Vijfde college algoritmiek

8 maart 2012

Exhaustive search en Backtracking

Exhaustive search: brute force benadering voor problemen die te maken hebben met het vinden van een element met een speciale eigenschap binnen een verzameling van bijv. permutaties of deelverzamelingen of toestanden of ...

Methode:

. construeer op een systematische manier alle kandidaat-oplossingen, bijvoorbeeld alle permutaties van de getallen 1 t/m n

. evalueer elk van deze mogelijke oplossingen

. retourneer een/de kandidaatoplossing met de gevraagde eigenschap (als die bestaat) (*)

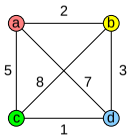
(*) soms, zoals bij optimalisatieproblemen, *moet* je daartoe alle kandidaatoplossingen gezien hebben

Traveling Salesman Problem (handelsreizigersprobleem)

Gegeven n steden waarvan alle onderlinge afstanden bekend zijn. **Gevraagd**: de/een kortste route die elke stad precies één keer aandoet, en weer terugkeert in het vertrekpunt.
Ofwel: vind de/een kortste Hamiltonkring in een samenhangende gewogen (volledige) graaf.

Voorbeeld:

minimale route
a b d c a
(of a c d b a)



Route	Lengte
a → b → c → d → a	$2+8+1+7=18$
a → b → d → c → a	$2+3+1+5=11$
a → c → b → d → a	$5+8+3+7=23$
a → c → d → b → a	$5+1+3+2=11$
a → d → b → c → a	$7+3+8+5=23$
a → d → c → b → a	$7+1+8+2=18$

Complexiteit: minstens $\Theta((n-1)!)$, immers alle $(n-1)!$ mogelijke Hamiltonkringen worden bekeken.

Knapzakprobleem

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:

object	gewicht	waarde
1	8	42
2	3	14
3	4	40
4	5	27

knapzakcapaciteit 12

deelverzameling	gewicht	waarde
$\emptyset$	0	0
$\{1\}$	8	42
$\{2\}$	3	14
$\{3\}$	4	40
$\{4\}$	5	27
$\{1,2\}$	11	56
$\{1,3\}$	12	82
$\{1,4\}$	13	te zwaar
$\{2,3\}$	7	54
$\{2,4\}$	8	41
$\{3,4\}$	9	67
$\{1,2,3\}$	15	te zwaar
$\{1,2,4\}$	16	te zwaar
$\{1,3,4\}$	17	te zwaar
$\{2,3,4\}$	12	81
$\{1,2,3,4\}$	20	te zwaar

Complexiteit: minstens $\Theta(2^n)$, immers alle 2^n deelverzamelingen van n objecten worden bekeken.

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:

	job 1	job 2	job 3	job 4
Anna	9	2	7	8
Bob	6	4	3	7
Carla	5	8	1	8
David	7	6	9	4

$n=4$

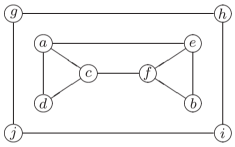
	job 1	job 2	job 3	job 4
Anna	9	2	7	8
Bob	6	4	3	7
Carla	5	8	1	8
David	7	6	9	4

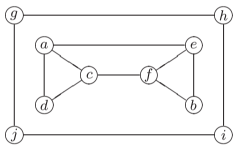
```
1,2,3,4 -> 9+4+1+4 = 18        2,3,1,4 -> ..        3,4,1,2 -> ..
1,2,4,3 -> 9+4+8+9 = 30        2,3,4,1 -> ..        3,4,2,1 -> ..
1,3,2,4 -> 9+3+8+4 = 24        2,4,1,3 -> ..        4,1,2,3 -> ..
1,3,4,2 -> 9+3+8+6 = 26        2,4,3,1 -> ..        4,1,3,2 -> ..
1,4,2,3 -> 9+7+8+9 = 33        3,1,2,4 -> ..        4,2,1,3 -> ..
1,4,3,2 -> 9+7+1+6 = 23        3,1,4,2 -> ..        4,2,3,1 -> ..
2,1,3,4 -> 2+6+1+4 = 13        3,2,1,4 -> ..        4,3,1,2 -> ..
2,1,4,3 -> 2+6+8+9 = 25        3,2,4,1 -> ..        4,3,2,1 -> ..
```

De goedkoopste toewijzing is hier `2,1,3,4`, met kosten 13.

