# Vijfde college algoritmiek

## 9 maart 2007

## Backtracking

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.

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 `3, 1, 4, 1, 5, 9, 2, 6, 5, 7`.
Gevraagd: de/een langste *stijgende* deelrij.

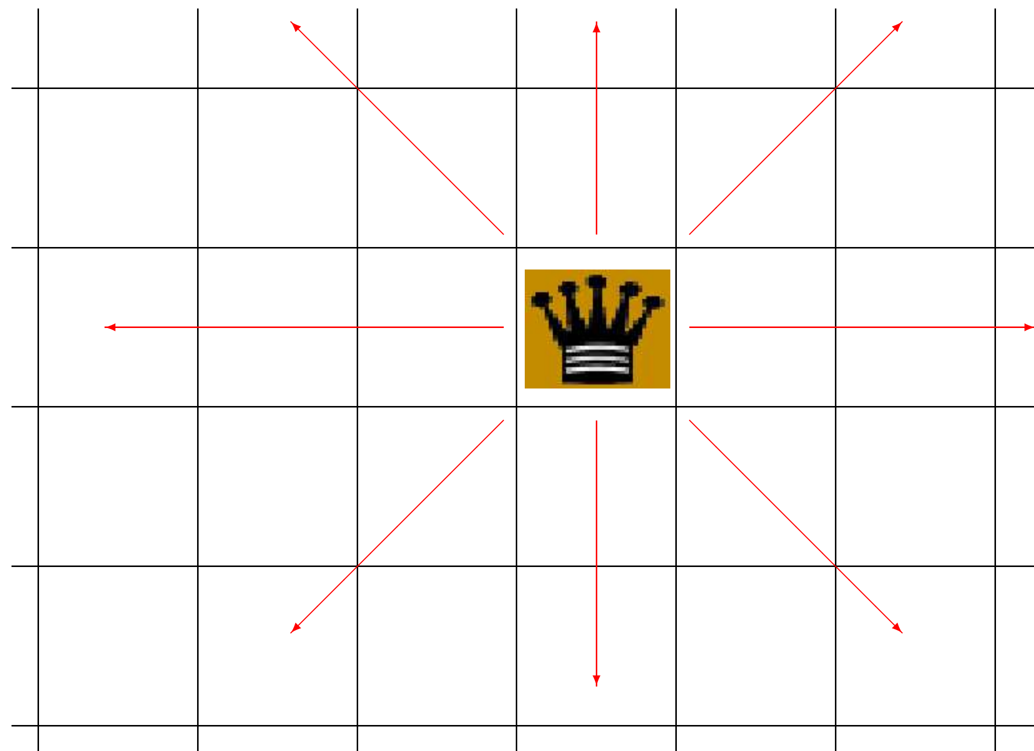
Het acht koninginnenprobleem luidt als volgt:

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?

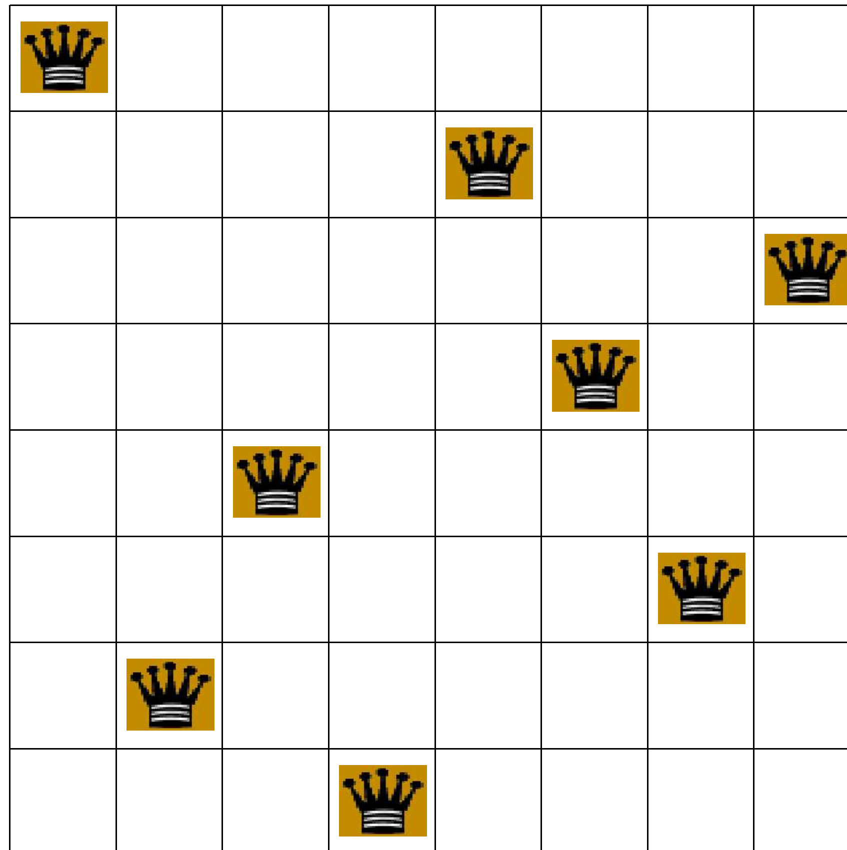
En nu **algemeen**:

