

# Informatica

## Les 6

### Basis- & slimme algoritmen

Jan Lemeire

**Informatica 2<sup>e</sup> semester**

*februari – mei 2021*



Vrije Universiteit Brussel

# **Slimme algoritmen**

## **Deel I**

### **Hoofdstuk 5**

# Beslissingsalgoritme

- ◆ Doel bereiken
- ◆ Aantal acties ter beschikking
- ➔ Actiesequentie bedenken om doel te bereiken

# Indeling volgens oplossingsmethode

- ◆ **Type 1:** De oplossing kan berekend worden met een formule (analytisch).
- ◆ **Type 2:** Je kunt de oplossing gericht zoeken of construeren (rechttoe-rechtaan).
- ◆ **Type 3:** Je gaat alle mogelijke actiesequenties af om een oplossing te vinden.
- ◆ **Type 4:** Door slimme keuzes (*heuristieken*) te maken, kan je verschillende actiesequenties uitsluiten.
- ◆ **Type 5:** Je leert al doende welke de juiste keuzes zijn.

# Type 1: analytisch

## 1. Je kan oplossing berekenen met formule

- *Analytisch*
- *Voorbeeld:* nulpunten van kwadratische vergelijking

# Type 2: gerichte 'constructie'

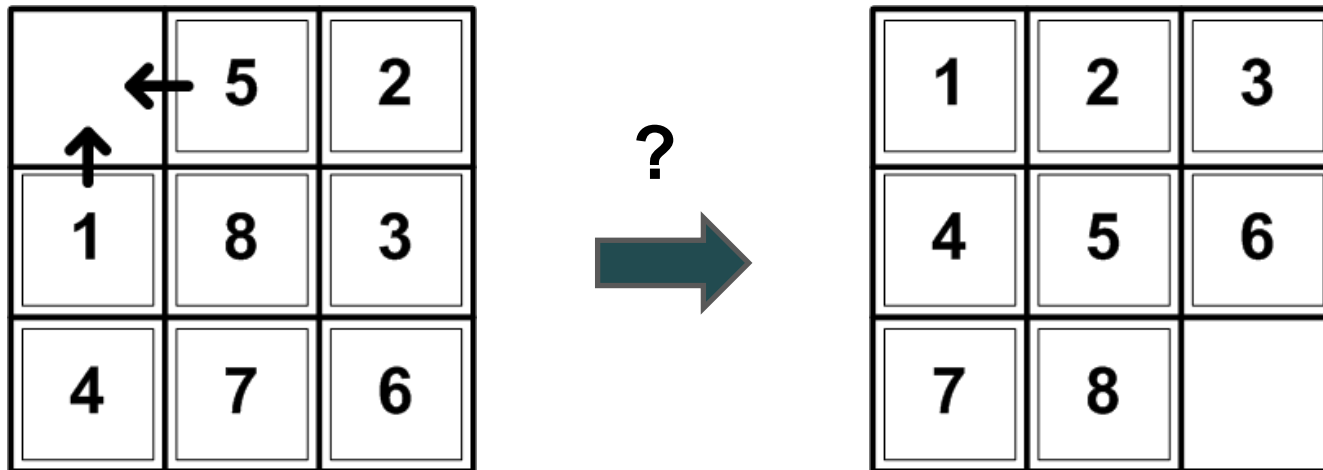
Je kan oplossing gericht zoeken met quasi-zekerheid om te arriveren

## *Voorbeelden:*

- ✦ Nulpunt van een functie zoeken met benaderingsalgoritme
- ✦ priemgetallen met zeef van Erathosteness
- ✦ Berekening grootst-gemene deler
- ✦ Dijkstra

# De schuifpuzzel

Puzzel 1 in schuifpuzzel.jar



- ◆ Staat van puzzel: posities van stukjes
- ◆ Mogelijke acties: LEFT, UP, RIGHT, DOWN
  - ✦ Niet alle 4 steeds mogelijk

◆ **We gaan 6 algoritmen zien**

# Java app

We hebben een app gemaakt waarmee je de 6 algoritmen kan vergelijken en zelf ook het spel spelen. Te vinden als **SchuifPuzzel.jar** op parallel website (Theorie – Les 7) of als **Schuifpuzzel.java** in package **Schuifpuzzel**.

Deze code verschilt van die uit **package zoeken** om het spel interactief te maken.

We tellen het aantal **bekeken** nodes (niet bezochte nodes) als maat voor de zoektijd.

