Complexiteit & Algoritmiek

Backtracking

Joram Sjamaar
462398

Stijn van den Berg
466072

Inhoudsopgave

Doolhof

In deze opdracht hebben we gewerkt met backtracking. Dit is een ‘slimme’ methode om informatie mee op te zoeken. Het is slim omdat niet elke oplossing berekend hoeft te worden.

Ontwerp

De opdracht omschreef een doolhof met twee pionnen. Een pion mag alleen over een lijn van de kleur waar de één van de pionnen op staan.

Dat betekend dat we een aantal model classes hebben. Namelijk:

- DirectedLine
- Maze
- Node
- Pawn
- State

Bekijk ook het Class Diagram in de bijlagen

DirectedLine

Heeft twee eigenschappen: Kleur en welke Node deze lijn verbind.

Maze

Bevat alle logica en backtracking algoritme.

Node

Stelt een blokje/rondje voor in het doolhof. Heeft twee eigenschappen: Een nummer en kleur.

Pawn

Stelt een pion voor. Deze class is niet nodig. We hebben toch voor deze class gekozen omdat het ons duidelijkheid gaf. De pion houd alleen bij op welke Node deze staat.

State

Houd de toestand bij van het doolhof.

De meest makkelijke vergelijking die we kunnen maken, is een foto. Stel je voor dat je een foto maakt van een schaakspel. Je ziet dan precies waar alle spelstukken staan. Dit doet de State ook. Het slaat op waar de pionnen staan.

Data representatie van een doolhof

We hebben gekozen voor een adjacency list. Een node is hierin het hoofd element. Een node bevat directedlines. Zie tabel hieronder

Node	DirectedLines
FINISH (BLUE)	
1 (PINK)	[PINK, pointsTo: Node4 (GREEN)] [BLACK, pointsTo: Node5 (GREEN)]
2 (BLACK)	[GREEN, pointsTo: Node6 (ORANGE)] [PINK, pointsTo: Node12 (PINK)]
3 (GREEN)	[ORANGE, pointsTo: Node4 (GREEN)] [ORANGE, pointsTo: Node1 (PINK)]
4 (GREEN)	[BLACK, pointsTo: Node13 (ORANGE)]
5 (GREEN)	[ORANGE, pointsTo: Node9 (PINK)]
6 (ORANGE)	[PINK, pointsTo: Node10 (BLACK)] [GREEN, pointsTo: Node9 (PINK)]
7 (ORANGE)	[GREEN, pointsTo: Node2 (BLACK)]
8 (PINK)	[PINK, pointsTo: Node3 (GREEN)]
9 (PINK)	[BLACK, pointsTo: Node14 (GREEN)] [GREEN, pointsTo: Node4 (GREEN)]
10 (BLACK)	[GREEN, pointsTo: Node15 (ORANGE)]
11 (ORANGE)	[PINK, pointsTo: Node10 (BLACK)] [GREEN, pointsTo: Node12 (PINK)]
12 (PINK)	[GREEN, pointsTo: Node7 (ORANGE)]
13 (ORANGE)	[GREEN, pointsTo: Node8 (PINK)] [GREEN, pointsTo: Node18 (BLACK)]
14 (GREEN)	[GREEN, pointsTo: Node-1 (BLUE)] [ORANGE, pointsTo: Node20 (GREEN)]
15 (ORANGE)	[GREEN, pointsTo: Node22 (BLACK)] [PINK, pointsTo: Node-1 (BLUE)]
16 (GREEN)	[GREEN, pointsTo: Node15 (ORANGE)]
17 (GREEN)	[BLACK, pointsTo: Node11 (ORANGE)] [PINK, pointsTo: Node12 (PINK)] [BLACK, pointsTo: Node16 (GREEN)]
18 (BLACK)	[ORANGE, pointsTo: Node9 (PINK)] [ORANGE, pointsTo: Node20 (GREEN)]
19 (ORANGE)	[GREEN, pointsTo: Node18 (BLACK)]
20 (GREEN)	[BLACK, pointsTo: Node19 (ORANGE)] [ORANGE, pointsTo: Node21 (BLACK)]
21 (BLACK)	[ORANGE, pointsTo: Node22 (BLACK)] [BLACK, pointsTo: Node-1 (BLUE)]
22 (BLACK)	[ORANGE, pointsTo: Node17 (GREEN)]

Ondersteuning voor meerdere doolhoffen

De ondersteuning voor meerdere doolhoffen wordt gerealiseerd met een JSON inlezing van een doolhof. Dit wordt geplaatst in een adjacency list .

De maze wordt standaard ingelezen vanuit het *maze.json* bestand.