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.



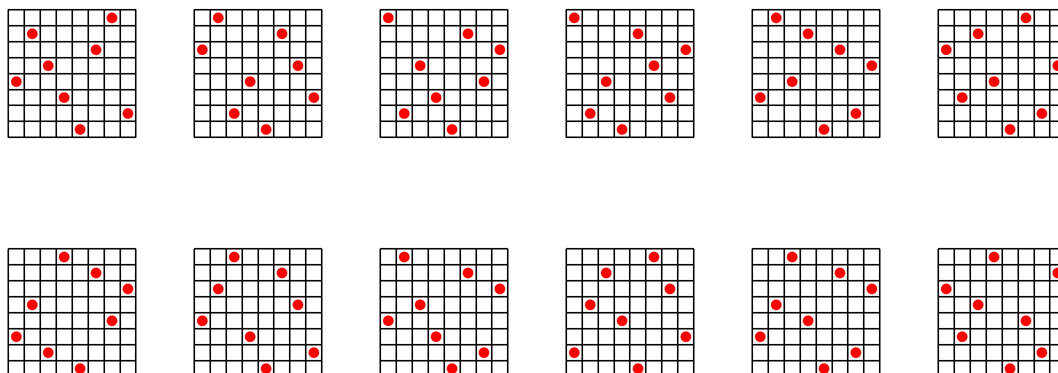
Een oplossing is onderstaande configuratie:

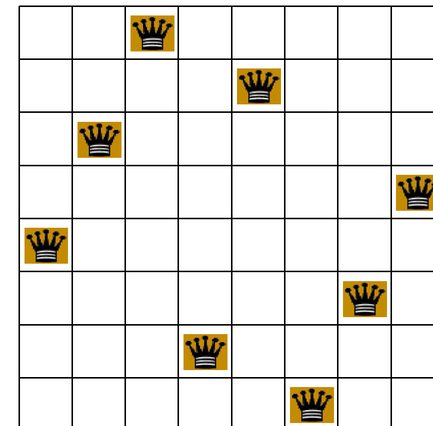


dit correspondeert met
de volgende permutatie:

1 5 8 6 3 7 2 4

Op het $8 \times 8$ 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.