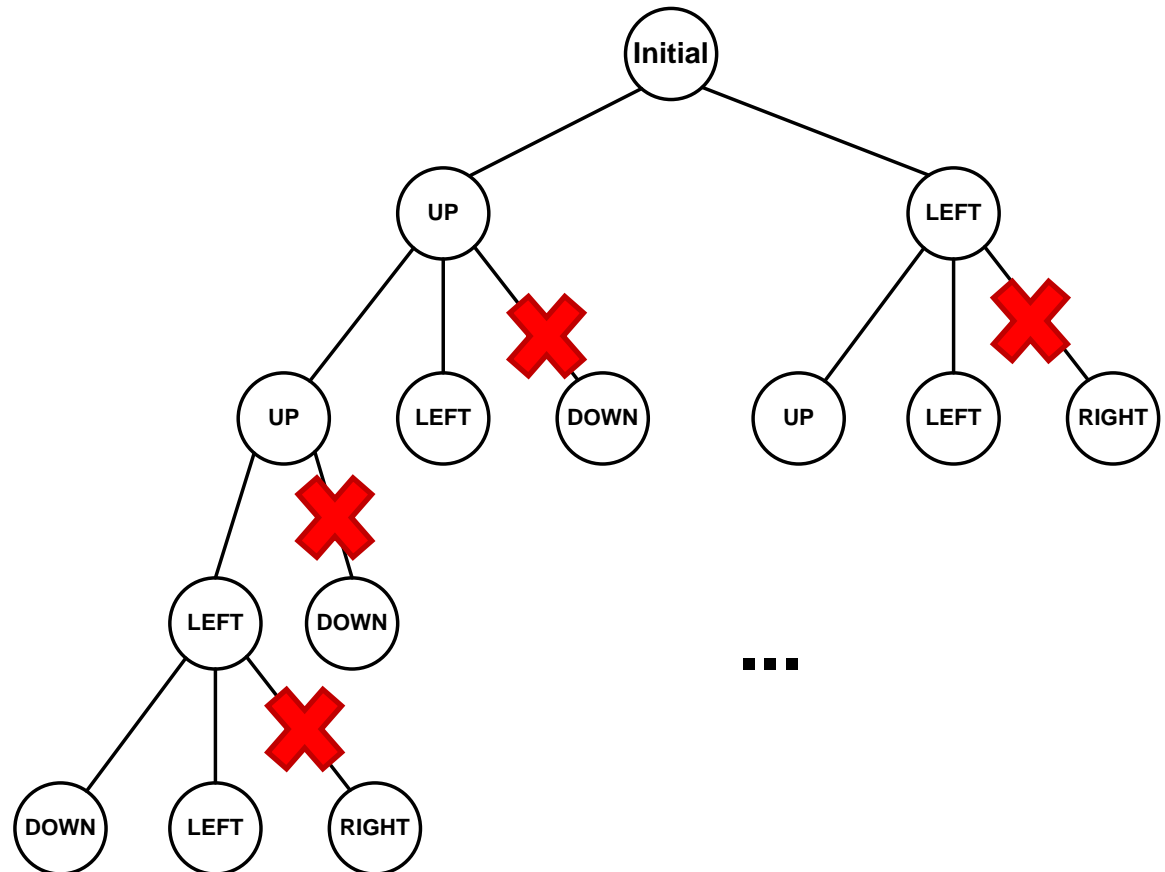


# Type 3: Alle mogelijkheden afgaan

- ◆ Je weet niet in welke richting de oplossing ligt.
  - ✦ Voor sommige problemen is men er vrijwel zeker van dat je alle mogelijkheden af moet lopen (*NP-complete problemen, zie later*)
  - ✦ Je genereert alle mogelijke sequenties van akties en kijkt welke tot *een* oplossing of tot *de beste* oplossing leidt.
- ◆ **het scannen van de volledige zoekruimte is nodig...**

# Zoekruimte = zoekboom

	← 5	2
↑ 1	8	3
4	7	6



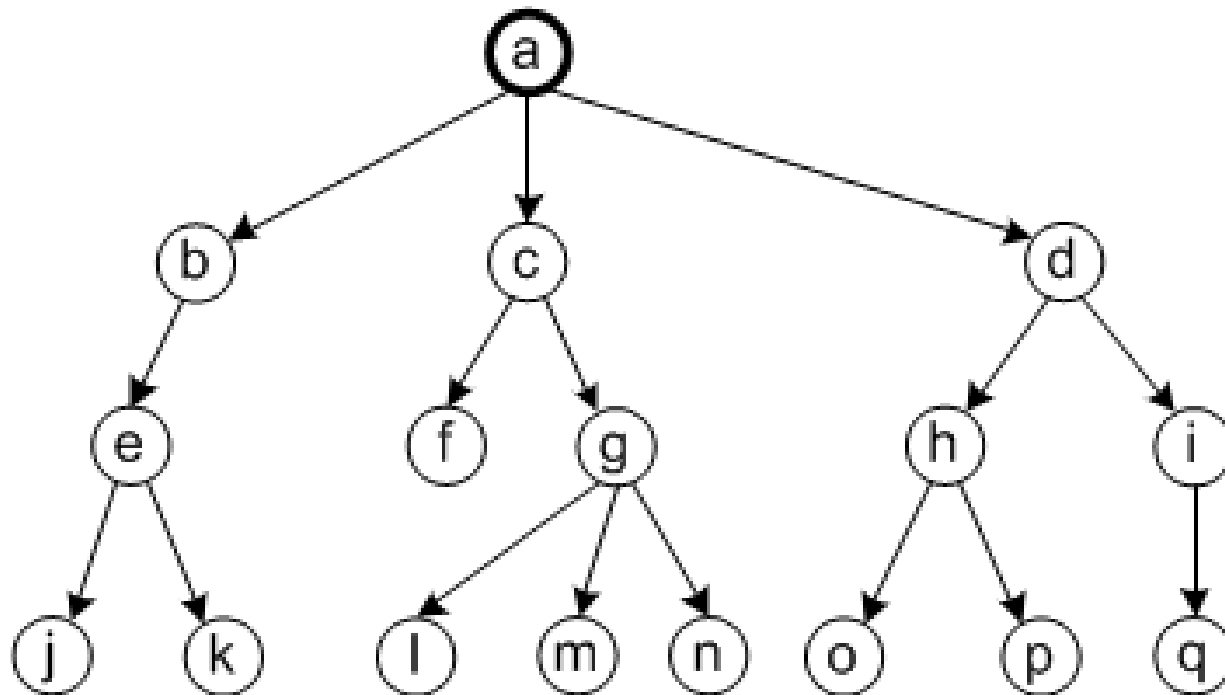
# Doorploeteren zoekboom

= **Brute-force search**

◆ 2 mogelijkheden:

✦ depth-first : a-b-e-j-k-c-f-g-l-m-n-d-h-o-p-i-q  
= *backtracking*

✦ breadth-first: a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q



# Sudoku met backtracking

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

◆ <http://en.wikipedia.org/wiki/Backtracking>

# (1) Backtracking

```
public interface ZoekNode<Zet> {  
    List<Zet> possibleMoves();  
    void move(Zet zet);  
    void undoMove(Zet zet);  
    boolean isSolution();  
    ZoekNode<Zet> clone();  
}
```

```
/** backtracking  
 * toepassing: sudoku  
 */  
public static <Zet> boolean backtracking (ZoekNode<Zet> node){  
    List<Zet> zetten = node.possibleMoves();  
  
    for(Zet zet: zetten ){  
        node.move(zet);  
        if (node.isSolution()){  
            System.out.println("Oplossing gevonden: "+node+"!");  
            return true;  
        }  
        if (backtracking(node))  
            return true; // stop met verder zoeken  
  
        node.undoMove(zet); // keer terug  
    }  
    return false; // geen children of geen enkele leidde naar oplossing  
}
```

# Generieke implementatie

- ◆ Via abstractie is het algoritme *algemeen bruikbaar*, voor elk probleem!

