

## Vijfde college algoritmiek

7 maart 2013

# Exhaustive search en Backtracking

1

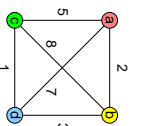
### Traveling Salesman Problem (handelsreizigersprobleem)

**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.

**Opmerking**: 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)



3

### Knappzakprobleem

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

#### Voorbeeld:

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

knappzakcapaciteit: 12

5

### Assignmentprobleem (toewijzingsprobleem)

**Gegeven**  $n$  personen en  $n$  taken (jobs). Persoon  $i$  kan taak  $j$  doen voor kosten  $c[i][j]$  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

$n = 4$

7

**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  $1 \dots n$
- 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

2

### 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.

4

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.

6

	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

$1,2,3,4 \rightarrow 9+4+1+4 = 18$        $2,3,1,4 \rightarrow \dots$        $3,4,1,2 \rightarrow \dots$   
 $1,2,4,3 \rightarrow 9+4+6+9 = 30$        $2,3,4,1 \rightarrow \dots$        $3,4,2,1 \rightarrow \dots$   
 $1,3,1,2,4 \rightarrow 9+3+6+4 = 24$        $2,4,1,3 \rightarrow \dots$        $4,1,1,2,3 \rightarrow \dots$   
 $1,3,4,1,2 \rightarrow 9+3+6+6 = 26$        $2,4,3,1,1 \rightarrow \dots$        $4,1,1,2,3 \rightarrow \dots$   
 $1,4,1,2,3 \rightarrow 9+7+6+9 = 33$        $3,1,1,2,4 \rightarrow \dots$        $4,2,1,3 \rightarrow \dots$   
 $1,4,3,1,2 \rightarrow 9+7+1+6 = 23$        $3,1,1,4,2 \rightarrow \dots$        $4,2,3,1,1 \rightarrow \dots$   
 $2,1,1,3,4 \rightarrow 2+6+1+4 = 13$        $3,2,1,1,4 \rightarrow \dots$        $4,3,1,1,2 \rightarrow \dots$   
 $2,1,4,1,3 \rightarrow 2+6+9+9 = 25$        $3,2,4,1,1 \rightarrow \dots$        $4,3,2,1,1 \rightarrow \dots$

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

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

8

- \* Exhaustive search algoritmen werken **alleen voor kleine probleeminstaties** in acceptabele tijd

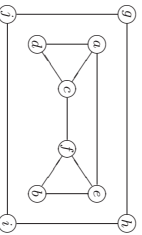
- \* Voor veel problemen zijn er veel efficiëntere algoritmen bekend (Eulerkring, kortste paden, toewijzingsprobleem)

- \* Voor andere problemen is exhaustive search (of varianten daarop) in essentie de enig bekende oplossing (handelsreizigersprobleem, knapzakprobleem)

9

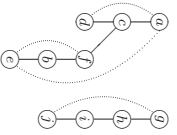
- De wandeling begint in een gegeven knoop  $v$  van de graaf.
- Vanuit een zoielijst bezochte knoop wordt vervolgens steeds een aangrenzende –nog onbezochte– knoop bezocht, en vandaaruit op dezelfde manier verder gelopen tot je niet verder kan.
- In dat geval wordt teruggegaan naar de knoop waar je net vandaan kwam, en wordt een andere aangrenzende knoop daarvan bezocht, en zo verder tot je weer bij  $v$  terug bent.
- Aangrenzende knopen kunnen bijvoorbeeld altijd in alfabetische volgorde bezocht worden.
- Een knoop wordt steeds als reeds bezocht gemarkeerd op het moment dat deze voor de eerste keer bekeken wordt.
- Alle knopen die vanuit  $v$  bereikbaar zijn worden zo precies één keer bezocht. Voor niet-samenhangende grafen moet bovenstaande telkens herhaald worden vanuit een resterende, nog niet bezochte knoop.
- Depth-first-search kan recursief of met behulp van een stapel worden geïmplementeerd.

11



```

e6,2      i10,7
d3,1      f4,4      g8,8
c2,5      h8,9
a1,6      g7,10
    
```



13

```

dfs(v)
// Bezocht recursief alle nog onbezochte knopen die via een pad
// met v zijn verbonden, en nummer 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);
    }
  }
    
```

Er is ook niet-recursieve implementatie, met expliciete stapel

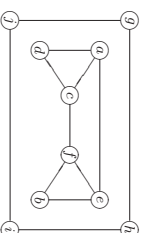
15

- Bij veel (graaf)problemen is het nodig om alle knopen van de graaf op een systematische manier te bezoeken

- **Graafwandelingen:**

1. **Depth-first-search**: te vergelijken met WLR-wandeling bij bomen
2. **Breadth-first-search**: te vergelijken met nivo-orde wandeling bij bomen