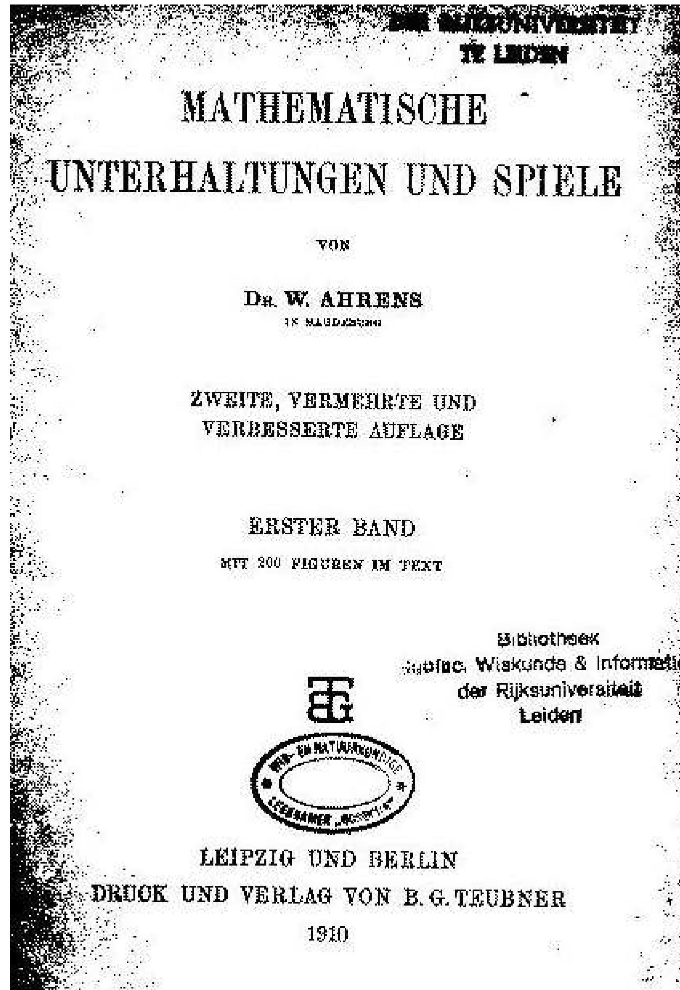




## Aantallen oplossingen

| $n$ | 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



RIJKSUNIVERSITEIT TE LEIDEN

MATHEMATISCHE UNTERHALTUNGEN UND SPIELE

VON

DR. W. AHRENS
IN MAGDEBURG

ZWEITE, VERMEHRTE UND VERBESSERTE AUFLAGE

ERSTER BAND

MIT 200 FIGUREN IM TEXT

Bibliotheek
Instituut Wiskunde & Informatica
der Rijksuniversiteit
Leiden

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

Kapitel IX.

**Das Achtköniginnenproblem.**

*Ein guter Mathematiker ist ein guter Schachspieler.*
Jean Paul,
„Die unsichtbare Loge". Erster Sektor.

*Die Schachspieler sind gewöhnliche Leute, die treiben keine Poesie.* [illegible]

*Wenn auf dem Schachbrett Bauern zu Ferzen*) werden,*
*So sind dafür Vesire Bauern hier auf Erden.*
Aus einem Gedicht des Muhammed ibn Scheraph.
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

*) „Vesir" 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 direkten persönlichen Mittheilung" angibt, dieser „Schachfreund" Max Bezzel in Ansbach. — Wenn wir trotzdem aber die Geschichte des Problems an jene Nauck'sche 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*

# Aantallen oplossingen

| n | Stammlösungen: doppelt-symmetrische | einfach-symmetrische | un-symmetrische | 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
- 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

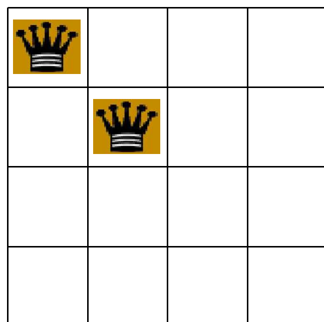
**Vergelijk**

- 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.
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. *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!

## Oplossingen dames: $n = 4$

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 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