- ✦ Enkel interface implementeren

- ◆ Code + uitleg staat op website

- ✦ Je moet enkel de code van de algoritmen begrijpen, niet de implementatie van ZoekNode

```
public interface ZoekNode<Zet> {  
    List<Zet> possibleMoves();  
    void move(Zet zet);  
    void undoMove(Zet zet);  
    boolean isSolution();  
    ZoekNode<Zet> clone();  
}
```

# Backtracking met gelimiteerde diepte

```
static int MAXIMALE_DIEPTE = 10;
```

```
public static <Zet> List<Zet> backtrackingMetPadEnMaximaleDiepte  
(ZoekNode<Zet> node, int max_diepte)  
{  
    MAXIMALE_DIEPTE = max_diepte;  
    return backtrackingMetPadEnMaximaleDiepte_rec(node, 1);  
}
```

```

private static <Zet> List<Zet> backtrackingMetPadEnMaximaleDiepte_rec
(ZoekNode<Zet> node, int diepte){
    List<Zet> zetten = node.possibleMoves();

    for(Zet zet: zetten ){
        node.move(zet);
        if (node.isSolution()){
            // pad wordt aangemaakt bij het terugkeren uit de recursie
            List<Zet> pad = new ArrayList<Zet>();
            pad.add(zet);
            return pad;
        }
        if (diepte < MAXIMALE DIEPTE){
            List<Zet> pad = backtrackingMetPadEnMaximaleDiepte_rec(node,
diepte+1);
            if (pad != null){
                pad.add(0, zet); // voeg vooraan toe
                return pad; // stop met verder zoeken
            }
        }
        node.undoMove(zet); // keer terug
    }
    return null; // geen children of geen enkele leidde naar oplossing
}

```



```

private static <Zet> List<Zet> backtrackingsMetPadEnMaximaleDiepte_rec
(ZoekNode<Zet> node, int diepte){
    List<Zet> zetten = node.possibleMoves();

    for(Zet zet: zetten ){
        node.move(zet);
        if (node.isSolution()){
            // pad wordt aangemaakt bij het terugkeren uit de recursie
            List<Zet> pad = new ArrayList<Zet>();
            pad.add(zet);
            return pad;
        }
        if (diepte < MAXIMALE_DIEPTE){
            List<Zet> pad = backtrackingsMetPadEnMaximaleDiepte_rec(node,
diepte+1);
            if (pad != null){
                pad.add(0, zet); // voeg vooraan toe
                return pad; // stop met verder zoeken
            }
        }
        node.undoMove(zet); // keer terug
    }
    return null; // geen children of geen enkele leidde naar oplossing
}

```

## Code breadth-first (2)

```
public static <Zet> boolean breadthfirst (ZoekNode<Zet> startnode){
    FIFOQueue<ZoekNode<Zet>> openNodes = new FIFOQueue<ZoekNode<Zet>>(1000);
    openNodes.add(startnode);

    while(!openNodes.isEmpty()){
        ZoekNode<Zet> node = openNodes.get();

        List<Zet> zetten = node.possibleMoves();
        for(Zet zet: zetten ){
            ZoekNode<Zet> child = node.clone(); //copy!!
            child.move(zet);
            if (child.isSolution()){
                System.out.println("Oplossing gevonden!");
                return true;
            }
            openNodes.add(child);
        }
    }
    return false; // geen oplossing gevonden
}
```

# Conclusies

## ◆ (1) Backtracking:

- ✦ zoekboom wordt niet in het geheugen gehouden
- ✦ De recursie houdt het pad bij
  - Pad construeer je bij het terugkeren van de recursie
- ✦ Niet steeds snelste/kortste oplossing

## ◆ (2) Breadth-first

- ✦ Je maakt boom aan in het geheugen
  - Steeds kopieën van de nodes nodig
  - Child moet zijn parent bijhouden (is nu niet in code)
  - En eventueel een parent al zijn children
- ✦ Pad haal je uit de boom

# 'Domme rekenkracht'

- ◆ Brute force: puur rekenen
- ◆ Niet echt intelligent
- ◆ Echter: zoekruimte wordt snel te groot!
  - ✦ 10 stappen  $> 2^{10} \approx 1000$  mogelijke sequenties
  - ✦ 6x6 puzzel  $> 100$  stappen  $> 2^{100} \approx 10^{30}$  mogelijke sequenties
- ◆ Kunnen we gerichter zoeken?

