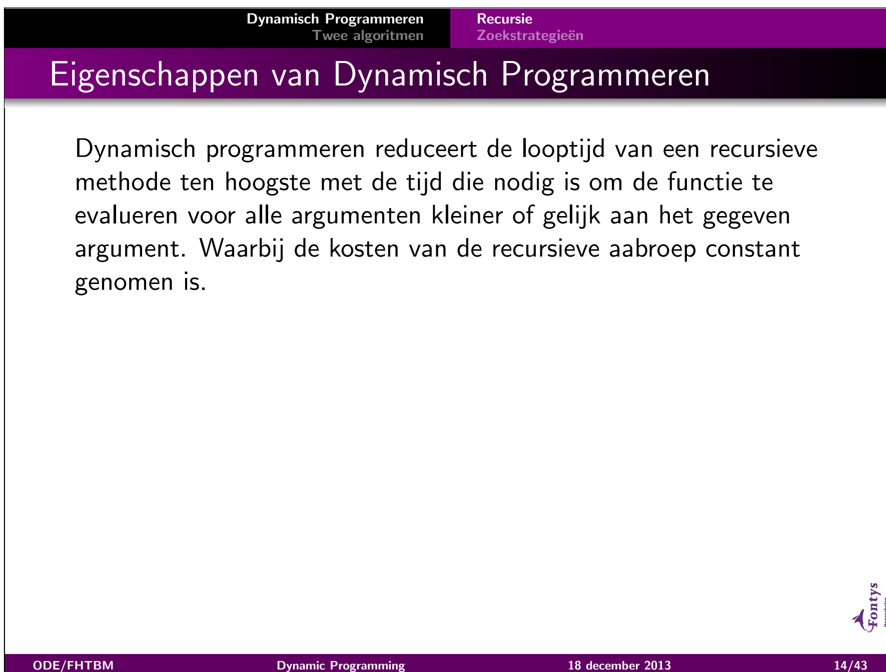
- Greedy algoritme
  - Diepte eerst zoeken: DFS
  - Breedte eerst zoeken: BFS
  - Padalgoritmen
- Recursie
  - Berekening van Fibonaccigetallen

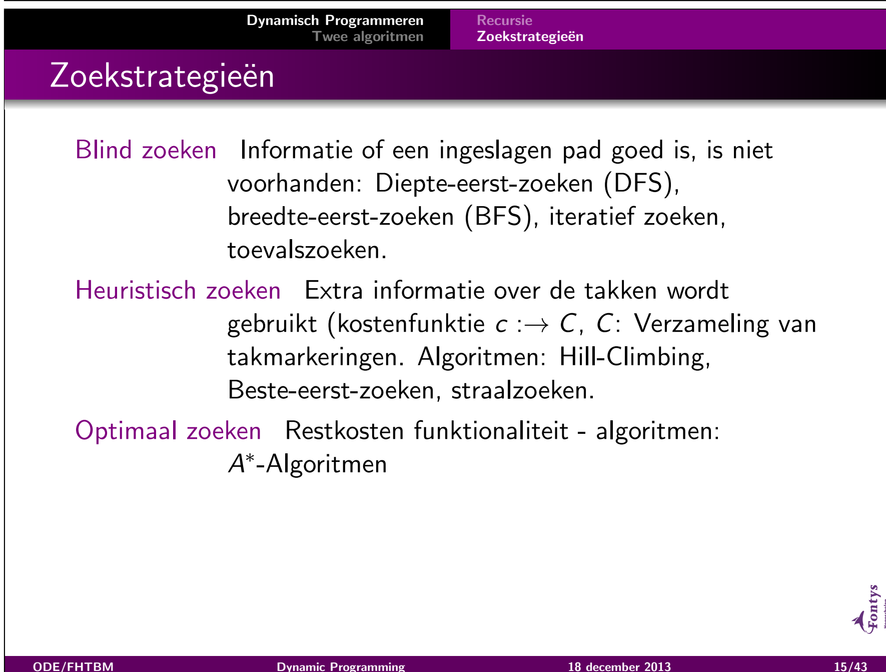

Dynamisch Programmeren
Twee algoritmen
Recursie
Zoekstrategieën
Recursie
Afhankelijkheid van deelproblemen
Het Divide and Conquer principe werkt goed, als de deelproblemen onafhankelijk zijn.
Als de deelproblemen niet onafhankelijk zijn, kunnen recursieve programma's veel meer tijd nodig hebben.
Als de deelproblemen NIET onafhankelijk zijn, kan men beter andere systematische technieken toepassen.
ODE/FHTBM
Dynamic Programming
18 december 2013
10/43