Dit JSON bestand bevat een array, met daarin Nodes en de daarbij behoordende properties

Specificatie

Array

- Object *(Node)*
  - Number *(int)*
  - Color *(“PINK” | “GREEN” | “BLACK” | “ORANGE” | “BLUE”)*
  - Lines *(Array)*
    - Object
      - Color *(“PINK” | “GREEN” | “BLACK” | “ORANGE” | “BLUE”)*
      - pointsTo *(int)*

Voorbeeld

```
[
    {
        "number": 1,
        "color": "PINK",
        "lines": [
            {
                "color": "PINK",
                "pointsTo": 4
            },
            {
                "color": "BLACK",
                "pointsTo": 5
            }
        ]
    }
]
```

Tests

Om de werking van het backtracking algoritme en het doolhof te testen maken we gebruik van JUnit5.4.

Er zijn drie unittesten.

totalStepsShouldBe37

Doolhof: *maze.json*

Deze test zal de functionaliteit van het backtracking algoritme testen. De kortst mogelijke oplossing voor dit doolhof is 37 stappen.

Let op! Er zit een bug in deze test. Als deze test tegelijkertijd met meerdere tests wordt gedaan dan neemt deze een andere 'afslag'. Op stap 26 gaat deze dan naar links i.p.v. naar rechts. Als je deze test alleen runt dan is de bug er niet.

emptyMazeShouldThrowException

Doolhof: *empty_maze.json*

Deze test verwacht een foutmelding. Als er een leeg doolhof wordt ingelezen, dan moet dit een fout opleveren.

mazeWithNoSolutionShouldThrowException

Doolhof: *impossible_maze.json*

Deze test verwacht een foutmelding. Als er doolhof zonder oplossing wordt ingelezen, dan moet dit een fout opleveren.

simpleMazeShouldTraverse

Doolhof: *simple_maze.json*

Deze test maakt gebruik van een heel simpel doolhof. En test daarmee ook weer het backtracking algoritme.

Resultaat

Het resultaat beperken we tot de hoofd opdracht. Dat is de maze.json.

Als we het algoritme uittesten dan komen we uit op 37 stappen.

Step 1

Pawn 1: 1, PINK

Pawn 2: 2, BLACK

Step 2

Pawn 1: 5, GREEN

Pawn 2: 2, BLACK

Step 3

Pawn 1: 5, GREEN

Pawn 2: 6, ORANGE

Step 4

Pawn 1: 9, PINK

Pawn 2: 6, ORANGE

Step 5

Pawn 1: 9, PINK

Pawn 2: 10, BLACK

Step 6

Pawn 1: 14, GREEN

Pawn 2: 10, BLACK

Step 7

Pawn 1: 14, GREEN

Pawn 2: 15, ORANGE

Step 8

Pawn 1: 20, GREEN

Pawn 2: 15, ORANGE

Step 9

Pawn 1: 21, BLACK

Pawn 2: 15, ORANGE

Step 10

Pawn 1: 22, BLACK

Pawn 2: 15, ORANGE

Step 11

Pawn 1: 17, GREEN

Pawn 2: 15, ORANGE

Step 12

Pawn 1: 17, GREEN

Pawn 2: 22, BLACK

Step 13

Pawn 1: 11, ORANGE

Pawn 2: 22, BLACK

Step 14

Pawn 1: 11, ORANGE

Pawn 2: 17, GREEN

Step 15

Pawn 1: 12, PINK

Pawn 2: 17, GREEN

Step 16

Pawn 1: 7, ORANGE

Pawn 2: 17, GREEN

Step 17

Pawn 1: 2, BLACK

Pawn 2: 17, GREEN

Step 18

Pawn 1: 2, BLACK

Pawn 2: 16, GREEN

Step 19

Pawn 1: 6, ORANGE

Pawn 2: 16, GREEN

Step 20

Pawn 1: 9, PINK

Pawn 2: 16, GREEN

Step 21

Pawn 1: 4, GREEN

Pawn 2: 16, GREEN

Step 22

Pawn 1: 4, GREEN

Pawn 2: 15, ORANGE

Step 23

Pawn 1: 4, GREEN

Pawn 2: 22, BLACK

Step 24

Pawn 1: 13, ORANGE

Pawn 2: 22, BLACK

Step 25

Pawn 1: 13, ORANGE

Pawn 2: 17, GREEN

Step 26

Pawn 1: 8, PINK

Pawn 2: 17, GREEN

Step 27

Pawn 1: 8, PINK

Pawn 2: 12, PINK

Step 28

Pawn 1: 3, GREEN

Pawn 2: 12, PINK

Step 29

Pawn 1: 3, GREEN

Pawn 2: 7, ORANGE

Step 30

Pawn 1: 3, GREEN

Pawn 2: 2, BLACK

Step 31

Pawn 1: 3, GREEN

Pawn 2: 6, ORANGE

Step 32

Pawn 1: 1, PINK

Pawn 2: 6, ORANGE

Step 33

Pawn 1: 1, PINK

Pawn 2: 10, BLACK

Step 34

Pawn 1: 5, GREEN

Pawn 2: 10, BLACK

Step 35

Pawn 1: 5, GREEN

Pawn 2: 15, ORANGE

Step 36

Pawn 1: 9, PINK

Pawn 2: 15, ORANGE

Step 37

Pawn 1: 9, PINK

Pawn 2: FINISH, BLUE

Waarom nu wel?