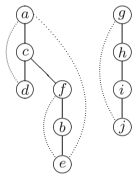
Complexiteit: minstens $\Theta(n!)$, immers alle $n!$ mogelijke toewijzingen worden bekeken.

* Exhaustive search algoritmen werken **alleen voor kleine probleeminstanties** in acceptabele tijd
* Voor veel problemen zijn er veel efficiëntere algoritmen bekend (Eulerkring, kortste paden, toewijzingsprobleem)
* Voor andere problemen is exhaustive search (of varianten daarop) in essentie de enig bekende oplossing (handelsreizigersprobleem, knapzakprobleem)





$e_{6,2}$
$b_{5,3}$ $j_{10,7}$
$d_{3,1}$ $f_{4,4}$ $i_{9,8}$
$c_{2,5}$ $h_{8,9}$
$a_{1,6}$ $g_{7,10}$

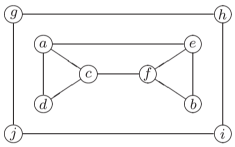


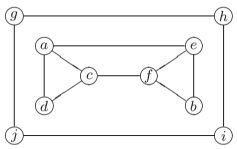

```
ALGORITME DFS (G)
// Implementeert DFS wandeling door gegeven graaf
// Invoer: Graaf G = (V,E)
// Uitvoer: Graaf G met zijn knopen genummerd in de volgorde
//          waarin ze bij DFS wandeling voor het eerst worden ontdekt

{ for  elke knoop v in V  do
    mark[v] = 0;   // nog niet bezocht
  od
  teller = 0;
  for  elke knoop v in V  do
    if  mark[v] == 0  then
      dfs (v);
    fi
  od
}
```



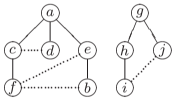
```
dfs (v)
// Bezoekt recursief alle nog onbezochte knopen die via een pad
// met v zijn verbonden, en nummert deze in de volgorde waarin
// ze worden ontdekt, met globale variabele 'teller'
{ teller ++;
  mark[v] = teller;
  for  elke buurknoop w van v  do
    if  mark[w] == 0  then
      dfs (w);
    fi
  od
}
```

Er is ook niet-recursieve implementatie, met expliciete stapel





a_1 c_2 d_3 e_4 f_5 b_6
g_7 h_8 j_9 i_{10}




```
ALGORITME BFS (G)
// Implementeert BFS wandeling door gegeven graaf
// Invoer: Graaf G = (V,E)
// Uitvoer: Graaf G met zijn knopen genummerd in de volgorde
//          waarin ze bij BFS wandeling worden bezocht

{ for  elke knoop v in V  do
    mark[v] = 0;   // nog niet bezocht
  od
  teller = 0;
  for  elke knoop v in V  do
    if  mark[v] == 0  then
      bfs (v);
    fi
  od
}
```

```
bfs (v)
// Bezoekt alle nog onbezochte knopen die via een pad
// met v zijn verbonden, en nummert deze in de volgorde waarin
// ze worden bezocht, met globale variabele 'teller'
{ teller ++;
  mark[v] = teller;
  initialiseer queue met v erin;
  while  queue is niet leeg  do
    for  elke buurknoop w van voorste-knoop-in-queue  do
      if  mark[w] == 0  then
        teller ++;
        mark[w] = teller;
        voeg w toe aan queue;  // achteraan
      fi
    od
    verwijder voorste knoop uit queue;
  od
}
```


	DFS	**BFS**
Data structuur	een stapel	een queue
Aantal volgordes knopen	twee volgordes	één volgorde
Soorten takken (onger. grf)	tree en back edges	tree en cross edges
Toepassingen	samenhang, acyclicliteit, 'articulation points'	samenhang acyclicliteit minimum-tak pad
Efficiëntie voor adj. matrix	$\Theta(\|V\|^2)$	$\Theta(\|V\|^2)$
Efficiëntie voor adj. list	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Bij veel problemen gaat het erom een element met een speciale eigenschap te vinden binnen een ruimte die exponentieel groeit als functie van de invoergrootte. Dan wordt meestal backtracking gebruikt als goed alternatief voor ES.