Dynamisch Programmeren
Twee algoritmen
Recursie
Zoekstrategieën
Voorbeeld: Fibonacci getalen
Het volgende recursieve programma voor de berekening van de Fibonacci getallen is zeer inefficient (en moet niet worden gebruikt):
static int F(int i)
{
if (i < 1) return 0;
if (i==1) return 1;
return F(i-1)+F(i-2);
}
Een lineair programma dat een array vult onder gebruikmaking van een recurrente betrekking is echter snel en gemakkelijk.
Het bijhouden van de laatste twee fibonacci getallen is voldoende om de berekening uit te kunnen voeren.
ODE/FHTBM
Dynamic Programming
18 december 2013
11/43


Dynamisch Programmeren
Twee algoritmen
Recursie
Zoekstrategieën
TBottom-up Dynamisch Programmeren
Dit geldt voor elke recursieve berekening, onder de voorwaarde dat we eerder berekende waarden kunnen opskaan. Bijvoorbeeld voorhet fibinacci voorbeeld:
F[0] = 0; F[1]=1;
for (int i =2; i<N; i++)
F[i] = F[i-1] + F[i-2];
ODE/FHTBM
Dynamic Programming
18 december 2013
12/43


Dynamisch Programmeren
Twee algoritmen
Recursie
Zoekstrategieën
Top-down Dynamisch Programmeren
Recursief maar vermijdt de herberekening door bekende waarden in een array op te slaan.
static final int maxN = 47;
static int knownF[] = new int [maxN];
static int F(int i)
{
if (knownF[i] != 0) return knownF[i];
int t=i;
if (i<0) return 0;
if (i>0) t = F(i-1) + F(i-2);
return knownF[i] = t;
}
ODE/FHTBM
Dynamic Programming
18 december 2013
13/43


Dynamisch Programmeren
Twee algoritmen
Recursie
Zoekstrategieën
Eigenschappen van Dynamisch Programmeren
Dynamisch programmeren reduceert de looptijd van een recursieve methode ten hoogste met de tijd die nodig is om de functie te evalueren voor alle argumenten kleiner of gelijk aan het gegeven argument. Waarbij de kosten van de recursieve aabroep constant genomen is.
ODE/FHTBM
Dynamic Programming
18 december 2013
14/43


Dynamisch Programmeren
Twee algoritmen
Recursie
Zoekstrategieën
Zoekstrategieën
Blind zoeken Informatie of een ingeslagen pad goed is, is niet voorhanden: Diepte-eerst-zoeken (DFS), breedte-eerst-zoeken (BFS), iteratief zoeken, toevalszoeken.
Heuristisch zoeken Extra informatie over de takken wordt gebruikt (kostenfunktie c :→ C, C: Verzameling van takmarkeringen. Algoritmen: Hill-Climbing, Beste-eerst-zoeken, straalzoeken.
Optimaal zoeken Restkosten funktionaliteit - algoritmen: A*-Algoritmen
ODE/FHTBM
Dynamic Programming
18 december 2013
15/43

Methode van het Brits museum

Methode van het Brits museum Constructie van alle paden en vergelijken van de kosten bij succesvolle paden, te beginnen met de kleinste. Succesvolle paden worden in een oplossingsverzameling geplaatst.

Complexiteit Als elke knoop maximaal b opvolgers heeft en de boom een diepte d heeft, dan worden er b^d verschillende paden gevonden, waarbij niet alle paden naar het doel leiden.

Hill Climbing Algoritme: Eigenschappen

- Het algoritme begint met een willekeurige (mogelijk slechte) oplossing, en vervolgens worden iteratief kleine verbeteringen in de oplossing aangebracht. Als het algoritme geen verbeteringen meer kan vinden stopt het.
- In het ideale geval is op dat punt de oplossing dichtbij de optimale, maar en zijn daarvoor geen garanties.
- Ofschoon geavanceerdere algoritmen een beter resultaat kunnen geven, werkt hill-climbing in sommige gevallen net zo goed.

(http://en.wikipedia.org/wiki/Hill_climbing)

Hill Climbing Algoritme: Voorbeelden

- Hill climbing kan gebruikt worden voor de problemen met veel oplossingen, waarvan sommige beter zijn dan andere.
- Als voorbeeld kan hill climbing toegepast worden op het zogenaamde handelsreizigerprobleem. Het is niet moeilijk een oplossing te vinden die alle steden aandoet, maar die zal lang niet optimaal zijn. Het algoritme start met zo'n oplossing en maakt dan kleine aanpassingem, zoals het verandering van de volgorde waarin de steden bezocht worden. Uiteindelijk resulteert een veel betere route.

