Vijfde college algoritmiek

8 maart 2013

Exhaustive search en Backtracking

Brute Force: los een probleem op via de meest voor de hand liggende (recht-toe-recht-aan) methode, meestal door eenvoudigweg de definitie van een oplossing te gebruiken. Vaak ook: alle mogelijkheden proberen.

Voorbeelden uit vorige colleges:

- Lineair zoeken
- Oplossen van een cijferpuzzel
- Bepaling van de ggd van twee getallen
- Selection sort
- Polynoomevaluatie
- Patroonherkenning
- Closest pair
- Convex hull

Brute force:

- **Voordelen:**
  - algemeen toepasbaar
  - eenvoudig
  - levert voor een aantal belangrijke problemen (zoeken, patroonherkenning) een zeer behoorlijk algoritme op
- **Nadelen:**
  - levert meestal geen efficiënt algoritme op
  - soms onacceptabel langzaam

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 deelverzamelingen van de getallen 1 t/m n

. evalueer elk van deze mogelijke oplossingen

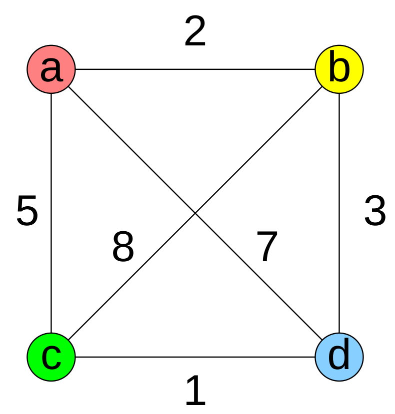
. retourneer een/de kandidaatoplossing met de gevraagde eigenschap (als die bestaat) (*)

(*) 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 (complete) graaf.

Voorbeeld:



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

Route	**Lengte**
a $\rightarrow$ b $\rightarrow$ c $\rightarrow$ d $\rightarrow$ a	$2+8+1+7=18$
a $\rightarrow$ b $\rightarrow$ d $\rightarrow$ c $\rightarrow$ a	$2+3+1+5=11$
a $\rightarrow$ c $\rightarrow$ b $\rightarrow$ d $\rightarrow$ a	$5+8+3+7=23$
a $\rightarrow$ c $\rightarrow$ d $\rightarrow$ b $\rightarrow$ a	$5+1+3+2=11$
a $\rightarrow$ d $\rightarrow$ b $\rightarrow$ c $\rightarrow$ a	$7+3+8+5=23$
a $\rightarrow$ d $\rightarrow$ c $\rightarrow$ b $\rightarrow$ 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.

Assignmentproblem (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 i.h.a. **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)

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 (stap voor stap) 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 deeloplossing 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 volledig en controleer van elk daarvan of deze **stijgend** is en bepaal welke de **langste** van deze deelrijen 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/eisen, 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)?