10



12

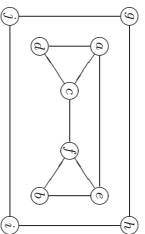
```

ALGORITHM 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);
    }
  }
    
```

14

- De wandeling begint in een gegeven knoop  $v$  van de graaf.
- Vanuit een zoielijst bezochte knoop worden eerst alle aangrenzende –nog onbezochte– knopen bezocht, dan de daaraan grenzende knopen (voor zover nog niet eerder bezocht), en zo verder totdat alle bereikbare knopen bezocht zijn.
- Knopen worden zo bezocht in volgorde van hun afstand vanaf  $v$ .
- Aangrenzende knopen kunnen bijvoorbeeld altijd in alfabetische volgorde bezocht worden.
- Bij de implementatie gebruiken we een rij. In de eerste stap wordt  $v$  gemarkeerd als bezocht en in de rij gezet. In elke volgende stap wordt de voorste knoop uit de rij gehaald, en worden diens buren gemarkeerd als bezocht en in de rij geplaatst.
- Alle knopen die vanuit  $v$  bereikbaar zijn worden zo precies één keer bezocht. Voor niet-samenhangende grafen moet bovenstaande telkens herhaald worden vanuit een resterende, nog niet bezochte knoop.

16



17

```

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
}
    
```

19

	DFS	BFS
Data structuur	een stack	een queue
Aantal volgorde knopen	twee volgorde	een volgorde
Soorten takken (onger. grf)	tree en back edges	tree en cross edges
Toepassingen	samenhang, acycliciteit, articulation points	acycliciteit
Efficiënte voor adl. matrix	$\Theta( V ^3)$	$\Theta( V ^3)$
Efficiënte voor adl. list	$\Theta( V  +  E )$	$\Theta( V  +  E )$

21

**Backtracking** versus **exhaustive search**

Exhaustive search bekijkt *alle* volledige kandidaatoplossingen.

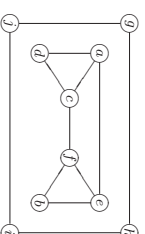
Backtracking controleert telkens van deeloplossingen of ze nog aan de eisen/restricties voldoen. zo niet, dan weet je zeker dat alle uitbreidingen van deze oplossing ook niet voldoen, dus die hoeft je dan niet meer expliciet te bekijken.

**Voorbeeld**

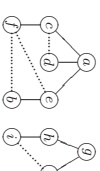
Gegeven de rij  $A = 3, 1, 4, 1, 5, 9, 2, 6, 5, 7$ .

Gevraagd: de/een langste *stijgende* deelrij (met volgorde der elementen als in  $A$  zelf).

23



a1 c2 d3 e4 f5 b6  
g7 h8 j9 i10



18

```

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;
}
    
```

20

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

**Exhaustive search** genereert alle kandidaatoplossingen en haait 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 probleeminstaties oplossen.

22

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

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

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

24

**Basisidee backtracking**

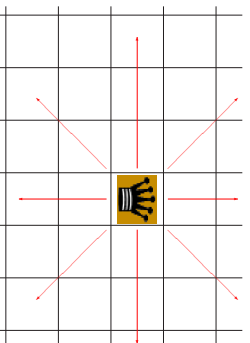
- bouw een oplossing stap voor stap op en controleer steeds of de deeloplossing in conflict komt met de restricties (en nog wel tot een oplossing kan leiden)
- op elk moment kun je kiezen uit een aantal mogelijke vervolgstappen; maak een keuze en ga langs die weg verder met het opbouwen van de oplossing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

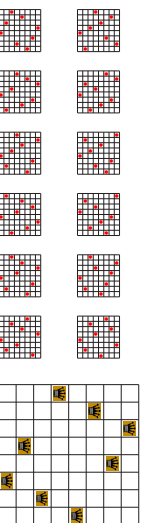
25

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

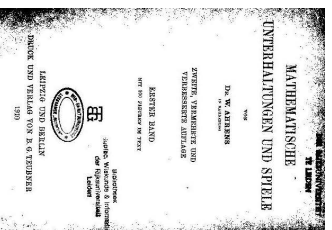


27

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.



29



Die W. Ahrens  
 MATHEMATISCHE  
 UNTERHALTUNGEN UND SPIELE  
 VON  
 DR. W. AHRENS  
 ZWEITE, VERBESSERTE AUF-  
 LAGE  
 BIRKÄUSER VERLAG  
 BASEL UND BASELSTADT  
 1910

Die W. Ahrens  
 MATHEMATISCHE  
 UNTERHALTUNGEN UND SPIELE  
 VON  
 DR. W. AHRENS  
 ZWEITE, VERBESSERTE AUF-  
 LAGE  
 BIRKÄUSER VERLAG  
 BASEL UND BASELSTADT  
 1910

31

Het **acht koninginnenprobleem** luidt als volgt:

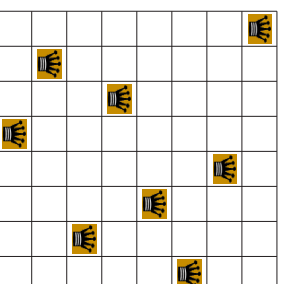
1. Kun je 8 dames (koninginnen) op een  $8 \times 8$  schaakbord zetten zonder dat zij elkaar aanvallen (= in één keer kunnen slaan)?
2. Zo ja, op hoeveel verschillende manieren kan dat?