- 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
```

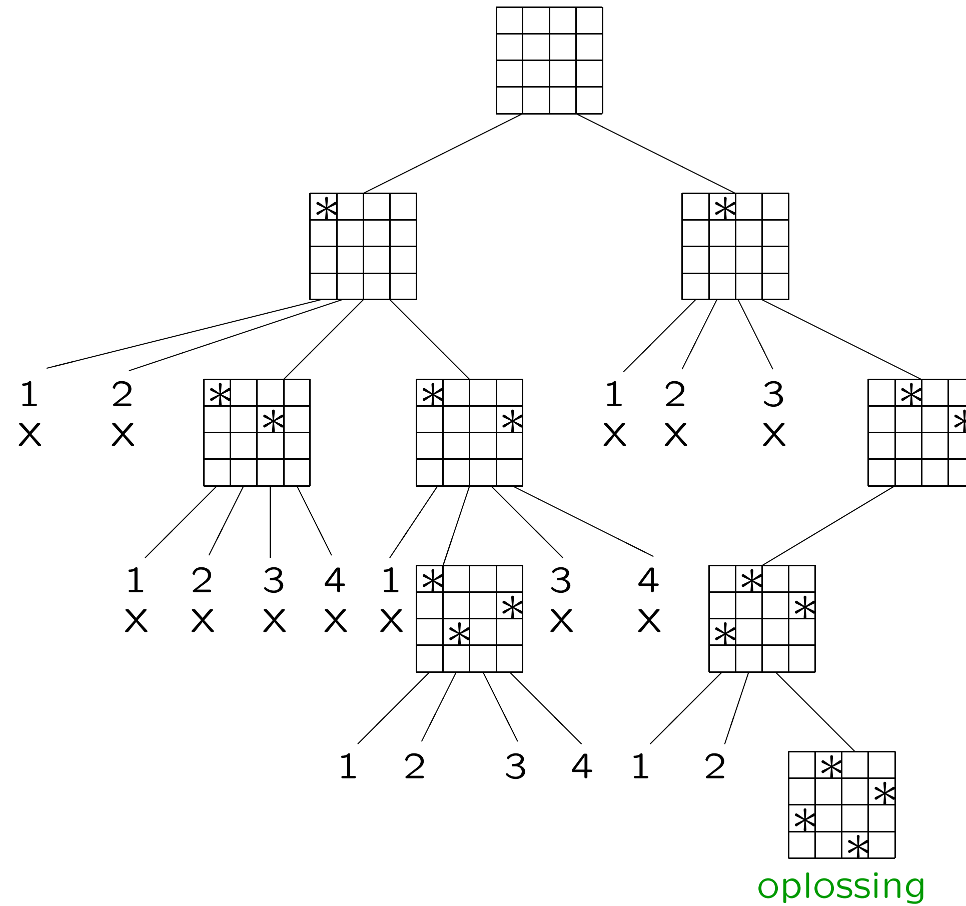
## Het hele programma