# Oplossingsmethoden

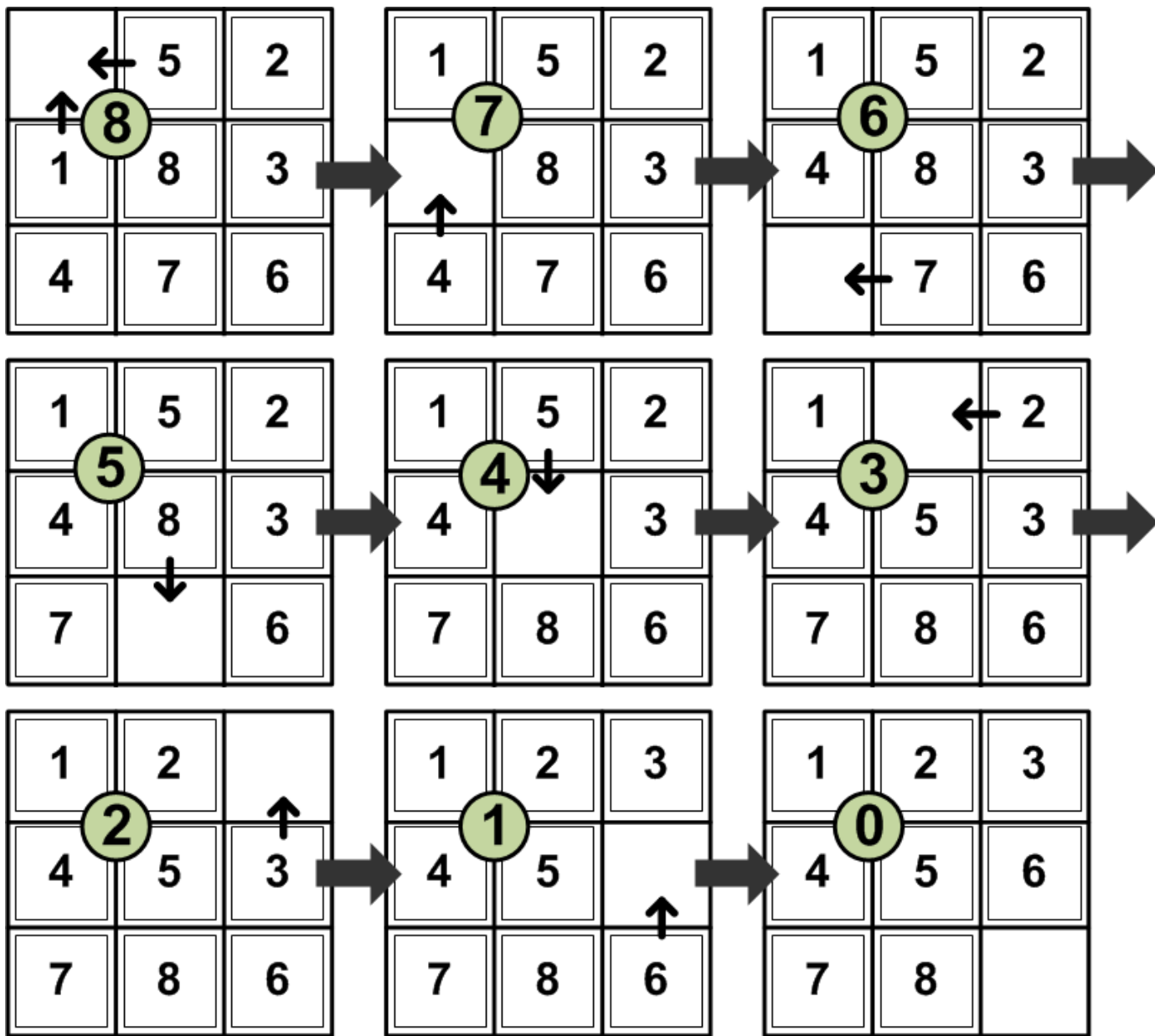
- ◆ **Type 1:** De oplossing kan berekend worden met een formule (analytisch).
- ◆ **Type 2:** Je kunt de oplossing gericht zoeken of construeren (rechttoe-rechtaan).
- ◆ **Type 3:** Je gaat alle mogelijke actiesequenties af om een oplossing te vinden.
- ◆ **Type 4:** Door slimme keuzes (*heuristieken*) te maken, kan je verschillende actiesequenties uitsluiten.
- ◆ **Type 5:** Je leert al doende welke de juiste keuzes zijn.

# Gericht zoeken: (3) *greedy search*

- ◆ Kies actie die oplossing korter bij brengt
- ◆ Gebaseerd op een **score**
- ◆ Puzzel: *totale afstand tot eindpositie*
  - ✦ Manhattan-afstand van stukje tot eindpositie

$$|x_1 - x_2| + |y_1 - y_2|$$

- ✦ Sommeren over alle stukjes => score



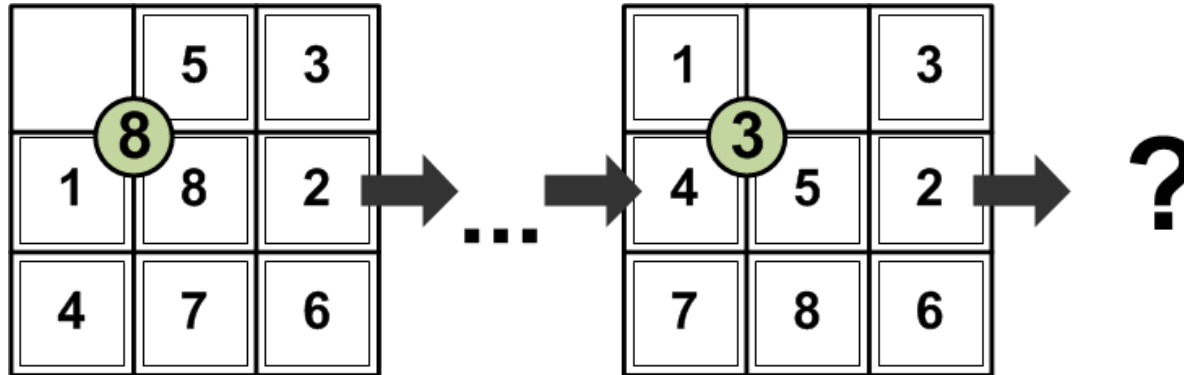
OK



stu...

Puzzel 9 in schuifpuzzel.jar is wel een goede puzzel

Puzzel 2 in schuifpuzzel.jar



**Opmerking:**

Met app kom je hierop uit, omdat je bij zet 5 twee mogelijkheden hebt

1	5	3
4	2	6
7	8	Score = 2

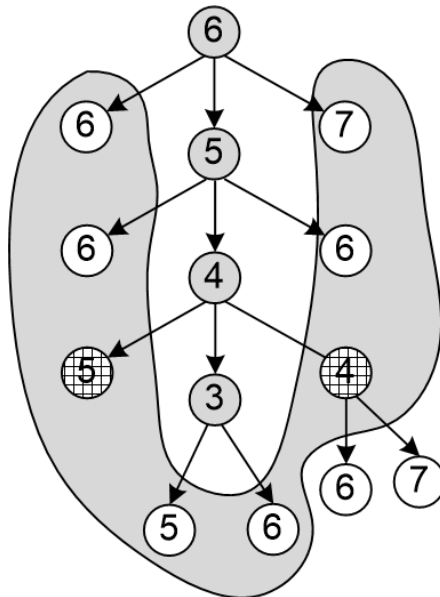
- ➡ terugkeren op stappen (algoritmen 4 & 5)
- ➡ of in de toekomst kijken noodzakelijk! Algoritme 6.