Exhaustive search genereert alle kandidaatoplossingen en haalt daar het speciale element tussenuit.

Backtracking
- bouwt kandidaatoplossingen component voor component op,
- kijkt al tijdens de constructie of de deeloplossing nog tot een oplossing kan leiden en
- zo niet, breidt dan de deeloplossing niet verder uit

Op deze manier spaar je soms veel werk uit en kun je dus grotere probleeminstanties oplossen.

Backtracking versus exhaustive search

Exhaustive search bekijkt *alle* volledige kandidaatoplossingen.
Backtracking controleert telkens van deeloplossingen of ze nog aan de eisen/restricties voldoen; zo niet, dan weet je zeker dat alle uitbreidingen van deze oplossing ook niet voldoen, dus die hoef je dan niet meer expliciet te bekijken.

Voorbeeld
Gegeven de rij $A = 3, 1, 4, 1, 5, 9, 2, 6, 5, 7$.
Gevraagd: de/een langste *stijgende* deelrij (met volgorde der elementen als in A zelf).

Exhaustive search. Genereer alle 2^{10} deelrijtjes van A en controleer van elk daarvan of hij stijgend is en bepaal wat de langste deelrij is.

Backtracking. Bouw de deelrijtjes stap voor stap op (hoe?) en controleer na elke stap of het deelrijtje nog wel stijgend is. Zo niet, dan hoeft het rijtje niet meer uitgebreid te worden (het kan toch niets worden). Houd de lengte van het deelrijtje bij en vergelijk die met de lengte van de tot dusver gevonden langste deelrij.

Deze methode kan erg veel werk uitsparen. Bijvoorbeeld het deelrijtje $3, 1$ is al niet stijgend, dus alle 2^8 deelrijtjes van A die met $3, 1$ beginnen zeker ook niet. Deze hoeven bij backtracking dus niet allemaal te worden gegenereerd.

Basisidee backtracking

- bouw een oplossing stap voor stap op en controleer steeds of de deeloplossing in conflict komt met de restricties (en nog wel tot een oplossing kan leiden)
- op elk moment kun je kiezen uit een aantal mogelijke vervolgstappen; maak een keuze en ga langs die weg verder met het opbouwen van de oplossing
- als een keuze op niets uitloopt, herzie je deze keuze en probeer je een andere mogelijkheid

Vergelijk

- het vinden van de uitgang in een doolhof: loop steeds verder en als je bij het zoeken vastloopt, ga terug op je pad om het laatste open alternatief te proberen

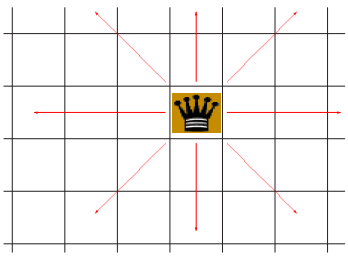
Het acht koninginnenprobleem luidt als volgt:

1. Kun je 8 dames (koninginnen) op een 8 bij 8 schaakbord zetten zonder dat zij elkaar aanvallen (= in één keer kunnen slaan)?
2. Zo ja, op hoeveel verschillende manieren kan dat?

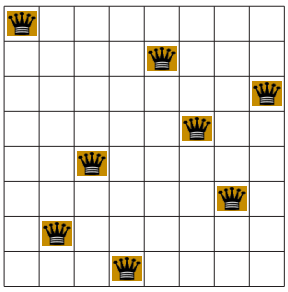
En nu **algemeen**:

Op hoeveel manieren kun je n dames op een n bij n bord plaatsen zonder dat zij elkaar aanvallen?