```
#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
```

# State-space tree dames



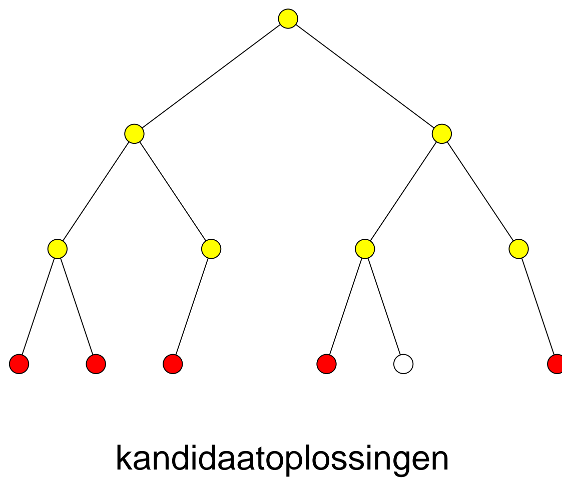

x: deeloplossing niet verder uitbreiden

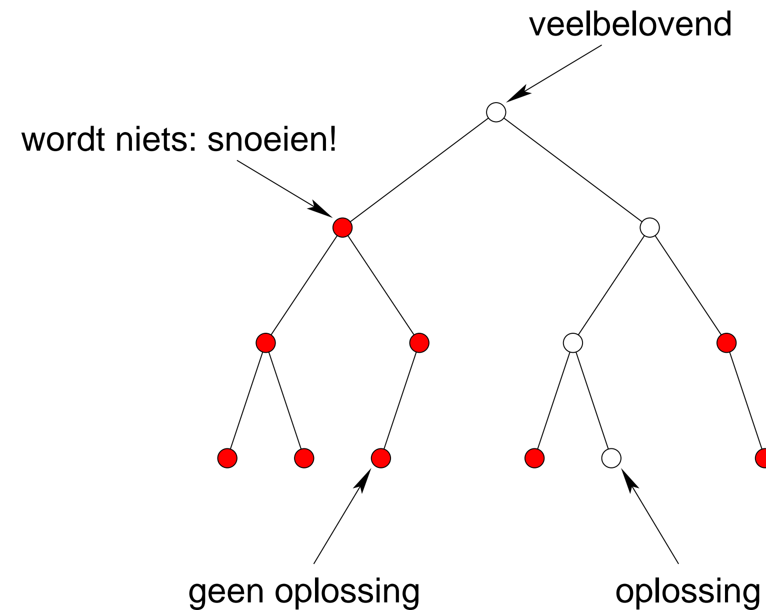
Om de werking van backtracking te *beschrijven* kunnen we een state-space *tree* (toestand-actie-*boom*) gebruiken.

- speciaal geval van een 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 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).

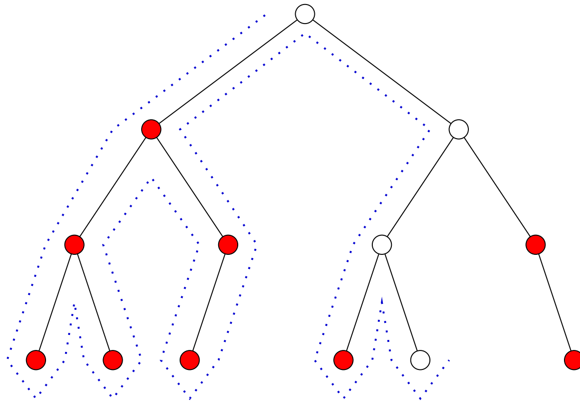
# State-space tree -1-



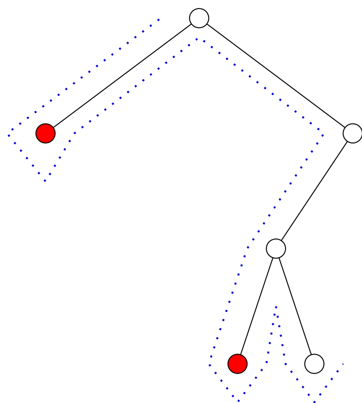




Exhaustive search: de hele boom wordt bekeken (tot een goede oplossing is gevonden)

Backtracking: hele subbomen kunnen soms worden gesnoeid



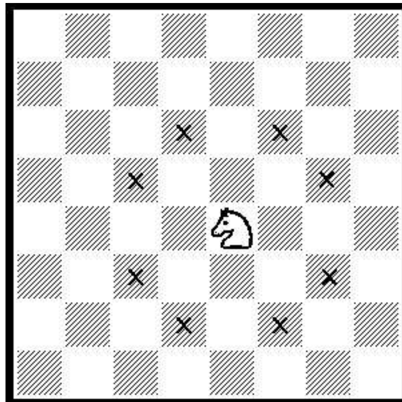
Backtracking doorzoekt de state-space tree via depth first search.



Als het meezit wordt er flink gesnoeid.

**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?



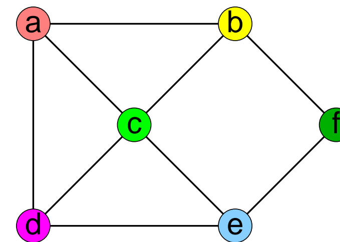
beweging van het paard

| 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

**Definitie**: Een Hamiltonkring in een (ongerichte) graaf is een kring die elke knoop precies één keer aandoet.

**Voorbeeld**: a b f e c d is een Hamiltonkring in nevenstaande graaf, echter a b c d e f is geen Hamiltonkring.



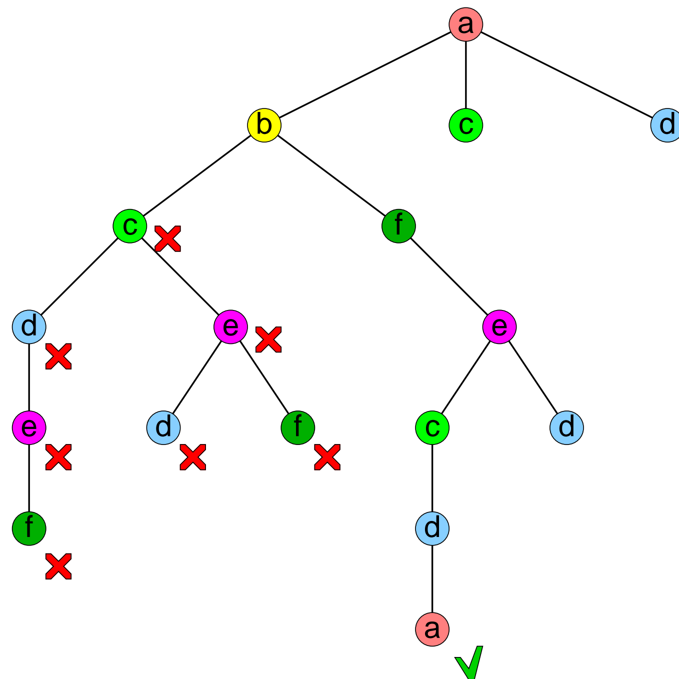