# (3) Greedy search

Zoekboom p.38 in cursus en schuifpuzzel.jar

Fictieve zoekboom, niet gelinkt aan een schuifpuzzel



4 **bezochte** noden (grijze nodes)

4+8 **bekeken** nodes (grijze nodes + grijze wolk)

## (4) Best-first search

- ◆ Idee student Ruben Pauwels (examen juni 2016): *greedy search* combineren met *backtracking*: als je vast zit, keer je terug op je stappen en neemt het tweede-beste pad.
- ◆ Best-first is gebaseerd op dit idee, maar gebruikt niet letterlijk backtracking, wel een queue. Veelgemaakte fout op examen!
- ◆ Dus: sorteren op score, in een lijst de open nodes bijhouden
  - ✦ **Open nodes** = tot waar je gekomen bent in de verschillende takken die je al bekeken hebt
  - ✦ **PriorityQueue** gebruiken: sorteert de elementen volgens score (ascending/opklimmend: laagste eerste)
- ◆ Implementatie zelfde als breadthfirst, maar dan **PriorityQueue** ipv **FIFOQueue**

## Code breadth-first => best-first

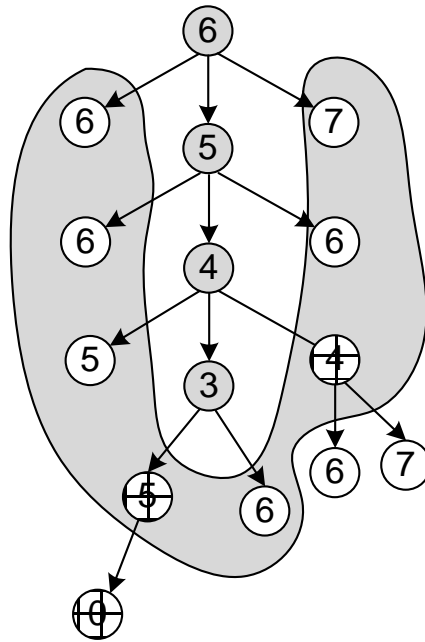
```
public static <Zet> boolean bestFirst(ZoekNode<Zet> startnode){
    PriorityQueue<ZoekNode<Zet>> openNodes =
        new PriorityQueue<ZoekNode<Zet>>(1000);
    openNodes.add(startnode);

    while(!openNodes.isEmpty()){
        ZoekNode<Zet> node = openNodes.poll();

        List<Zet> zetten = node.possibleMoves();
        for(Zet zet: zetten){
            ZoekNode<Zet> child = node.clone(); //copy!!
            child.move(zet);
            if (child.isSolution()){
                System.out.println("Oplossing gevonden!");
                return true;
            }
            openNodes.add(child);
        }
    }
    return false; // geen oplossing gevonden
}
```

# Voorbeeld best-first

## Zoekboom p.38 in cursus en schuifpuzzel.jar



*We gaan verder waar we gestopt zijn met greedy search*

Als meerdere nodes met *gelijke score*: we nemen de jongste, volgens het lifo-principe (last-in, first-out) zoals bij een stack.  
=> de gearceerde 5 wordt genomen.

# (5) A-star: voorbeeld



- ◆ Robot moet weg vinden (rood -> groen)
  - ✦ *g-score* = afstand vanaf start
  - ✦ *h-score* = afstand tot doel
  - ✦ **Laagste scores (*g-score* + *h-score*) worden eerst bekeken: hierop sorteren met PriorityQueue**
  - ✦ **Hetzelfde als best-first, enkel dat *g-score* wordt toegevoegd**
- ◆ [http://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm](http://en.wikipedia.org/wiki/A*_search_algorithm)

## Code breadth-first => aStar

```
public static <Zet> boolean aStar(ZoekNode<Zet> startnode){
    PriorityQueue<ZoekNode<Zet>> openNodes =
        new PriorityQueue<ZoekNode<Zet>>(1000);
    openNodes.add(startnode);

    while(!openNodes.isEmpty()){
        ZoekNode<Zet> node = openNodes.poll();

        List<Zet> zetten = node.possibleMoves();
        for(Zet zet: zetten){
            ZoekNode<Zet> child = node.clone(); //copy!!
            child.move(zet);
            if (child.isSolution()){
                System.out.println("Oplossing gevonden!");
                return true;
            }
            openNodes.add(child);
        }
    }
    return false; // geen oplossing gevonden
}
```