En nu **algemeen**:

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

26

Een oplossing is onderstaande configuratie:



dit correspondeert met  
 de volgende permutatie:  
 1 5 8 6 3 7 2 4

28

$n$	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	40320
9	352	46	362880
10	724	92	3628800
11	2680	341	39916800
12	14200	1787	479001600
13	73.712	9233	
14	365.596		

30

$n$	doppelt- symme- trische	Stammlosungen			Gesamt- zahl aller Lösungen
		einfa- ch- symme- trische	un- symme- trische	mit sammen	
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

32

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

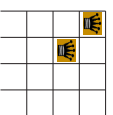
-  $n!$  onder de aanname: één dame per rij en één per kolom; dit zijn gewoon alle permutaties van  $1$  t/m  $n$

33

**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!

35

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!**

65

```
void zetdames (int n, int rij, int stand[], int & aantal) {
    // probeert de rij-de dame meer 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 (geenaval (rij, stand))
                zetdames (n, rij+1, stand, aantal);
        } // for
    } // zetdames
}
```

67

**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

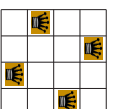
**Vergelijk**

- 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

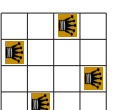
34

1	2	3	4
2			
3			
4			

oplossing 1



oplossing 2



64

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.

66

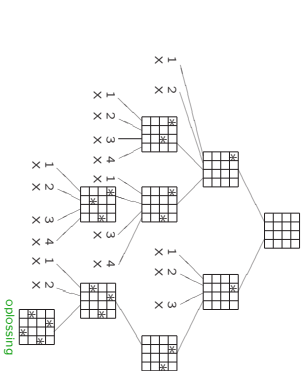
```
#include <iostream>
using namespace std;
const int MAX = 20;
bool geenaval (int rij, int stand[]) {
    while (veilig == true; int hulprij = 1;
           geenaval (rij, stand)) && (stand[rij]-stand[hulprij] !=
           -rij-hulprij) && (stand[rij]-stand[hulprij] != hulprij-rij));
    hulprij++;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

68

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) && (!geenaaval (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
```

69



x: deeloplossing niet verder uitbreiden, keuze herzien

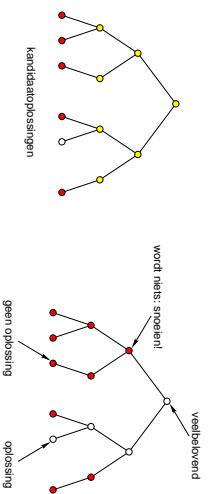
70

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

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

Exhaustive search (niet 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).

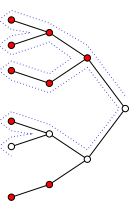
71



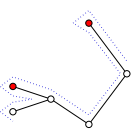
**Exhaustive search:** de hele boom wordt bekeken (tot een goede oplossing is gevonden)

**Backtracking:** hele subbomen kunnen soms worden **ge-snoeid**

72



Backtracking doorzocht de state-space tree via **depth first search**.



Als het meezit wordt er flink gesnoeid.

73

De volledige toestandruimte (state space) is **exponentieel**:

- 1 beginttoestand (leeg bord)
- $n!$  eindtoestanden ( $n$  dames geplaatst) (\*)
- $n + n * (n - 1) + n * (n - 1) * (n - 2) + \dots + n * (n - 1) * \dots * 2$  tussen-gelegen toestanden (de eerste 'zoveel' dames geplaatst) (\*)

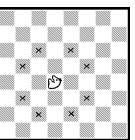
Backtracking zal voor grote probleeminstanties toch exponentieel zijn.

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

74

**Vraag:**

Hoewel verschillende series van  $n * n - 1$  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	14	8	5
16	11	12	3	4	21	18
13	2	15	18	9	4	7

een oplossing op het 3 bij 7 bord

75

- **Lezen/leren bij dit college:** Paragraaf 3.4, 3.5, sheets, 12 inl., 12.1
- **Werkcollege** brute force en backtracking donderdag 7 maart 2013, 13:45–15:30, in zaal Plein
- **Opgaven:** zie <http://www.liaos.nl/home/rvliet/algoritmiek/>
- **Volgend college:** donderdag 14 maart 2013

76