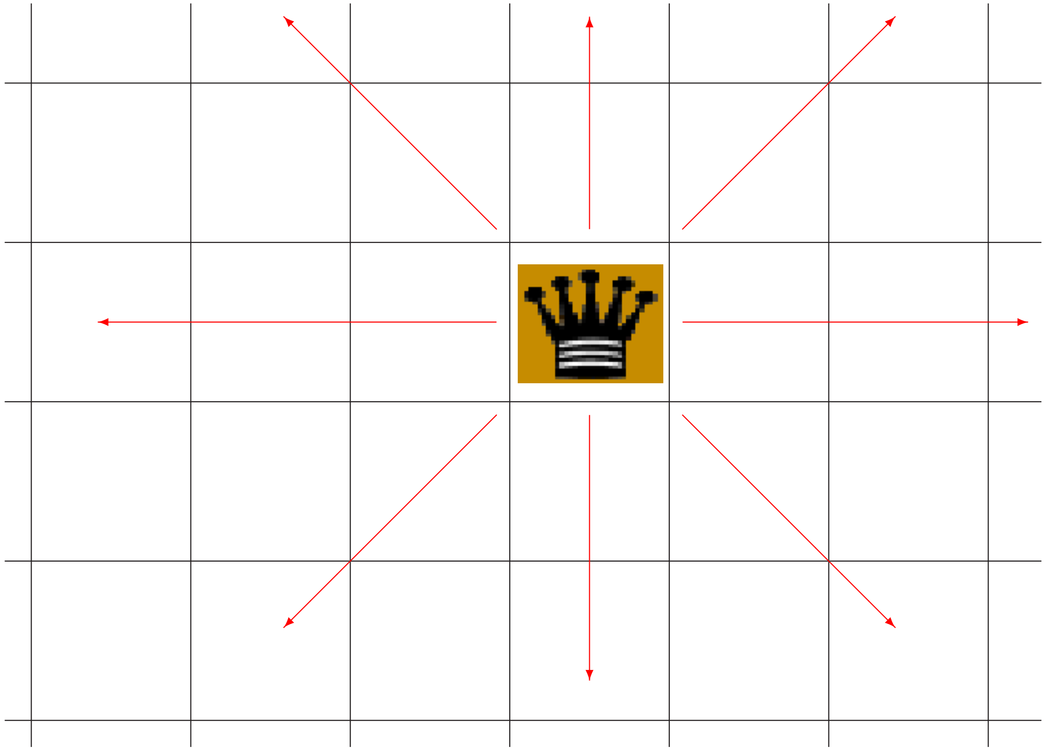
2. Zo ja, op hoeveel verschillende manieren kan dat?

En nu **algemeen**:

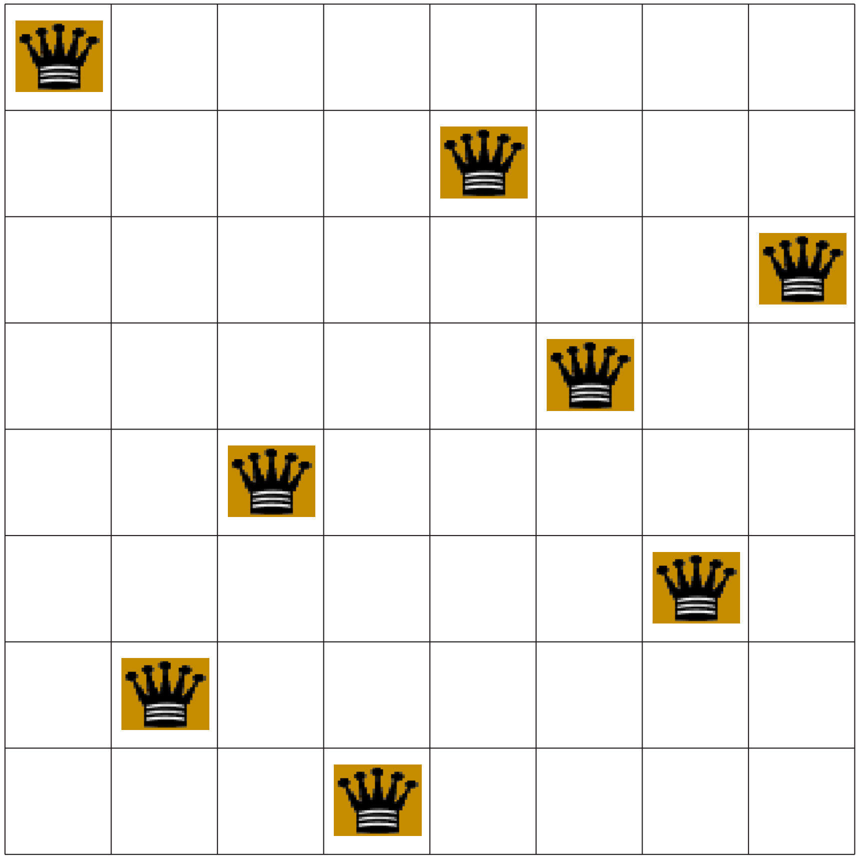
Op hoeveel manieren kun je n dames op een n bij n bord plaatsen zonder dat zij elkaar aanvallen?

