Addendum bij hoofdstuk 5
Generieke implementatie van de zoekalgoritmen

De implementatie wordt kort besproken in 5.2.6

1. Programmatie Zoekalgoritme

Definitie van boom: we hebben geen binaire boom met exact twee kinderen, maar een boom waarin elke node een willekeurig aantal kinderen kan hebben. Hiervoor gebruiken we een lijst van nodes, *children*. In elke node houden we ook de laatste actie bij die leidde tot deze node, de toestand van de puzzel op dat moment (hier komen we dadelijk op terug) en de oudernode (*parent*), zodat we steeds weten van welke node we komen.

```
class ZoekNode<Actie>{
	List<ZoekNode<Actie>> children;
	ZoekNode<Actie> parent;
	ZoekToestand<Actie> toestand;
	Actie actie;
	int diepte;
}
```

We wensen ons algoritme algemeen maken, zodat ze kan worden gebruikt voor gelijkaardige problemen. Het zal een **abstract algoritme** worden dat *onafhankelijk is van het specifieke probleem dat je wenst op te lossen*! We gaan gebruik maken van de mogelijkheid om abstracte concepten te definiëren in object-georiënteerde programmeertalen.

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. Alles draait dus rond de toestand van het probleem. Merk op dat de mogelijke acties óók afhankelijk zijn van de toestand.

We programmeren dit met een *interface*:

```
public interface ZoekToestand<Actie> {

	List<Actie> possibleMoves();

	ZoekToestand<Actie> doMove(Actie actie);

	boolean isEndGoal();

	float score();
}
```

Deze interface vertelt alles wat je moet weten om het probleem op te lossen! De *uitwerking* van de methodes is afhankelijk van het probleem. We gaan dit doen in een implementatie van deze interface.

Dit toont de kracht van object-georiënteerd programmeren. Je kan oplossingen programmeren die heel algemeen zijn en werken voor alle gelijkaardige problemen. Het enige wat de gebruiker moet doen is de methodes van de interface implementeren en die implementatie meegeven aan het abstract algoritme.

Hier het begin van de implementatie van de schuifpuzzel:

```
public class Schuifpuzzel implements ZoekToestand<Schuifpuzzel.Actie>{

	public enum Actie { UP, RIGHT, DOWN, LEFT }

	int breedte, hoogte; // afmetingen van de puzzel
	int[][] puzzel; // het lege vakje krijgt de waarde 0
	int leegVakjeRij, leegVakKolom; // coordinaten van het lege vakje

	/**
	 * Maakt een schuifpuzzel aan met een willekeurige begintoestand
	 */
	Schuifpuzzel(int  breedte, int hoogte){
		this.breedte = breedte;
		this.hoogte = hoogte;
		puzzel = new int[breedte][hoogte];
		// TODO: opvullen met random waarden
	}
	Schuifpuzzel(int[][] beginToestand){
		this.hoogte = beginToestand.length;
		this.breedte = beginToestand[0].length;
		puzzel = beginToestand; // bvb new int[][]{ {0, 5, 2}, {1, 8, 3}, {4,
7, 6}};
		// TODO: leegVakjeRij, leegVakKolom bepalen
	}
	@Override
	public List<Actie> possibleMoves(){
		return null; // TODO Auto-generated method stub
	}
	@Override
	public Schuifpuzzel doMove(Actie actie){
		return null;  // TODO Auto-generated method stub
	}
	@Override
	public boolean isEndGoal() {
		return false;  // TODO Auto-generated method stub
	}
	@Override
	public float score() {
		return 0; // TODO Auto-generated method stub
	}
}
```

We definiëren met een **Java enumeratie** (*enum*) de vier mogelijke acties. Een variabele van het type Actie kan één van deze vier waarden aannemen.
We definiëren de toestand van de puzzel met een datastructuur. Hier kiezen we voor een matrix van integers (*puzzel*). We houden ook de plaats van het lege vakje bij (*leegVakjeRij*, *leegVakKolom*). Die informatie zit in feite ook al in de matrix (aangegeven door een waarde 0), maar het is handig om die informatie expliciet te hebben. Anders moeten we telkens de hele matrix doorlopen op zoek naar het lege vakje.
We hebben hier twee constructors voorzien: eentje waarmee we zelf een begintoestand meegeven, eentje waarmee een willekeurige begintoestand gekozen wordt. Tot slot komen de drie methodes van de interface. Deze methodes moeten nog geïmplementeerd worden.

