

1 Schuifpuzzels

De klassieke schuifpuzzel bestaat uit N bij N vakjes met daarin puzzelstukjes genummerd van 1 tot $N^2 - 1$. Het laatste stukje ontbreekt waardoor een gat in de puzzel ontstaat. Dit maakt het mogelijk om puzzelstukjes aangrenzend aan het gat te verschuiven. Doel van een schuifpuzzel is, door het verschuiven van één puzzelstukje per keer, een willekeurige begintoestand om te zetten in de eindconfiguratie, die hier voor een 4 bij 4 puzzel is weergegeven.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Per toestand zijn er hoogstens vier verschuivingen mogelijk. We kunnen deze aangeven met de richting waarin het gat zich beweegt: *Noord*, *Oost*, *Zuid* en *West*. Zoals je ziet zijn er in de getekende toestand slechts 2 verschuivingen mogelijk: het gat kan alleen richting *Noord* of *West* verplaatst worden.

2 Leerdoelen

In deze opgave vragen we je om een algoritme te implementeren dat een (liefst kortste) oplossing van een N bij N schuifpuzzel bepaalt. Het zoeken van een oplossing in de zoekruimte van alle mogelijke schuifpuzzels moet gedaan worden volgens de *breadth-first* zoekstrategie (BFS). Bij de cursus Inleiding AI zijn fundamentele zoekmethoden al aan de orde gekomen. Kortweg komt breadth-first zoeken erop neer dat je vanuit een bepaalde situatie eerst alle *directe* opvolgers onderzoekt alvorens je vanuit iedere opvolger verder gaat zoeken.

Na afloop van deze opdracht ben je in staat om:

- lijsten en queues te gebruiken om zoekalgoritmes te implementeren;
- met behulp van de breadth-first strategie een zoekruimte te doorlopen;
- eenvoudige generieke klassen te kunnen definiëren;
- de hash-methode te herdefiniëren en te gebruiken;
- de *best-first* optimalisatie van breadth-first search te gebruiken;
- priority queues uit de Java API te gebruiken.

3 Instructie

Bestudeer allereerst de onderdelen uit hoofdstuk 16 en 20 van het boek die tijdens het college zijn besproken en de dia's over dit onderwerp. Maak eerst op papier een ontwerp van je breadth-first search implementatie voordat je de code invoert en test. Het is verstandig om tijdens het ontwikkelen en testen van je code met kleine voorbeelden te werken. Begin bijvoorbeeld met bijna correcte oplossingen op een 2×2 of 3×3 bord.

4 Probleemschets

Voor het implementeren van een breadth-first zoekstrategie heb je een *queue* (of wachtrij) nodig. De essentie van deze datastructuur is dat de elementen die het langst in de queue staan het eerste aan de beurt komen, net zoals bij de wachtrij bij de kassa in de supermarkt. Nieuwe kandidaten moeten achteraan de rij worden toegevoegd en op hun beurt wachten. Voor het oplossen van de schuifpuzzel ga je globaal als volgt te werk: Zolang de queue van puzzels niet leeg is neem je het voorste element. Als deze puzzel overeenkomt met de eindconfiguratie dan ben je klaar. Anders ga je alle directe opvolgers van deze puzzel onderzoeken. Deze opvolgers zijn zelf natuurlijk ook weer puzzels en worden achteraan in de wachtrij gezet. Ze komen dan automatisch aan de beurt zodra alle puzzels die nu nog in de wachtrij staan zijn bekeken. Dit algoritme werkt niet alleen voor klassieke schuifpuzzels, maar voor ook voor tal van andere spelletjes, waarin zoeken een rol speelt. Om deze methode op abstracte wijze te beschrijven is het voldoende om te veronderstellen dat je concrete puzzel het volgende, eenvoudige interface implementeert:

```
public interface Configuration extends Comparable<Configuration> {  
  
    public Collection<Configuration> successors();  
  
    public boolean isSolution();  
}
```

Waarom dit interface tevens het (standaard Java) Comparable interface implementeert komt straks aan de orde. Je kunt dit bij de eerste opdrachten weglaten of door NetBeans (of Eclipse) zelf een default implementatie hiervoor laten genereren.