Beweging van de dame

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



Een oplossing is onderstaande configuratie:

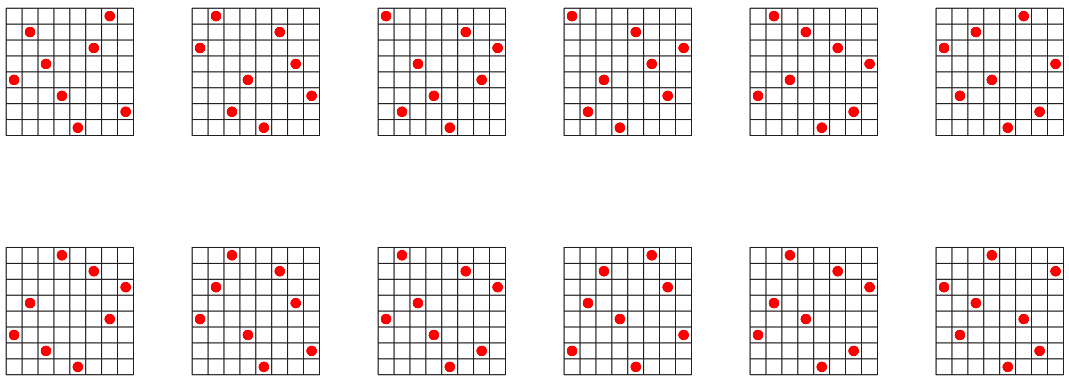


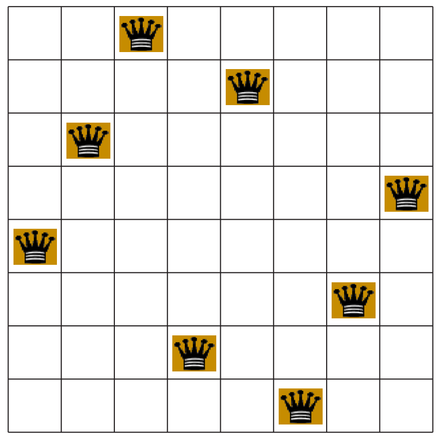
dit correspondeert met
de volgende permutatie:

1 5 8 6 3 7 2 4

Alle oplossingen

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.





Aantallen oplossingen

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		

W. Ahrens, 1910

MATHEMATISCHE
UNTERHALTUNGEN UND SPIELE

VON

DR. W. AHRENS
IN MAGDEBURG

ZWEITE, VERMEHRTE UND
VERBESSERTE AUFLAGE

ERSTER BAND
MIT 200 FIGUREN IM TEXT

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

Kapitel IX.

Das Achtköniginnenproblem.

Jean Paul,
„Die unsichtbare Loge". Erster Sektor.

Aus einem Gedicht des Muhammed ibn Scheref.
übers. v. Hammer-Purgstall

§ 1. **Historische Einleitung.**

In der „Illustrirten Zeitung" vom 1. Juni 1850 (Nr. 361, 14. Bd., p. 352) findet sich unter der Rubrik „Schach" „Eine in das Gebiet der Mathematik fallende Aufgabe von Herrn Dr Nauck in Schleusingen" folgenden Inhalts: „**Man kann 8 Schachfiguren, von denen jede den Rang einer Königin hat, auf dem Brett so aufstellen, daß keine von einer anderen geschlagen werden kann.**"[†]) In der Nummer vom 21. September

*) „Wesir" für unser „Königin". Ich entnehme dies Wort aus A. v. d. Linde, „Geschichte u. Litteratur des Schachspiels", II, p. 257.