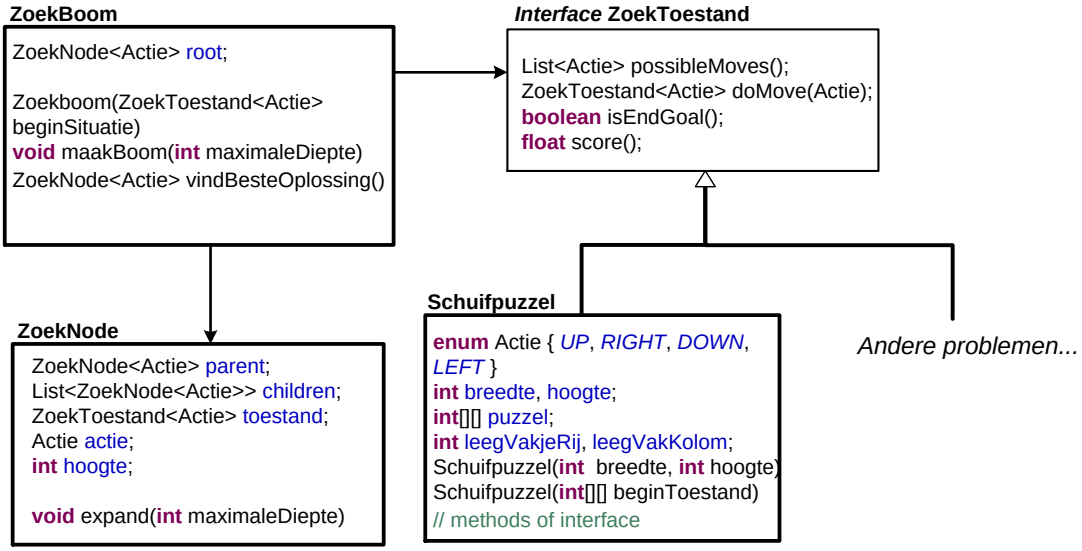
Het aanmaken van de zoekboom is nu redelijk eenvoudig. We definiëren een nieuwe klasse *Zoekboom*. Net als bij de *BinaryTree* moeten we enkel de *root* bijhouden. We definiëren ook een maximale diepte van de boom om de grootte van de boom te beperken. Het aanmaken van de boom gebeurt recursief: we voeren een *expand* uit van alle nodes. De *expand* vraagt de mogelijke acties op en genereert voor elke actie een nieuwe node. We kunnen de *expand* implementeren in de *ZoekBoom*-klasse, maar omdat deze operatie iets typisch is van een *ZoekNode* en daarom voegen we dit toe aan de *ZoekNode*-klasse. De opgegeven maximale diepte zorgt ervoor dat onze recursie stopt.

```
class ZoekNode<Actie>{
	// beschrijving van de boom
	ZoekNode<Actie> parent;
	List<ZoekNode<Actie>> children;
	// eigenschappen van node
	ZoekToestand<Actie> toestand;
	Actie actie;
	int diepte;

	// constructor en toString weggelaten

	void expand(int maximaleDiepte){
		if ( diepte < maximaleDiepte){
		   List<Actie> acties = toestand.possibleMoves();
		   for(Actie actie: acties){
		       ZoekToestand<Actie> nieuw = toestand.doMove(actie);
		       ZoekNode<Actie> childNode = new ZoekNode<Actie>(nieuw, actie,
diepte + 1);
		       children.add(childNode);
		       childNode.expand(maximaleDiepte); // recursieve oproep
		   }
		}
	}
```

```
public class Zoekboom<Actie> {

	ZoekNode<Actie> root;

	public Zoekboom(ZoekToestand<Actie> beginSituatie){
		root = new ZoekNode<Actie>(beginSituatie, null); // root heeft geen
actie
	}

	public void maakBoom(int maximaleDiepte){
		root.expand(maximaleDiepte);
	}
}
```

Merk op dat de code voor het aanmaken redelijk eenvoudig is! Dit komt omdat alle probleem-specifieke dingen afgehandeld worden binnen de specifieke *ZoekToestand*-implementatie. Bij het aanmaken van de boom moet je je er niets van aantrekken hoe de mogelijke acties berekend moeten worden of hoe de toestand er uit ziet. Dat is de bedoeling van object-georiënteerd programmeren: we verdelen de ‘verantwoordelijkheden’ over verschillende klassen. Elke klasse moet zich enkel bekommeren om zijn eigen dingen. Die klasse wordt eenvoudiger én ook algemener. Dit noemen we *generiek*: je oplossing kan gebruikt worden in alle gelijkaardige gevallen.

Dit uitgewerkt voorbeeld laat de structuur zien van een modern object-georiënteerd programma. Hier het schema die de relaties tussen de klasses toont. Merk op dat de relatie tussen *Zoekboom* en *Schuifpuzzel* gedefinieerd is door de *interface ZoekToestand*. Daarom heet het een *interface*: het bepaalt de connectie tussen twee dingen. Het is ook het enige wat de ene over de andere moet weten. Een *interface* kan je ook zien als het *venster* waardoor de ene klasse naar de andere kijkt.