Een dame kan in één zet een willekeurig aantal vakjes naar links, rechts, onder, boven of diagonaal schuiven.



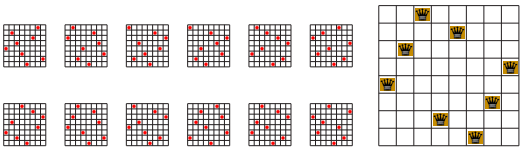
Een oplossing is onderstaande configuratie:



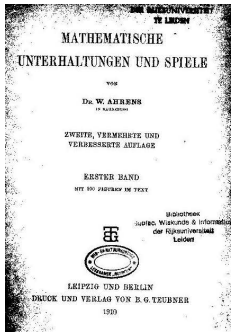
dit correspondeert met de volgende permutatie:

1 5 8 6 3 7 2 4

Op het 8×8 schaakbord zijn er 92 oplossingen. In essentie zijn er 12 verschillende oplossingen, waaruit je door draaien en spiegelen (8 mogelijkheden) ze allemaal kunt maken. Er is één wat meer symmetrische oplossing.



n	aantal	echt aantal	$n!$
1	1	1	1
2	0	0	2
3	0	0	6
4	2	1	24
5	10	2	120
6	4	1	720
7	40	6	5040
8	92	12	40.320
9	352	46	362.880
10	724	92	3.628.800
11	2680	341	39.916.800
12	14.200	1787	479.001.600
13	73.712	9233	
14	365.596		



MATHEMATISCHE
UNTERHALTUNGEN UND SPIELE

von

Dr. W. AHRENS

ZWEITE, VERMEHRTE UND VERBESSERTE AUFLAGE

ERSTER BAND

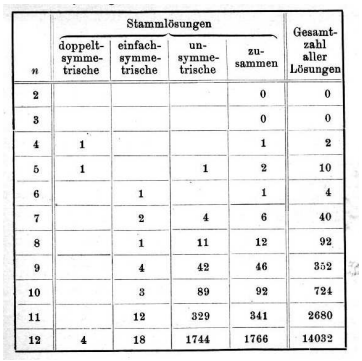
MIT 100 FIGUREN IM TEXT

LEIPZIG UND BERLIN
DRUCK UND VERLAG VON B. G. TEUBNER
1910

Kapitel IX.

Das Achtköniginnenproblem.

§ 1. Historische Einleitung.



n	Stammlösungen: doppelt-symmetrische	Stammlösungen: einfach-symmetrische	Stammlösungen: unsymmetrische	Stammlösungen: zusammen	Gesamtzahl aller Lösungen
2				0	0
3				0	0
4	1			1	2
5	1		1	2	10
6		1		1	4
7		2	4	6	40
8		1	11	12	92
9		4	42	46	352
10		3	89	92	724
11		12	329	341	2680
12	4	18	1744	1766	14032

Een brute force (exhaustive search) aanpak:

> Genereer alle mogelijke configuraties van n dames op een n bij n bord, en controleer van elk daarvan of de dames elkaar al dan niet aanvallen.

Het aantal te controleren kandidaatoplossingen is hier exponentieel:

- n^n onder de aanname: één dame per rij
- $n!$ onder de aanname: één dame per rij en één per kolom; dit zijn gewoon alle permutaties van 1 t/m n

Basisidee backtracking

- bouw een oplossing stap voor stap op en controleer steeds of de deeloplossing in conflict komt met de restricties (en nog wel tot een oplossing kan leiden)
- op elk moment kun je kiezen uit een aantal mogelijke vervolgstappen; maak een keuze en ga langs die weg verder met het opbouwen van de oplossing
- als een keuze op niets uitloopt, herzie je deze keuze en probeer je een andere mogelijkheid