†) Irrtümlicherweise wird diese Stelle zumeist als das erste Vorkommen unseres Problems zitiert. Die Aufgabe ist jedoch bereits in der Schachzeitung, herausgegeben von der Berliner Schachgesellschaft, Bd. III, 1848, p. 363 von einem ungenannten „Schachfreund" gestellt worden, und zwar war, wie Max Lange („Handbuch der Schachaufgaben", Leipzig 1862, p. 30, Anm. 6) „nach einer directen persönlichen Mittheilung" angibt, dieser „Schachfreund" Max Bezzel in Ansbach. — Wenn wir trotzdem oben die Geschichte des Problems an jene Naucksche Behandlung anknüpfen, so bestimmt uns hierbei der Umstand, daß die Fragestellung in der „Schachzeitung" zunächst nur 2 spezielle Lösungen (s. Schachzeitung IV, 1849, p. 40) gezeitigt und anscheinend überhaupt kein sonderliches Interesse für unser Problem

14*

n	Stammlösungen: doppelt-symmetrische	Stammlösungen: einfach-symmetrische	Stammlösungen: un-symmetrische	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/eisen, 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

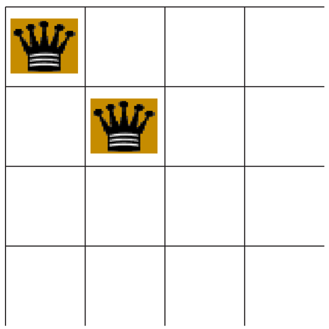
Backtracking versus exhaustive search

Exhaustive search bekijkt *alle* volledige kandidaatoplossingen. Dat zijn hier alle permutaties van 1 t/m n.

Backtracking controleert telkens van deeloplossingen (standen waarbij nog niet alle dames op het bord (hoeven te) staan) 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:

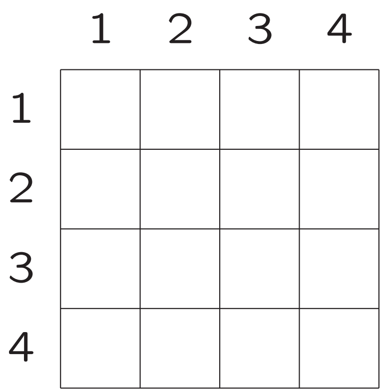


Alle $(n-2)!$ kandidaatoplossingen met de eerste twee dames op de aangegeven posities behoeven niet verder onderzocht te worden!

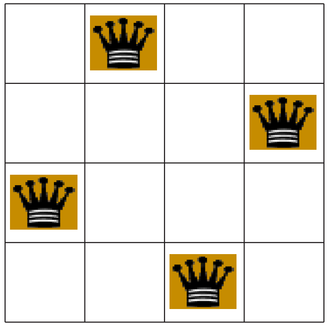
De eerste twee dames vallen elkaar immers al aan.