**Probleem**: vindt een Hamiltonkring in een gegeven ongerichte graaf.

Exhaustive search: genereer alle $(n-1)!$ kandidaatoplossingen (permutaties van de knopen) en controleer daarna van elk of het een Hamiltonkring voorstelt.

Backtracking: genereer de mogelijke Hamiltonkringen stap voor stap en controleer tijdens de constructie al of de deelpermutatie wel een pad/kring voorstelt. Kies daartoe in elke uitbreidingsstap steeds de eerste **buurknoop**. Blijkt die keuze toch (verderop in de constructie) op niets uit te lopen, kies dan de volgende buurknoop. Als er geen buurknopen meer zijn kan de deeloplossing blijkbaar niet meer uitgebreid worden en moet je je vorige keuze hetzien.

# State space tree HC

Gebruik van backtracking voor het vinden van een Hamiltonkring in de voorbeeldgraaf leidt tot de volgende state space tree:




- breid het pad telkens met één knoop uit (keuzes in alfabetisch volgorde)
- in de knopen met een rood kruis backtrackt het algoritme zodra blijkt dat die niet tot een oplossing leiden

**Probleem**:
Gegeven een verzameling $S = \{s_1, s_2, \ldots s_n\}$ van positieve ($> 0$) gehele getallen en een geheel getal $d$. Vindt een deelverzameling (of alle deelverzamelingen) van $S$ waarvan de som der getallen gelijk is aan $d$.

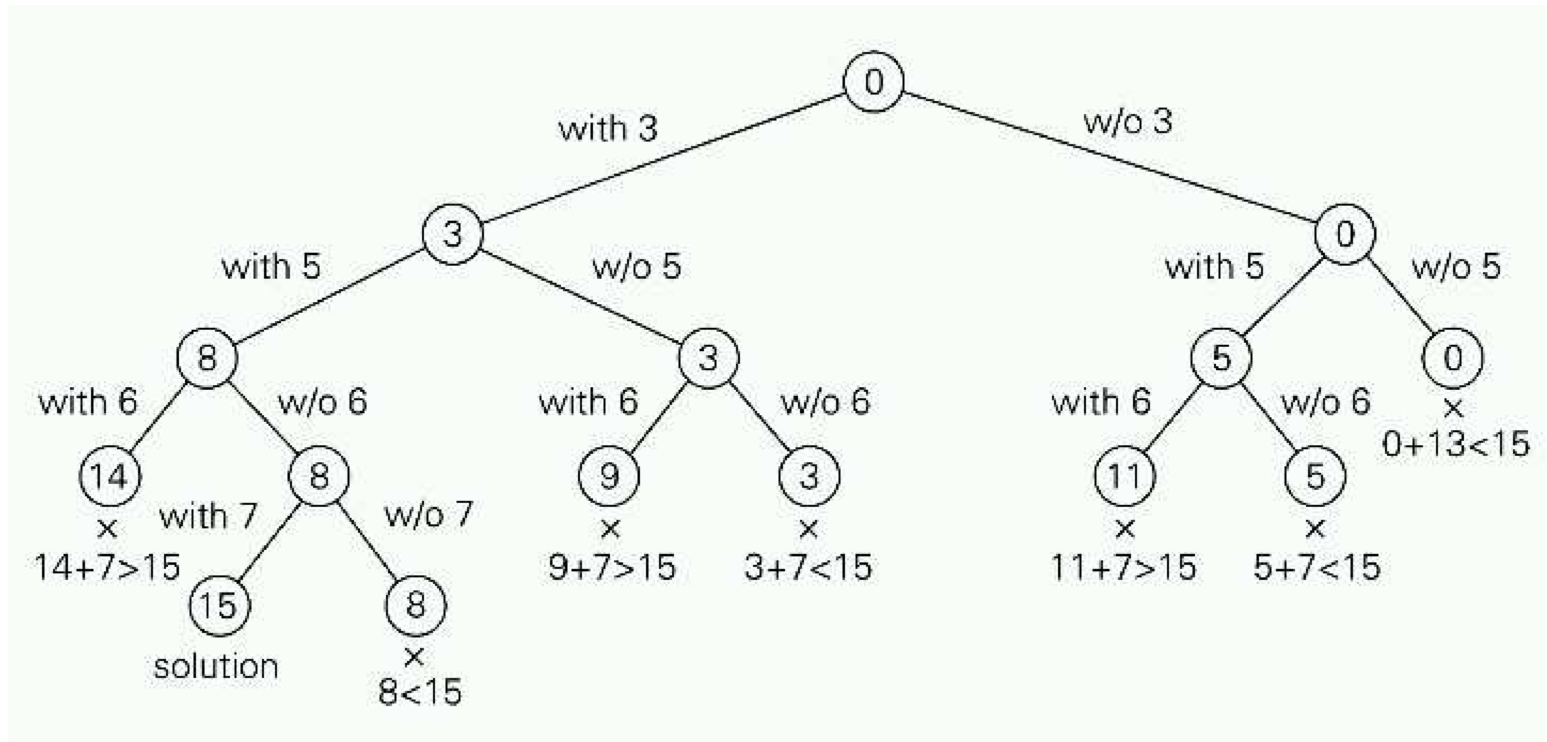
**Voorbeeld**:
$S = \{1, 2, 5, 6, 8\}$ en $d = 9$. Er zijn twee oplossingen, namelijk $\{1, 2, 6\}$ en $\{1, 8\}$.