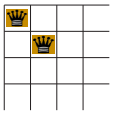
Vergelijk

- het vinden van de uitgang in een doolhof: loop steeds verder en als je bij het zoeken vastloopt, ga terug op je pad om het laatste open alternatief te proberen

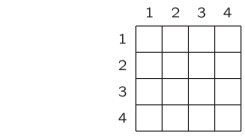
Backtracking versus exhaustive search

Exhaustive search bekijkt *alle* volledige kandidaatoplossingen. Dat zijn hier alle permutaties van 1 t/m n.
Backtracking controleert telkens van deeloplossingen of ze nog aan de eisen/restricties voldoen; zo niet, dan weet je zeker dat alle uitbreidingen van deze deeloplossing ook niet voldoen, dus die hoef je dan niet meer expliciet te bekijken. *Soms* spaar je zo heel veel uit.

Voorbeeld:



Alle $(n-2)!$ kandidaatoplossingen met de eerste twee dames op de aangegeven posities behoeven niet verder onderzocht te worden!



oplossing 1

oplossing 2

Alle oplossingen voor het n bij n bord kunnen we vinden met behulp van backtracking.

- plaats de dames een voor een
- probeer de dame in alle kolommen:
  - als ze geplaatst kan worden, ga dan op dezelfde manier verder met de *volgende* dame
  - zo niet: probeer haar in de volgende kolom (keuze herzien)
- als ze nergens geplaatst kan worden, verschuif dan de *vorige* dame: eerdere keuze herzien!

Als de `rij`-de dame in alle n kolommen is geprobeerd, wordt de dame uit de vorige rij herzien, dus een plek naar rechts gezet, etc, . . .

Bij de recursieve oplossing wordt automatisch een niveau teruggesprongen, bij de iteratieve oplossing moeten we dit zelf expliciet doen.

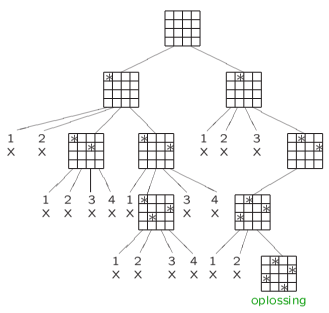

```
void zetdames (int n, int rij, int stand[], int & aantal) {
 // probeert de rij-de dame neer te zetten; de eerste rij-1
 // dames staan al goed: backtracking met recursie
  int kolom;
  if (rij == n+1) {
    drukaf (n, stand); // druk goede stand af
    aantal++; // en tel het aantal goede standen
  }  // if
  else
    for (kolom = 1; kolom <= n; kolom++) {
      stand[rij] = kolom;
      if (geenaanval (rij, stand))
        zetdames (n, rij+1, stand, aantal);
    }  // for
}  // zetdames
```



```
#include <iostream>
using namespace std;
const int MAX = 20;
bool geenaanval (int rij, int stand[ ]) {
  bool veilig = true; int hulprij = 1;
  while (veilig && (hulprij < rij)) {
    veilig = ((stand[rij] != stand[hulprij]) && (stand[rij]-stand[hulprij] !=
              rij-hulprij) && (stand[rij]-stand[hulprij] != hulprij-rij));
    hulprij++; }//while
  return veilig; }//geenaanval
void drukaf (int n, int stand[ ]) {
  for (int i = 1; i <= n; i++) cout << stand[i] << " ";
  cout << endl; }//drukaf
void zetdames (int n, int rij, int stand[ ], int & aantal) {
  int kolom;
  if (rij == n+1) { drukaf (n, stand); aantal++; }//if
  else
    for (kolom = 1; kolom <= n; kolom++) { stand[rij] = kolom;
      if (geenaanval (rij, stand)) zetdames (n, rij+1, stand, aantal); }//for
}//zetdames
int main ( ) {
  int stand[MAX]; int grootte; int teller = 0;
  do {
    cout << "Grootte schaakbord ( < " << MAX << " ) .. "; cin >> grootte;
  } while (grootte < 1 || grootte >= MAX);
  zetdames (grootte, 1, stand, teller);
  cout << endl << "Aantal: " << teller << endl << endl; return 0;
}//main
```