Onze *breadth-first solver* kunnen we als volgt beschrijven:

```
public class Solver {  
    Queue<Configuration> toExamine;  
  
    public Solver( Configuration g ) {  
        ...  
    }  
  
    public String solve () {  
        while ( ! toExamine.isEmpty() ) {  
            Configuration next = toExamine.remove();  
            if ( next.isSolution() ) {  
                return "Success!";  
            } else {  
                for ( Configuration succ: next.successors() ) {  
                    toExamine.add (succ);  
                }  
            }  
        }  
        return "Failure!";  
    }  
}
```

Het is eenvoudig te bedenken dat het geen zin heeft om tussenoplossingen die je al bent tegengekomen nog eens aan de queue van te bekijken puzzels toe te voegen. Het toevoegen zo'n puzzel levert je zeker geen kortere oplossing. Je zou zelfs in een oneindige berekening kunnen komen. Om dit te voorkomen onthouden we alle gevonden tussenoplossingen.

Voor de representatie van de puzzel zelf maken we gebruik van de volgende klasse *SlidingGame*. De toestand van de puzzel wordt geadministreerd in een tweedimensionale rij van integers. Verder wordt de positie van het gat expliciet in de klasse bijgehouden. We moeten vaak weten waar het gat zich bevindt en het is dus handig als we niet eerst hoeven zoeken in de tweedimensionale rij.

De gegeven klasse is verre van volledig. Een aantal methoden en mogelijk ook nog wat attributen zul je nog moeten toevoegen om het geheel werkend te krijgen.

```
public class SlidingGame implements Configuration {  
    public static final int N = 3, SIZE = N * N, HOLE = SIZE;
```

```

private int [][] board;
private int holeX, holeY;

public SlidingGame (int [] start) {
    board = new int [N][N];

    assert start.length == N*N: "Length of specified board incorrect";

    for( int p = 0; p < start.length; p++) {
        board[p % N][p / N] = start[p];
        if ( start[p] == HOLE ) {
            holeX = p % N;
            holeY = p / N;
        }
    }
}

@Override
public String toString() { ... }

@Override
public boolean equals(Object o) { ... }

@Override
public boolean isSolution () { ... }

@Override
public Collection<Configuration> successors () { ... }
}

```

5 Deelopdrachten

Maak het programma op de hieronder beschreven wijze af.

5.1 De klasse `SlidingGame`

Maak de klasse `SlidingGame` af. Omdat deze klasse het interface `Configuration` implementeert moet je in ieder geval de volgende methodes definiëren:

1. `isSolution`: geeft aan of de betreffende puzzel is opgelost of niet.
2. `successors`: levert de directe opvolgers van de huidige puzzel op. Stop deze puzzels in een of ander datatype dat het `Collection` interface implementeert, bijvoorbeeld een `LinkedList` of `ArrayList`.

5.2 De klasse `Node`

Het resultaat van de hierboven beschreven oplosser is wat teleurstellend; je krijgt enkel te zien of de puzzel wel of niet kan worden opgelost. In het eerste geval ben je natuurlijk ook benieuwd naar de oplossing zelf: Welke stukjes moet je verschuiven om de eindconfiguratie te bereiken? Om zo'n oplossing te kunnen geven is het nodig dat je vanaf iedere tussenoplossing z'n directe voorganger terug kunt vinden. Dit doen we door gebruik te maken van een aparte datastructuur, genaamd `Node`. Zo'n *node* bevat een configuratie en (een verwijzing naar) de knoop met de voorganger, die weer uit een configuratie en een verwijzing naar de voorganger van de voorganger bevat, enz. Om de gegevens die in de knopen kunnen worden opgeslagen algemeen te houden gebruiken we hier een *generic*. Het type van de opgeslagen elementen wordt pas bij gebruik van de datastructuur aangegeven.

```

public class Node <T> {
    private T item;
    private Node<T> previous;
}

```

```

    public Node (Node<T> from, T item) {
        this.previous = from;
        this.item      = item;
    }

    public T getItem() {
        return item;
    }

    public int length () { /* zie Node.java */ }
    public String toString () { ... }
}

```

- De methode `length` geeft het aantal knopen op het pad dat begint bij de wortel en leidt tot de huidige knoop.
- De methode `toString` geeft een string waarin alle elementen vanaf de wortel (de beginknoop) tot aan het element in de huidige knoop zijn weergegeven. Per knoop wordt zowel de padlengte vanaf de wortel gegeven als de `toString` van het element in die knoop. Voor een pad met puzzels krijgen we bijvoorbeeld:

```

0:
1 2
4 6 3
7 5 8
1:
1 2 3
4 6
7 5 8
2:
1 2 3
4 6
7 5 8
3:
1 2 3
4 5 6
7 8
4:
1 2 3
4 5 6
7 8

```

Verander nu de oplosser zodanig dat de hierboven gegeven verwijzingsstructuur van knopen op de juiste wijze wordt opgebouwd en aan het einde een correcte oplossing wordt opgeleverd (mits de puzzel oplosbaar is).