**Code identiek aan best-first!**  
**Enkel score is anders**



# (6) Iterative Deepening

**Niet in cursus**

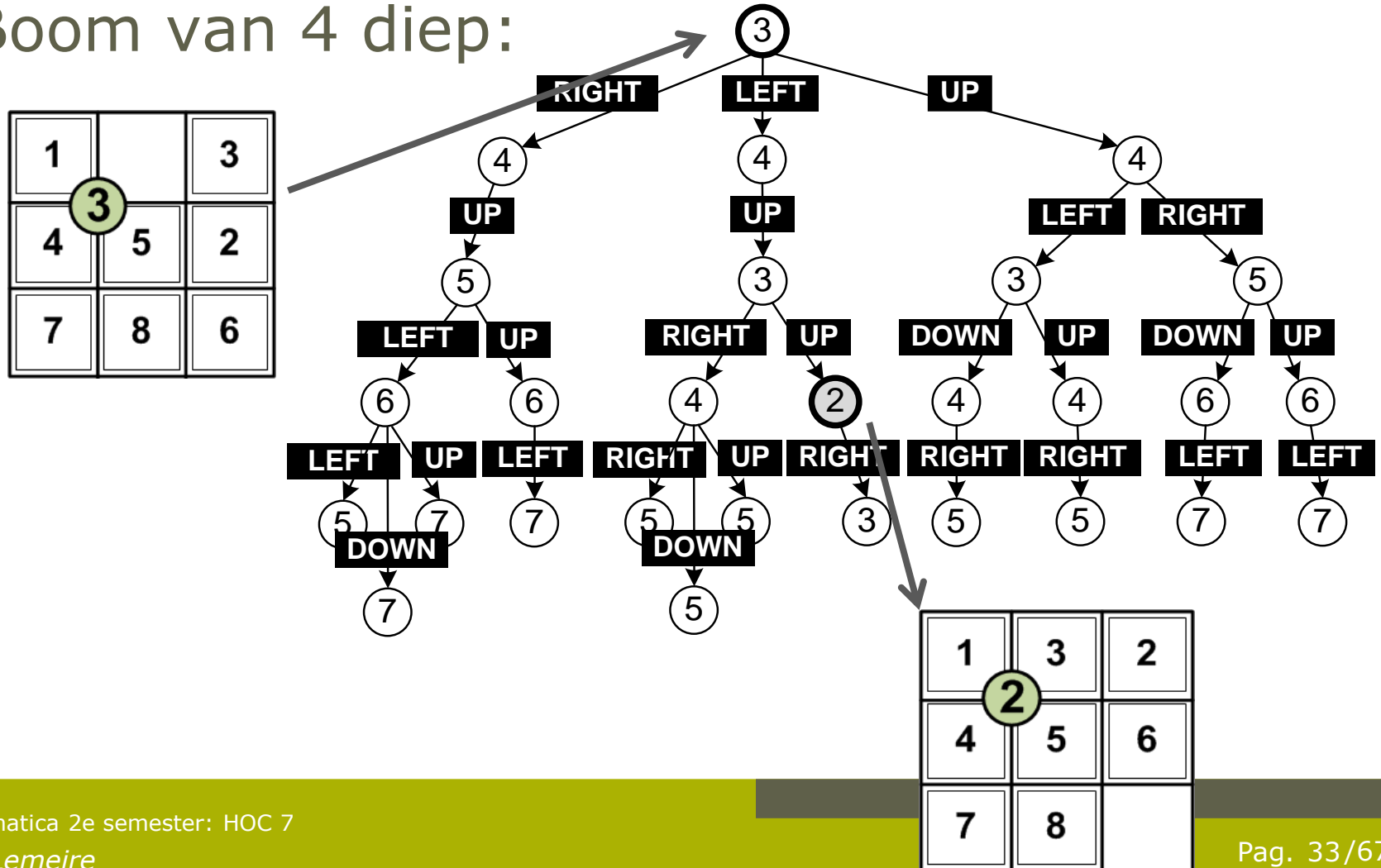
- ◆ Start met beginsituatie als beginnode
- ◆ Doe iteratief tot oplossing gevonden of maximale diepte bereikt
  - ✦ Vanaf de node de zoekboom uitwerken (breadth-first van links naar rechts) tot een zekere diepte (**horizon**)
  - ✦ Stoppen als een oplossing gevonden wordt
  - ✦ Selecteer hoogste score uit de bladeren (nodes op diepte van horizon)
    - Als verschillende nodes met dezelfde score, behouden we de eerste node die we tegenkwamen
  - ✦ Voer de zet uit die naar de hoogste score leidt
    - Deze node wordt de nieuwe beginnode voor de volgende iteratie

Je zet dus 1 stapje en werkt telkens opnieuw de zoekboom uit tot de horizon. Boom wordt dus telkens 1 level dieper uitgewerkt.



# (6) Iterative Deepening

◆ Boom van 4 diep:



# Iterative Deepening

## ♦ Boom uitwerken vanaf

Puzzel 2 in schuifpuzzel.jar  
Waar greedy search stopt

1	3	2
4	2	6
7	8	

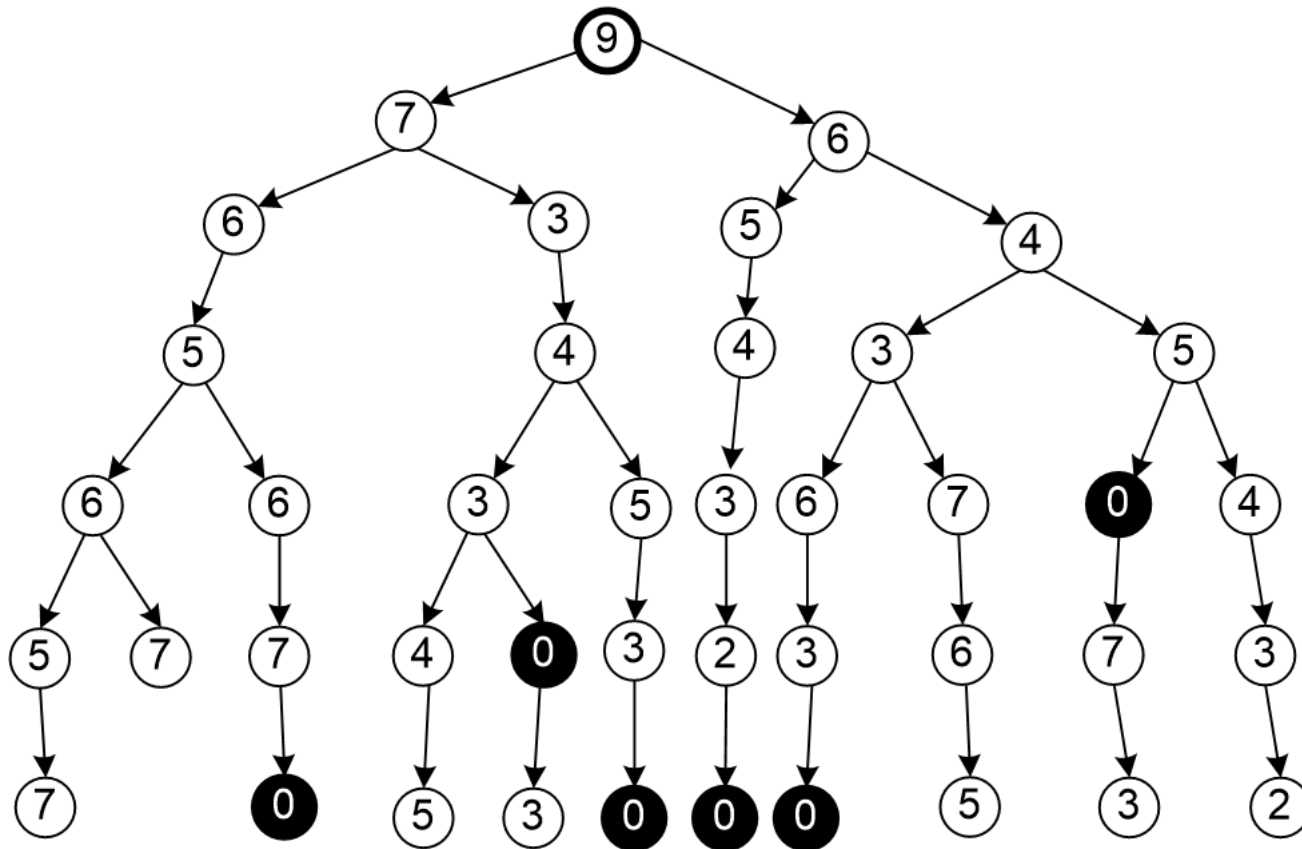
♦ Met diepte 4 komen we er niet...