Backtracking (depth-first)
Door de boom lopen op zoek naar de beste kunnen we als volgt doen (depth-first):

```
public ZoekNode<Actie> vindBesteOplossing(){
	return vindBesteOplossing(root, null);
}
private ZoekNode<Actie> vindBesteOplossing(ZoekNode<Actie> node, ZoekNode<Actie>
besteNodeTotNu){
	if (node.toestand.isEndGoal()){
		if (besteNodeTotNu == null || besteNodeTotNu.diepte > node.diepte)
			besteNodeTotNu = node; // betere gevonden
	}
	for(ZoekNode<Actie> child: node.children) // recursie
		besteNodeTotNu = vindBesteOplossing(child, besteNodeTotNu);
	return besteNodeTotNu;
}
```

Als we de node met de beste oplossing gevonden hebben, kunnen we de actiesequentie afleiden die ons brengt van de begintoestand tot de oplossing.

```
/** Geeft de actiesequentie die ons brengt van de begintoestand tot de gegeven
node
 */
public List<Actie> actieSequentie(ZoekNode<Actie> node){
	ArrayList<Actie> actiesequentie = new ArrayList<Actie>();
	while (node.parent != null){
		actiesequentie.add(0, node.actie); // voeg toe aan de voorkant
		node = node.parent;
	}
	return actiesequentie;
```

```
}
```

Het aanmaken van de objecten en opstellen van de boom gebeurt als volgt:

```
/** PROGRAMMA */
public static void main(String[] args) {

  int[][] beginToestand = new int[][]{ {0, 5, 2}, {1, 8, 3}, {4, 7, 6}};
  Schuifpuzzel puzzel = new Schuifpuzzel(beginToestand);
  Zoekboom<Schuifpuzzel.Actie> zoekboom =
                          new Zoekboom<Schuifpuzzel.Actie>(puzzel);

  zoekboom.maakBoom(10);
  ZoekNode<Schuifpuzzel.Actie> oplossing = zoekboom.vindBesteOplossing();
  List<Actie> actieSequentie = zoekboom.actieSequentie(oplossing);
  System.out.println(actieSequentie);
}
```

2. Vergelijken van algoritmes

We hebben verschillende oplossingen bedacht voor de schuifpuzzel. Stel dat we deze algoritmes willen vergelijken. Dan kunnen we een lijst van algoritmen maken, deze één-voor-één vergelijken en onze conclusies trekken. Maar het zou goed zijn als we in de toekomst gemakkelijk nieuwe versies kunnen toevoegen. We willen dus een 'framework' voor zoekalgoritmes die een probleem oplossen. Dit kan met behulp van de abstractie-eigenschappen van Java.

Wat willen we doen met een zoekalgoritme?

- We moeten het algoritme kunnen starten met een gegeven beginsituatie.
- Eenmaal het algoritme gestopt, willen we weten of aldanniet de gevraagde eindsituatie bereikt is, de actiesequentie van de oplossing, het aantal acties (de lengte van de sequentie) en eventueel de score. In dit laatste geval kunnen we ook algoritmen bestuderen die niet de gewenste eindconfiguratie bereiken, maar die in de buurt stranden. We vragen daarom ook de bereikte eindtoestand op (die dus niet perfect kan zijn).

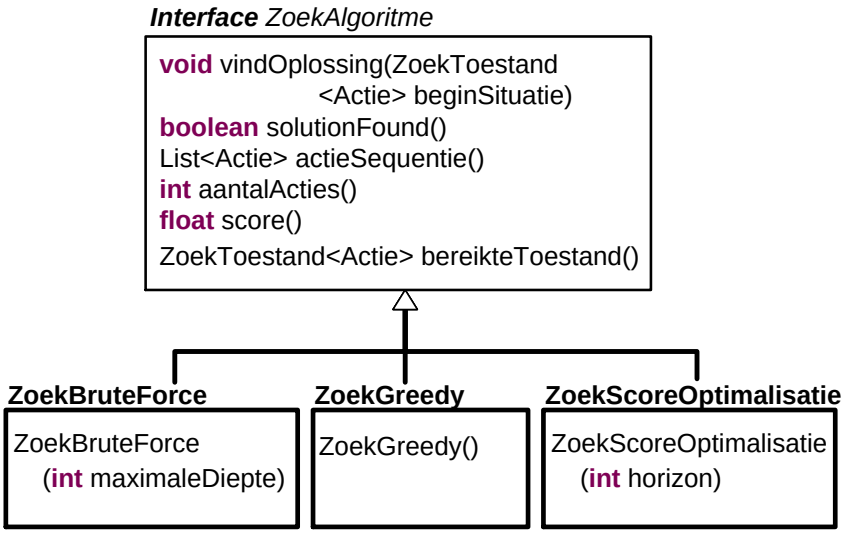
We definiëren dit op een abstracte manier met een *interface*. Merk op dat we meerdere resultaten wensen, maar een methode kan maar één ding teruggeven. We lossen het hier op door één methode per teruggeefwaarde te definiëren.

```
public interface ZoekAlgoritme<Actie> {

      void vindOplossing(ZoekToestand<Actie> beginSituatie);

      boolean solutionFound();

      List<Actie> actieSequentie();

      int aantalActies();

      float score(); // score van de bereikte toestand

      ZoekToestand<Actie> bereikteToestand();
 }
```