Hill Climbing Algoritme: Wikundig

- Hill climbing probeert een functie $f(x)$ te maximaliseren (of the minimaliseren), waarin x discrete toestanden zijn. Deze toestanden zijn typisch de knopen in een graaf, waarin de takken in de graaf de nabijheid of evereenkomst van de knopen bevat. Hill climbing volgt de graaf van knoop naar knoop, steeds lokaal zoekend naar een toenemende (afnemende) waarde van f. totdat een lokaal maximum (of minimum) x_m is bereikt.

Hill Climbing Pseudo Code

```
Hill Climbing Algorithm
   currentNode = startNode;
   loop do
      L = NEIGHBORS(currentNode);
      nextEval = -INF;
      nextNode = NULL;
      for all x in L
         if (EVAL(x) > nextEval)
              nextNode = x;
              nextEval = EVAL(x);
      if nextEval <= EVAL(currentNode)
         //Return current node
         //since no better neighbours exist
         return currentNode;
      currentNode = nextNode;
```

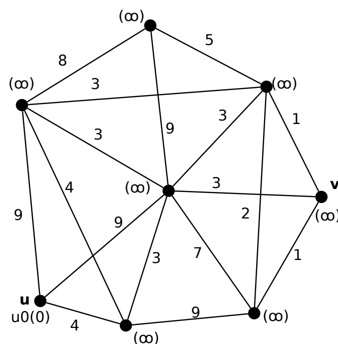
Conclusie

- Pas Hill-Climbing toe als de hoeveelheid data groot is en optimale methoden teveel tijd kosten.
- Je treft op het internet veel voorbeelden aan die niet nuttig zijn. Maak je eigen oplossing!
- Voor de enthousiastelingen: `http://en.wikipedia.org/wiki/Dynamic_programming`

Kortste pad algoritme van Dijkstra: idee

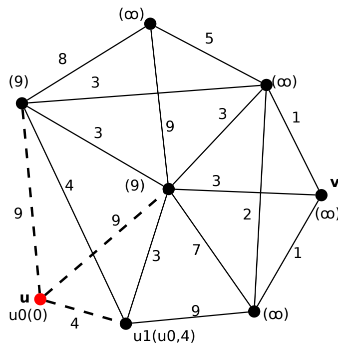
- In een gewogen graaf zoeken we het kortste pad van knoop **u** naar knoop **v**.
- Beginnend bij **u** wordt een boom opgebouwd.
- Alle knopen die over een tak vanuit de boom bereikt kunnen worden, worden onderzocht.
- Bij elke knoop: bereken de kortste afstand tot **u** en bewaar die afstand in een label.
- De knoop en de corresponderende tak met de kleinste afstand wordt aan de boom toegevoegd.
- *Er kan geen ander pad van* **u** *naar deze nieuwe knoop zijn dat korter is dat het pad via de boom!*
- Stopcriteria zijn: **v** is bereikt, of alle knopen zijn aan de boom toegevoegd.

Kortste pad algoritme van Dijkstra: voorbeeld



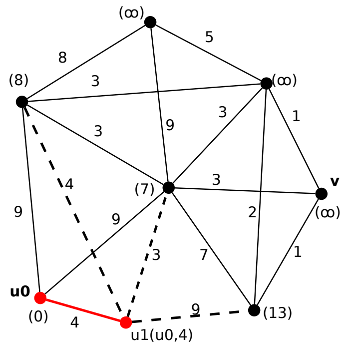
Figuur: We zoeken het kortste pad van **u** naar **v**. Bij elke knoop plaatsen we een label met de kortste afstand tot **u**.

Kortste pad algoritme van Dijkstra 1



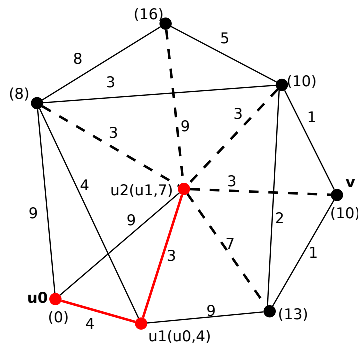
Figuur: Het resultaat is een boom waarin elk pad tussen twee knopen ook het kortste pad is. Hier is alleen **u** al in die boom, die we rood maken. De labels moeten ook de voorganger bevatten.