In de tests is er wat raars aan de hand. Dan komen we uit op 54 met dezelfde code. Waarom het gebeurd weten we niet, maar we weten wel waar het gebeurd. Bij stap 26 gebeurd er wat raars. De Maze gaat plotseling ergens anders heen.

Apl		MazeTests	
Step 25		**Step 25**	
Pawn 1:	13, ORANGE	Pawn 1:	13, ORANGE
Pawn 2:	17, GREEN	Pawn 2:	17, GREEN
Step 26		**Step 26**	
Pawn 1:	8, PINK	Pawn 1:	18, BLACK
Pawn 2:	17, GREEN	Pawn 2:	17, GREEN
Step 27		**Step 27**	
Pawn 1:	8, PINK	Pawn 1:	18, BLACK
Pawn 2:	12, PINK	Pawn 2:	11, ORANGE

Waarom dit gebeurd weten we niet. We krijgen het er ook niet uit. Uiteindelijk op stap 43 komt het toch nog goed.

Step 43	
Pawn 1:	17, GREEN
Pawn 2:	8, PINK

Vanaf hier gaat het door zoals in Apl (maar dan met pion 1 en 2 omgewisseld).

Conclusie

Uit de resultaten is te concluderen dat het algoritme in ieder geval werkt. De bug in de unittesten hebben we samen met Gerralt voor gezet. Na een grote worsteling leek het opgelost te zijn, maar het probleem kwam helaas weer terug.

Wat was de opdracht?

“De bedoeling van de puzzel is, dat één van de pionnen uiteindelijk op de FINISH komt. Met de pionnen kunnen zetten worden gedaan: een pion mag worden verplaatst over een pijl in dezelfde kleur als de positie van de andere pion. Eenzelfde pion mag meerdere keren achter elkaar worden verplaatst.”

En

“Ontwerp en implementeer eerst een geschikte datastructuur voor de representatie van het puzzeldoolhof. Ontwikkel daarna een algoritme voor het oplossen van de puzzel. Bedenk hierbij, dat op elk moment de toestand van de puzzel kan worden beschreven door bijvoorbeeld de twee posities waarop de pionnen staan. Vanuit zo’n toestand is een aantal zetten mogelijk. Door een zet uit te voeren ontstaat een nieuwe toestand. Een oplossing is dan een opeenvolging van toestanden met als laatste een eindtoestand: één van de pionnen staat op FINISH.”

Wat zijn de requirements die we hier uit halen?

- Één van de pionnen komt op de FINISH
- Een pion mag over dezelfde kleur pijl als de positive van de andere pion (in de richting van de pijl)
- Elk moment de toestand van de puzzle beschreven moet kunnen worden
- Door een zet uit te voeren ontsaat een nieuwe toestand
- Een oplossing is de opeenvolding van toestanden met als eindtoestand: één van de pionnen op de FINISH
- Bij een zet verplaats maximaal 1 pion

Wat werkt er dan precies?

Alles van de bovenstaande punten. Maar dat is makkelijk gezegd, dus dat gaan we nu onderbouwen.

Één van de pionnen komt op de FINISH / Een oplossing is de opeenvolding van toestanden met als eindtoestand: één van de pionnen op de FINISH
Zodra één van de pionnen op de FINISH komt, stopt het spel. Hiermee kan er dus altijd maar één pion op de FINISH staan.

Een pion mag over dezelfde kleur pijl als de positive van de andere pion (in de richting van de pijl)
De getNeighbours() methode geeft alleen de buren weer van pijlen waar die pion overheen kan. Een pion heeft dus geen besef van buren waar hij niet over kan.

Elk moment de toestand van de puzzle beschreven moet kunnen worden / Door een zet uit te voeren ontsaat een nieuwe toestand
Elke zet is automatisch een State. Een state houd bij waar de pionnen staan. De states worden in volgorde opgeslagen.

Bij een zet verplaats maximaal 1 pion
Zo is het gecodeerd. Een zet is eigenlijk een nieuwe State. En die verplaatst maar één pion.

Eindconclusie

Naast de requirements hebben we ook een aantal leuke extra's. Zoals het inlezen van maze's via een JSON bestand, en UNITtesten. Hiermee concluderen we dat de opdracht tot een mooi einde is gekomen en we trots kunnen zijn op ons product.

Bijlagen

Class Diagram