Elk algoritme kunnen we nu implementeren. Hier de implementatie van de *brute force search*:

```
public class ZoekBruteForce<Actie> implements ZoekAlgoritme<Actie> {
	int maximaleDiepte;
	List<Actie> actieSequentie;
	ZoekToestand<Actie> eindToestand;

	public ZoekBruteForce(int maximaleDiepte){
		this.maximaleDiepte = maximaleDiepte;
	}
	@Override
	public void vindOplossing(ZoekToestand<Actie> beginSituatie) {
		Zoekboom<Actie> zoekboom = new Zoekboom<Actie>(beginSituatie);
		zoekboom.maakBoom(maximaleDiepte);

		ZoekNode<Actie> besteOplossing = zoekboom.vindBesteOplossing();
		actieSequentie = zoekboom.actieSequentie(besteOplossing);

		eindToestand = besteOplossing.toestand;
	}
	@Override
	public boolean solutionFound() {
		return score() == 0;
	}
	@Override
	public List<Actie> actieSequentie() {
		return actieSequentie;
	}
	@Override
	public int aantalActies() {
		return actieSequentie.size();
	}
	@Override
	public float score() {
		return eindToestand.score();
	}
	@Override
	public ZoekToestand<Actie> bereikteToestand() {
		return eindToestand;
	}
	@Override
	public String toString() {
		return "bruteforce (max "+maximaleDiepte+")";
	}
 }
```

We implementeren ook de *toString()*-methode. Dit is handig omdat we hiermee de naam van het algoritme kunnen printen.
Deze klasse doet in feite niet echt iets. Het ‘kapselt’ het zoeken in een object die de gevraagde *interface* implementeert. Het eigenlijke zoeken gebeurt in *Zoekboom*. Merk op dat de andere algoritmes die we bespraken ook gebruik maken van een zoekboom. De nodige methodes voor deze algoritmes stoppen we dus ook best in de *Zoekboom*-klasse. Daarom is het ook goed dat we voor elk nieuw algoritme een aparte subklasse van *ZoekAlgoritme* definiëren, zoals getoond in het volgende schema:



De code gebruiken we nu als volgt:

```
public static void main(String[] args) {

   // definitie van het probleem
   int[][] beginToestand = new int[][]{ {0, 5, 2}, {1, 8, 3}, {4, 7, 6}};
   Schuifpuzzel puzzel = new Schuifpuzzel(beginToestand);

   // lijst alle beschikbare algoritmes op
   List<ZoekAlgoritme<Schuifpuzzel.Actie>> algoritmes =
       new ArrayList<ZoekAlgoritme<Schuifpuzzel.Actie>>();
   algoritmes.add( new ZoekBruteForce<Schuifpuzzel.Actie>(10) ); // 10 diep
   algoritmes.add( new ZoekBruteForce<Schuifpuzzel.Actie>(15) ); // 15 diep
   // andere algoritmes kunnen hier toegevoegd worden...

   // laat alle algoritmes hun werk doen
   for(ZoekAlgoritme<Schuifpuzzel.Actie> alg: algoritmes){
       System.out.print("Algoritme "+alg+": ");
       alg.vindOplossing(puzzel);
       System.out.println(" vindt oplossing met score "+alg.score()+" te bereiken
in "+alg.aantalActies()+" stappen.");
   }
}
```

We definiëren het probleem (*puzzel*) en maken een lijst van algoritmes. Ik heb hier tweemaal *ZoekBruteForce* toegevoegd, maar met een verschillende diepte. Al deze algoritmes vragen we het probleem op te lossen. Dan printen we de score en het aantal acties van de gevonden oplossing. Deze code is dus redelijk simpel en op een ‘abstract’ niveau: de details zitten verstopt in de objecten. Zo krijgen we de klasse *ZoekBoom* helemaal niet te zien! In het laatste gedeelte werken we enkel met *ZoekAlgoritme* -objecten.

We hebben twee niveaus van abstractie ingevoerd: we kunnen nieuwe problemen toevoegen via een implementatie van *ZoekToestand* en we kunnen nieuwe algoritmen toevoegen via een implementatie van *ZoekAlgoritme*. De rest van de code moeten we niet aanpassen. Je moet enkel in de *main* van de laatste code het probleem instantiëren en het zoekalgoritme toevoegen.