Het kan natuurlijk ook **niet-recursief**:

```
void zetdames (int n) {  // niet recursief
  int stand[MAX];
  int rij = 1; stand[1] = 0;  // zet eerste dame klaar
  while (rij > 0) {
    stand[rij]++;  // volgende kolom
    while ((stand[rij] <= n) && (!geenaanval (rij, stand)))
      stand[rij]++;  // eerste de beste goede kolom
    if (stand[rij] <= n)
      if (rij == n)  // n-de dame gezet
        drukaf (n, stand);
      else {  // nog niet alle dames gezet
        rij++;
        stand[rij] = 0;
      }
    else  // alle kolommen van een rij geprobeerd
      rij--;  // vorige dame herzien
  } // while
} // zetdames
```

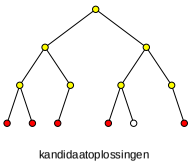



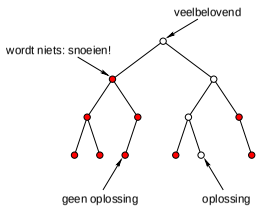
x: deeloplossing niet verder uitbreiden, keuze herzien

Om de werking van backtracking te *beschrijven* kunnen we een state-space *tree* (toestand-actie-*boom*) gebruiken.

- speciaal soort toestandsruimte
- toestand (=knoop) $\Longleftrightarrow$ deeloplossing;
  actie (=tak) $\Longleftrightarrow$ keuze uitbreiding deeloplossing
- blad $\Longleftrightarrow$ (kandidaat)oplossing
- pad van wortel naar blad $\Longleftrightarrow$ stap-voor-stap-constructie van (kandidaat)oplossing

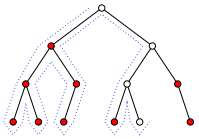
Exhaustive search (met stap-voor-stap-constructie van kandidaatoplossingen) doorloopt de hele boom en evalueert alle bladeren. Backtracking stopt met het doorlopen van een subboom bij een knoop als de betreffende knoop nooit tot een oplossing kan leiden (en backtrackt dan naar de ouder van die knoop om van daaruit verder te zoeken).



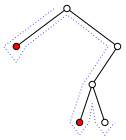


Exhaustive search: de hele boom wordt bekeken (tot een goede oplossing is gevonden)

Backtracking: hele subbomen kunnen soms worden gesnoeid



Backtracking doorzoekt de state-space tree via depth first search.



Als het meezit wordt er flink gesnoeid.

De volledige toestandsruimte (state space) is exponentieel:

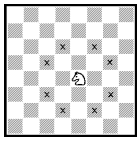
- 1 begintoestand (leeg bord)
- $n!$ eindtoestanden (n dames geplaatst)(*)
- $n+n*(n-1)+n*(n-1)*(n-2)+\ldots+n*(n-1)*\ldots*2$ tussengelegen toestanden (de eerste 'zoveel' dames geplaatst)(*)

Backtracking zal voor grote probleeminstanties toch exponentieel zijn.

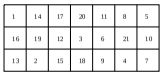
(*) We bekijken alleen toestanden met hooguit één dame per rij en hooguit één per kolom.

Vraag:

Hoeveel verschillende series van $m*n-1$ sprongen van het paard zijn er op een m bij n bord, zodat elk veld precies één keer bezocht wordt?



beweging van het paard



een oplossing op het 3 bij 7 bord

- **Lezen/leren bij dit college:**
  Paragraaf 3.4, 3.5, sheets, 12 inl., 12.1
- **Werkcollege** brute force en backtracking
  donderdag 8 maart 2012, 13:45–15:30, in zaal Plein
- **Opgaven:**
  zie http://www.liacs.nl/home/rvvliet/algoritmiek/
- **Volgend college:**
  donderdag 22 maart 2012
  dus geen college op 15 maart !