♦ *In feite heeft deze puzzel helemaal geen oplossing, het verwisselen van '2' en '3' geeft een onmogelijke situatie!!*

Puzzel 9 in schuifpuzzel.jar is wel een correcte puzzel

1	3	6
4	Score = 6	2
7	5	8

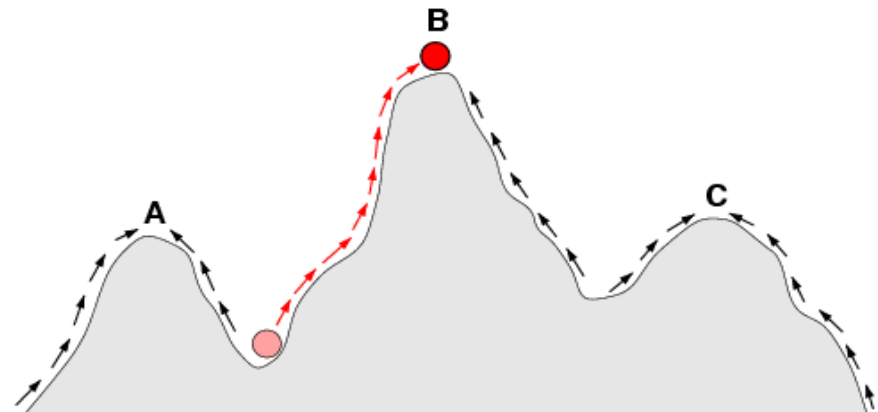
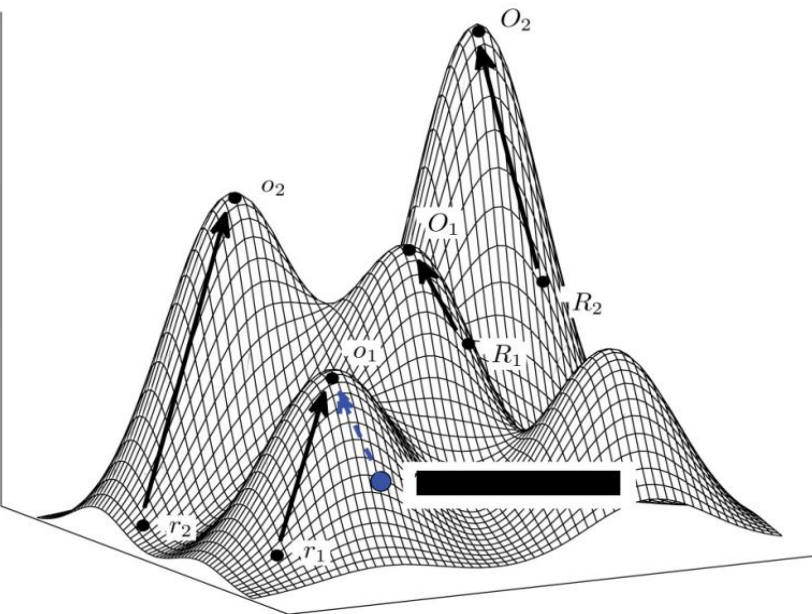
- ♦ Greedy search loopt vast
- ♦ Vergelijk iterative deepening met horizons 1, 2 en 3.
- ♦ Vergelijk met breadth-first en a-star



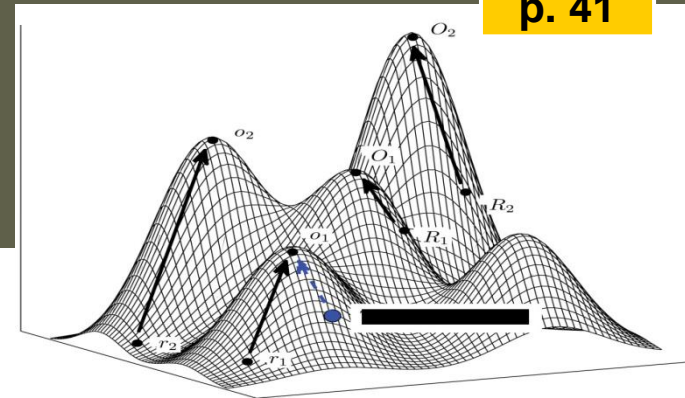
- Gegeven: zoekboom met scores (niet zeker dat een perfecte inschatting is).
- Doel: score 0
- Pas de 6 verschillende zoekalgoritmen toe (horizon=2, test ook horizon 3).
- Vergelijk de snelheid om een oplossing (zwarte node) te vinden.
- Vergelijk voor het vinden van de kortste oplossing (minst aantal stappen).
- Welke algoritmen garanderen de kortste oplossing?

# Score in zoekruimte

◆ **Probleem:** terechtkomen in **lokaal maximum**

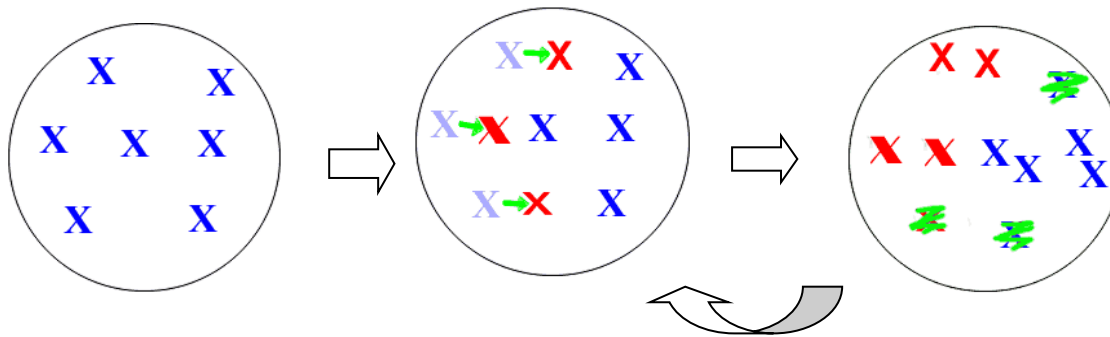


# Fitness landscape



- ◆ Biologische evolutie van soorten
  - ✦ *Fitheid*: bepaald door zijn aangepastheid aan de omgeving
  - ✦ *Natuurlijke selectie*
  - ✦ Verandering bouw en het gedrag door *genetische mutaties*
- ➔ Evolutie van soorten
  - ✦ Klimmen en sprongen door het fitness landscape
  - ✦ elke stabiele soort op een bergtopje zit
- ◆ Evolutie nabootsen: genetisch algoritme