Nog meer dames neerzetten heeft dus geen zin

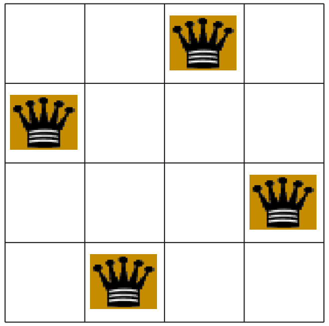
Oplossingen dames: $n = 4$



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
```

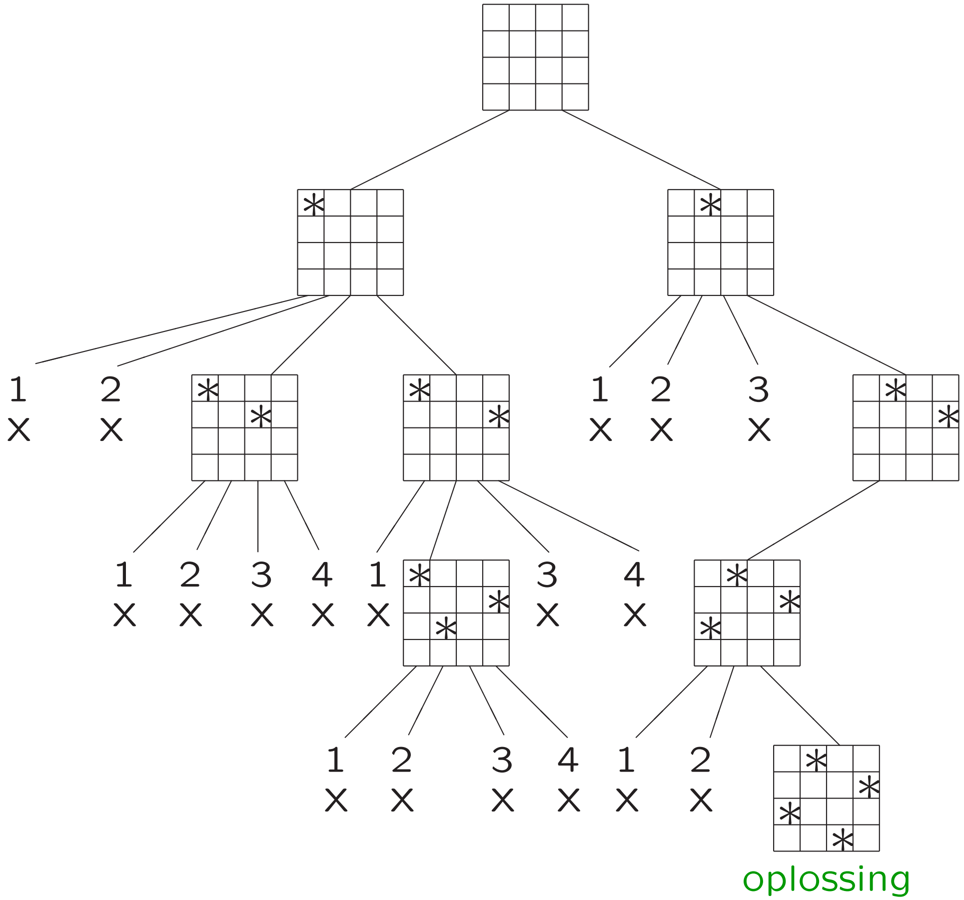
```
bool geenaanval (int rij, int stand[ ]) {
  bool veilig = true; int hulprij = 1;
  while (veilig && (hulprij < rij)) {
    veilig = ((stand[rij] != stand[hulprij]) &&
              (stand[rij]-stand[hulprij] != rij-hulprij) &&
              (stand[rij]-stand[hulprij] != hulprij-rij));
    hulprij++;
  }//while
  return veilig;
}//geenaanval
```