5.3 Bekeken puzzels

Om te voorkomen dat je tijdens het zoeken in (een oneindige reeks van) herhalingen terechtkomt, moet je op de een of andere manier bijhouden welke toestanden (puzzelconfiguraties) je al gehad hebt. Het makkelijkst is om hiervoor een lijst van puzzels bij te houden. Om te kijken of een puzzel nieuw is kun je dan met de `contains` methode nagaan of de nieuwe puzzel al in deze administratie voorkomt. Om dit onderdeel flexibel te houden is het verstandig gebruik te maken van het `Collection` interface. Naast `contains` bevat dit interface de methode `add` waarmee nieuwe configuraties kunnen worden toegevoegd.

Door het opslaan van al die configuraties gebruikt je programma veel geheugen. Indien je de melding krijgt dat alle *heap space* gebruikt is heb je mogelijk een fout gemaakt bij het voorkomen van oneindige

berekeningen. Als dat niet het geval is (test bijvoorbeeld een 2×2 puzzel), dan kun je het java programma meer geheugenruimte geven met de optie `-Xmx250m`. In NetBeans staat dit in het *File* menu bij de *project properties*, onder *run* bij *VM options*. Standaard is deze maximum heap space 128Mb. Met de gegeven optie vergroot je de heap space naar 250Mb.

5.4 Een basisoplossing

Maak met deze ingrediënten een Java programma dat een gegeven puzzel probeert op te lossen. Als de oplossing gevonden is dient je programma het pad met alle configuraties in de juiste volgorde af te drukken.

Overigens, niet alle puzzels zijn op te lossen. Door het omwisselen van 2 naast elkaar zittende stukjes in de puzzel verandert de *pariteit* en kun je door schuiven nooit meer een oplossing vinden. In het geval dat je programma een oplosbare puzzel krijgt aangeboden dient het een nette melding hiervan te geven.

5.5 Tussenoplossingen efficiënter administreren

Het aantal oplosbare configuraties in een $N \times N$ puzzel is $(N^2)!/2$. Zelfs voor een bescheiden 3×3 puzzel zijn dit al 181440 configuraties. Voordat je programma concludeert dat een gegeven puzzel niet op te lossen is zal het een lijst met alle bereikbare configuraties hebben aangemaakt. En om na te gaan of een puzzel nieuw zal deze lijst moeten worden afgelopen. Dit is dus per nieuwe puzzel $O(n)$ werk.

We kunnen dit zoeken naar eerder voorgekomen configuraties versnellen door gebruik te maken van een *hashtabel*. In een hashtabel worden elementen opgeslagen op basis van een *hashwaarde*. Ook bij het bepalen of een element al is opgeslagen wordt deze hashwaarde gebruikt. Dit kan in praktisch $O(1)$ tijd hetgeen een duidelijke verbetering is van de $O(n)$ bij normale lijsten.

Het is overigens niet erg als we voor twee verschillende puzzels dezelfde hashwaarde hebben. De hashtabel zal er zelf voor zorgen dat beide puzzels worden opgeslagen, zonder dat dit tot noemenswaardige vertragingen bij het opzoeken zal leiden.

Java heeft bibliotheken die zo'n hashtabel implementeren, bijvoorbeeld `HashSet`. De `HashSet` maakt gebruik van de methode `public int hashCode()` om een hashwaarde te bepalen. Deze methode is afkomstig uit de klasse `Object` en wordt dus door iedere klasse geërfd. Maar zoals zo vaak is de standaard implementatie van deze methode niet erg waardevol en dien je zelf een betere definitie te geven. De eis die aan de implementatie van `hashCode` gesteld wordt is dat objecten die gelijk zijn ook dezelfde hashwaarde moeten krijgen; Verder dienen ongelijke objecten zo veel mogelijk verschillende hashwaardes te krijgen.