Kortste pad algoritme van Dijkstra 2



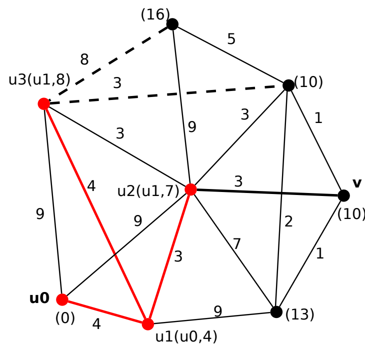
Figuur: Onderzoek alle knopen die over één tak vanuit de boom bereikt kunnen worden: afstanden worden aangepast. De knoop met de kleinste afstand tot **u** wordt aan de boom toegevoegd, met de bijbehorende tak. Een korter pad van **u** tot deze knoop is er niet. Ga na.

Kortste pad algoritme van Dijkstra 3



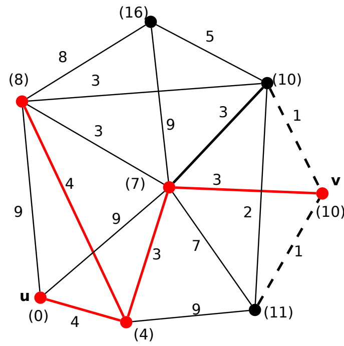
Figuur: Alle andere knopen zijn nu bereikbaar vanuit de boom over een enkele tak. De knoop met afstand 8 wordt als volgende toegevoegd. De voorganger is de tweede knoop in de boom. De voorganger worden niet meer aangegeven, we tonen alleen het orincipe van het algoritme.

Kortste pad algoritme van Dijkstra 4



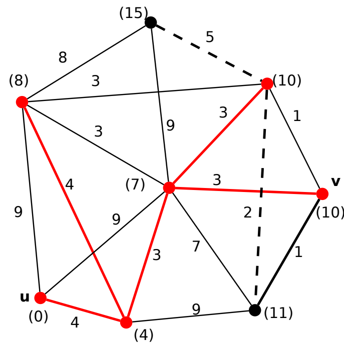
Figuur: Er zijn nu twee knopen met minimum afstand 10. We kiezen natuurlijk voor **v**.

Kortste pad algoritme van Dijkstra 5



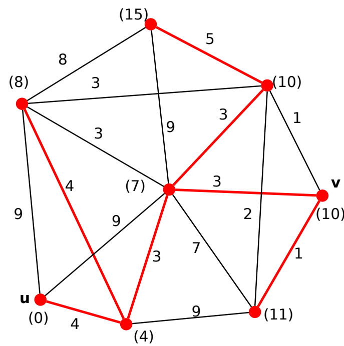
Figuur: In de volgende stap wordt de andere knoop met afstand 10 genomen. We laten de complete knoopinformatie achterwege.

Kortste pad algoritme van Dijkstra 6



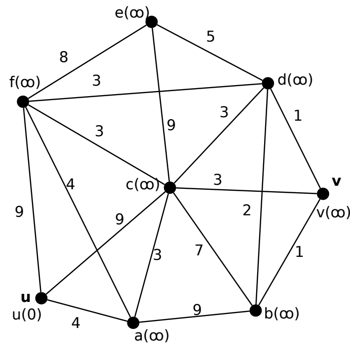
Figuur: Nog twee knopen te gaan. Volg de figuren en controleer de stappen.

Kortste pad algoritme van Dijkstra 7



Figuur: Het eindresultaat: een opspannende boom.

Kortste pad algoritme van Dijkstra



Figuur: maak een tabel.

Dijkstra in table

u	a	b	c	d	e	f	v
0	∞	∞	∞	∞	∞	∞	∞
—	4, u	∞	9, u	∞	∞	9, u	∞
—	—	13, a	7, a	∞	∞	8, a	∞
—	—	13, a	—	10, c	16, c	8, a	10, c
—	—	13, a	—	10, c	16, c	—	10, c
—	—	11, v	—	10, c	16, c	—	—
—	—	11, v	—	—	15, v	—	—
—	—	—	—	—	15, v	—	—
—	—	—	—	—	—	—	—

Tabel: Legenda: (distance,previous node on shortest path), - if finished.

A^*-Algoritme

Concept:

- A^* is een netwerk zoekalgoritme, dat een "afstand-tot-doel + pad-kosten"-score gebruikt.
- Terwijl het netwerk doorlopen wordt en alle buren zoekt, volgt het een laagste-score-pad, dat wordt bijgehouden in een prioriteite-queue met alternatieve paden of padsegmenten.
- Als op een punt in het pad dat gevolgd wordt een hogere score optreedt, dan in andere padsegmenten, wordt het hogere-score-pad verlaten en wordt in plaats daarvan het lagere-score-pad gevolgd.
- Dit gaat door totdat het doel bereikt is.