# Genetisch algoritme



***Initële populatie***

***Sommige individuen muteren***

***Enkel de besten overleven***

# Verdere verbeteringen puzzelalgoritme

## a) puzzel rij-per-rij afwerken.

- Score enkel op eerste rij definiëren, vervolgens op 2<sup>e</sup> rij, ...
- Behalve de laatste twee rijen, die tegelijkertijd aanpakken
- Beperkte horizon OK

## b) Leren (type 5): zie later

## c) Regels opstellen

- Type 1 probleem

# Hoe generiek programmeren?

niet te kennen,  
optioneel – als vervangvraag



# Generieke implementatie

- ◆ Via abstractie is het algoritme algemeen bruikbaar
- ◆ Code + uitleg staat op website
  - ✦ Java package **'zoeken'**
  - ✦ Niet te kennen
  - ✦ Optioneel: als vervangvraag (in de plaats van een andere vraag)

```
interface Node {  
    boolean isOplossing();  
    List<Zet> volgendeZetten();  
    void doeZet(Zet zet);  
    void ontdoeZet(Zet zet);  
    Node clone();  
}  
interface Zet{  
  
}
```

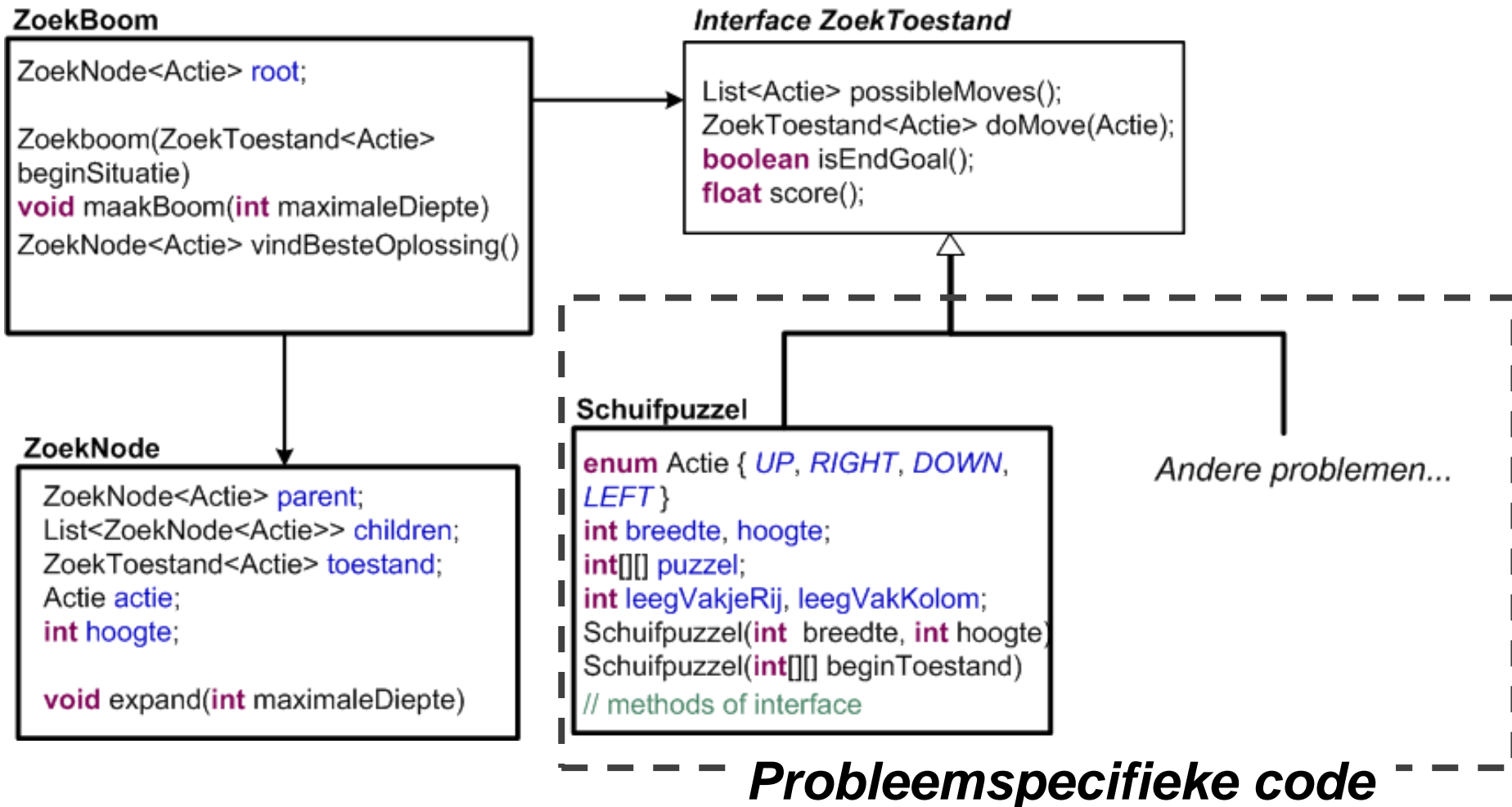
# Uitdaging: algemene oplossing

*onafhankelijk van het specifieke probleem dat je wenst op te lossen!*

◆ Wat moet je weten om de boom aan te maken?

Je moet de mogelijke *acties* kennen. Met de actie kom je in een nieuwe situatie, je bereikt een nieuwe toestand. Je wilt ook weten of je de gevraagde eindsituatie bereikt hebt.

➔ **Allemaal operaties op de toestand**



Link tussen algoritme en probleem gegeven door interface