Voor onze puzzel ligt het voor de hand om wat te rekenen met de plaats en de waarde van de puzzelstukjes. Zo zou je de hashwaarde van een puzzel kunnen berekenen met de volgende formule:

$$\text{hashwaarde} = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \text{board}[x][y] \cdot 31^{x+y \cdot N}$$

Aangezien `HashSet` net als lijsten het `Collection` interface implementeert hoeft je als het goed is maar heel weinig aan je programma te veranderen.

5.6 Een efficiënter zoekalgoritme

Als je een rechttoe rechtaan implementatie van het beschreven algoritme maakt zul je merken dat het oplossend vermogen vrij beperkt is: zodra je je programma een wat complexere puzzel aanbiedt zal het té veel tijd of geheugen nodig hebben om de oplossing te bepalen. Er zijn verschillende manieren denkbaar om het algoritme te verbeteren.

Een voor de hand liggende manier is om *breadth-first search* te verbeteren tot *best-first search*, waarin je de voortzettingen die het best lijken het eerst doet. We kunnen dit eenvoudig voor elkaar krijgen door de beste elementen vooraan in de queue te zetten.

Natuurlijk weten we pas echt wat de beste voortzetting is als we de oplossing van de puzzel (het pad van de startpuzzel naar de eindconfiguratie) kennen. We kunnen gelukkig een heuristiek gebruiken om kwaliteit van tussenoplossingen te schatten. Voor onze puzzel gebruiken we bijvoorbeeld de som van de *Manhattan distances* van alle puzzelstukjes als maat voor kwaliteit. De Manhattan distance is niets anders dan het aantal vakjes horizontaal en verticaal dat een puzzelstukje nog moet afleggen om de

gewenste lokatie (zijn positie in de eindconfiguratie) te bereiken. Om herberekening hiervan te voorkomen kan het handig zijn om deze Manhattan distance bij iedere puzzel op te slaan.

Om dit te implementeren is het voldoende om in plaats van een gewone queue een `PriorityQueue` te gebruiken. De datastructuur zorgt er zelf voor dat beste elementen vooraan komen. Om te bepalen welke plek een nieuw element krijgt gebruikt deze queue de methode **public int** `compareTo(T e)` uit het interface `Comparable<T>`. Dit verklaart dan ook meteen waarom `Configuration` het interface `Comparable<Configuration>` uitbreidt. Voor puzzels kunnen we `compareTo` bijvoorbeeld het verschil in Manhattan distance laten opleveren.

Algemeen gelden voor de `compareTo` methode de volgende voorwaarden:

- $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$ voor alle x en y . De functie `sgn` is de *signum-functie*. Het resultaat is $+1$ voor alle positieve argumenten, 0 als het argument 0 is, en -1 als het argument negatief is.

Misschien had je verwacht dat $x.\text{compareTo}(y) == -y.\text{compareTo}(x)$, maar dit hoeft dus niet het geval te zijn.

- De vergelijking dient transitief te zijn. Dat wil zeggen als $x.\text{compareTo}(y) > 0$ en $y.\text{compareTo}(z) > 0$ dan is ook $x.\text{compareTo}(z) > 0$.
- Soms is het handig als $x.\text{compareTo}(y) == 0$ dat dan ook $x.\text{equals}(y)$, maar dat is niet vereist. In deze opgave, bijvoorbeeld, kunnen twee puzzels met dezelfde Manhattan distance (en dus 0 als resultaat van `compareTo` hebben) toch verschillend zijn.

Omgekeerd is het wel zeer gewenst dat als $x.\text{equals}(y)$ dat dan ook $x.\text{compareTo}(y) == 0$.

Het is overigens waarschijnlijk dat je (het generische type van) de `Node` klasse ook moet aanpassen, immers, zodra je knopen aan een priority queue toevoegt moet je ze kunnen vergelijken. En voor die vergelijking heb je nodig dat de opgeslagen items (met het generische type `T`) ook vergeleken kunnen worden. Bedenk zelf hoe je dit kunt aangeven in de header van de `Node` klasse.

6 Hulpbestanden

Alle codefragmenten die in deze opdracht getoond worden, zijn ook terug te vinden op Blackboard in de vorm van een aantal klassedefinities. Je mag deze voor je eigen uitwerking gebruiken mits je ze op de juiste wijze van commentaar voorziet.

7 Inleveren

Vóór zondag 15 maart, 11:00 uur, via Blackboard. Lever alle `.java` files uit je project in (dus ook de bestanden die zijn voorgegeven).