A^* Algoritme

- Het A^* algoritme verenigt de methoden: Branch and Bound + Restkostenafschatting + Dynamisch Programmeren
- Het pad dat de laagste kosten heeft en de geringste geschatte kosten heeft wordt uitgebreid. Daartoe moeten de totale kosten minimaal zijn.
- Dan worden alle paden verwijderd, waarvan de eindknopen gelijk zijn, uitgezonderd die paden die de kleinste kosten hebben.

(http://en.wikipedia.org/wiki/A*_search_algorithm)

Het algoritme

- $g(x)$: de actuele kortste afstand doorlopen tot de huidige knoop
- $h(x)$: de geschatte (of "heuristische") afstand van de huidige knoop tot het doel
- $f(x)$: the sum of $g(x)$ and $h(x)$.

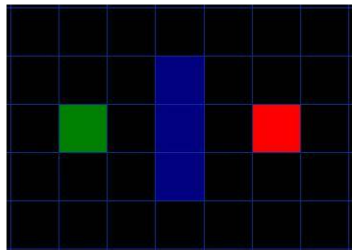
Het algoritme

- Startend met de beginknoop, wordt een prioriteitenqueue met knopen, die doorlopen moeten worden, bijgehouden: die we de open verzameling noemen.
- Hoe lager $f(x)$ is voor een gegeven knoop x, des te hoger zijn prioriteit.
- Bij elke stap van het algoritme, de knoop met de laagste $f(x)$ waarde wordt uit de queue genomen, de f en h waarden van de buren worden geüpdate, en deze buren worden aan de queue toegevoegd.
- Het algoritme gaat door totdat een doelknoop een lagere f waarde heeft dan elke ander knoop in de queue (of totdat de queue leeg is).
- Doelknopen kunnen meermaals gepasseert worden als er knopen overblijven met een lagere f waarde, omdat ze kunnen leiden tot een korter pad naar het doel.

Het algoritme

- De f waarde van het doel is dan de lengte van het kortste pad, aangezien h bij het doel nul is bij een goede heuristiek.
- Als het actuele kortste pad gewenst is, kan het algoritme ook alle buren en hun directe voorgange updaten in het tot zover beste pad.
- Deze informatie kan gebruikt worden om het pad te reconstrueren door terugwaards vanuit de doelknoop te lopen.
- Als verder de heuristiek monotoon is en consistent, kan een verzameling van reeds gevonden knopen gebruikt worden om het zoekproces efficiënter te maken.

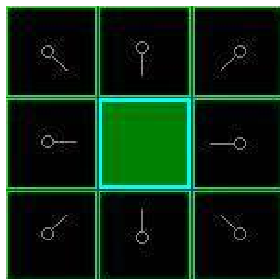
Voorbeeld: Speelveld



Figuur: Cellen waar een object zich kan bevinden en een vertikaal blokkerend object. Een object ter grootte van een cel wil van de groene naar de rode cel (doel) bewegen.

(http://www.policyalmanac.org/games/aStarTutorial.htm)

Basiscel



Figuur: Vanuit een cell zijn er acht richtingen waarin bewogen kan worden.

Eerste stap

Figuur: In elke cel zetten we $f(x) = g(x) + h(x)$ in de linker bovenhoek. We zetten $g(x)$ in de linkeronderhoek en tenslotte zetten we $h(x)$ in de rechterbenedenhoek. Deze $h(x)$ is gelijk aan de kosten: een schatting voor de afstand tot het doel. De Manhattan methode is gebruikt: de som van de horizontale en verticale verschillen op het grid.

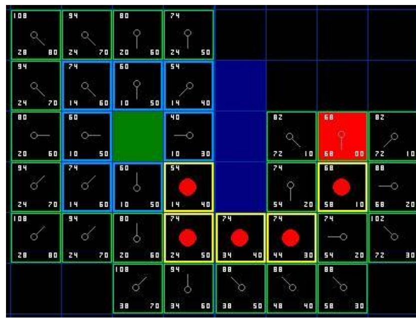
Tweede stap

Figuur: Het algoritme heeft het onderste pad gekozen (om een of andere reden). Netjes de hoek omgaan!

Klaar

Figuur: Het algoritme heeft het doel bereikt. Geen reden om nog verder te gaan zoeken naar een kortere weg. We zouden dat kunnen doen door het hill-climbing algoritme te gebruiken.

Het pad



Figuur: In dit eenvoudig voorbeeld is het resultaat optimaal. Bedenk eens een vorm van het obstructieobject, dat zou leiden tot een gecompliceerdere situatie met een mogelijk minder optimaal pad.