```
#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 << 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
```

State-space tree dames



x: deeloplossing niet verder uitbreiden, keuze herzien

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.

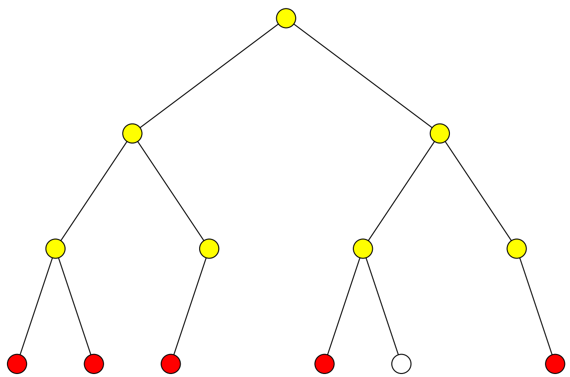
Om de *werking* van backtracking te *beschrijven* kunnen we een state-space *tree* (toestand-actie-*boom*) gebruiken.

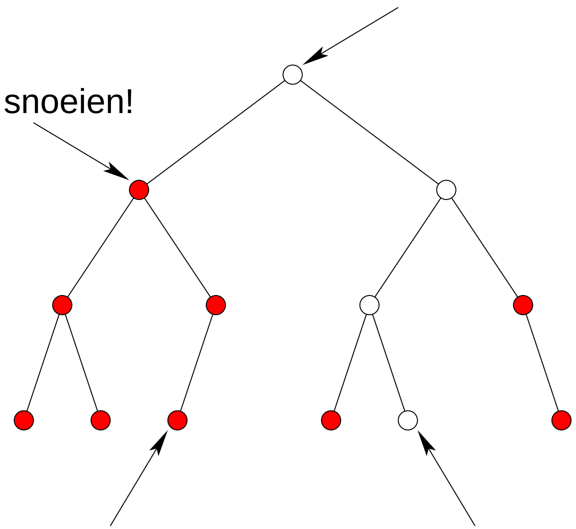
- speciaal soort toestandsruimte
- toestand (=knoop) $\Longleftrightarrow$ deeloplossing; actie (=tak) $\Longleftrightarrow$ keuze uitbreiding deeloplossing
- blad $\Longleftrightarrow$ (kandidaat)oplossing
- pad van wortel naar blad $\Longleftrightarrow$ stap-voor-stap-constructie van (kandidaat)oplossing

Exhaustive search (met stap-voor-stap-constructie van kandidaatoplossingen) doorloopt de hele state-space tree 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).

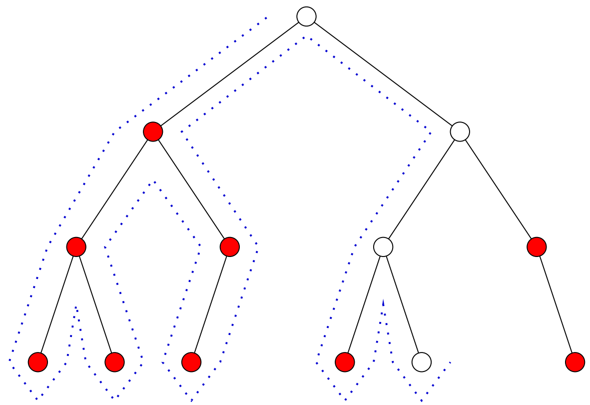
State-space tree -1-



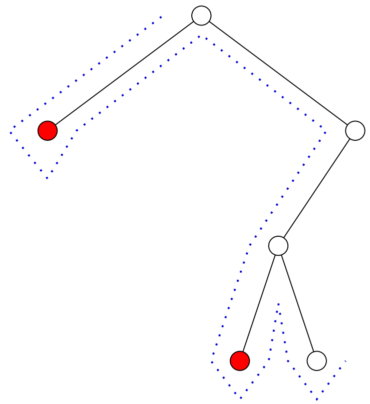


Exhaustive search: de hele boom wordt bekeken (tot een goede oplossing is gevonden)

Backtracking: hele subbomen kunnen soms worden gesnoeid



Backtracking doorzoekt de state-space tree via depth first search (zie later).

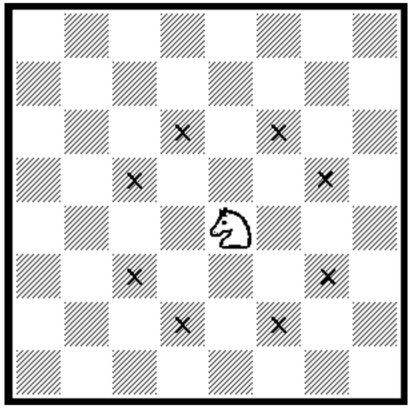


Als het meezit wordt er flink gesnoeid.

Opmerking: backtracking is niet alleen maar een verbeterde versie van exhaustive search. Het kan algemener toegepast worden (doolhof, paardensprong)

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

1	14	17	20	11	8	5
16	19	12	3	6	21	10
13	2	15	18	9	4	7

een oplossing op het 3 bij 7 bord

Genereer alle permutaties van 1 t/m n met backtracking.

```
void permutaties(int n, int perm[ ], int hierzo) {
    int i;
    if (hierzo == n+1)
        drukaf(perm,n); // permutatie gevonden
    else {
        for (i=1; i<=n; i++) {
            perm[hierzo] = i;
            if (!aanwezig(perm, i, hierzo))  // test of i
            // al in perm[1] t/m perm[hierzo-1] voorkomt
               permutaties(n, perm, hierzo+1);
        } // for
    } // else
} // permutaties
```

Vraag: wat heeft dit met torens op een schaakbord te maken?

- **Lezen/leren bij dit college:**
  Paragraaf 3.4, sheets, 12 inleiding, 12.1

- **Werkcollege** brute force en backtracking:
  donderdag 14 maart 2013, 13:45–15:30, in zaal 412/B2

- **Opgaven:**
  zie http://www.liacs.nl/home/graaf/ALGO/

- **Volgend college:**
  vrijdag 15 maart 2013

- **Vragenuur ??**