Exhaustive search: genereer alle $2^n$ deelverzamelingen van $S$ en controleer of hun som geljk is aan $d$.

De stap-voor-stap constructie van deeloplossingen doen we (bijvoorbeeld) zo: gegeven een deeloplossing (= een veelbelovende deelverzameling van $\{s_1, s_2, \ldots, s_i\}$), dan zijn er twee mogelijke vervolgstappen: óf $s_{i+1}$ wordt toegevoegd, óf $s_{i+1}$ wordt niet toegevoegd.

**Backtracking** bekijkt zo ook alle deelverzamelingen, maar hoeft ze niet allemaal volledig te genereren. Een (veelbelovende) deelverzameling van $\{s_1, s_2, \ldots, s_i\}$ met som $s'$ hoeft niet verder uitgebreid te worden als $s' + s_{i+1} > d$ (*), of als $s' + \Sigma_{j=i+1}^{n} s_j < d$.

**Opmerking**:
Als (*) geldt levert elke uitbreiding, met welke $s_j$ ($j \geq i+1$) dan ook een te grote totaalsom op. Dus herzie je vorige keuze. Hier is gebruikt dat $S$ oplopend gesorteerd is.




Volledige state-space tree bij toepassing van backtracking op probleeminstantie $S = \{3, 5, 6, 7\}$ en $d = 15$. De knopen stellen deeloplossingen voor. Het getal in een knoop is de som van de $s_j$ uit de corresponderende deelverzameling. De ongelijkheid onder een blad geeft aan waarom daar backtracking plaatsvindt.

- **Werkcollege**

  donderdag 15 maart 2007 in zaal 306/308: eerste programmeeropdracht

- **Opgaven:**

  zie http://www.liacs.nl/home/graaf/ALGO/algo2007.html

- **Volgend college:**

  vrijdag 16 maart 2007