Gegevensstructuren en Algoritmen: Opdracht 2

Academiejaar 2021-2022

Inhoudsopgave

1 Wat is de bedoeling van deze opdracht?

In dit practicum moet je een solver maken voor het 8-puzzel probleem[1] (en zijn natuurlijke veralgemening). Hierbij maak je gebruik van het A* zoekalgoritme. Voor de geïnteresseerde student geven we hier enkele links naar meer info over het A* algoritme: [2, 3, 1].

Het doel van de reeks milestones zoals die op TOLEDO staan uitgelegd is om inzicht te verwerven over de performantie van een geïmplementeerde algoritme, dit correct te analyseren, en om te kunnen redeneren over mogelijke verbeteringen of inherente limitaties.

We voorzien zowel java als python bestanden om jullie op weg te helpen bij deze opdracht en implementatie van de milestones.

1.1 Probleem

Het 8-puzzel probleem is een populaire puzzel uitgevonden door Noyes Palmer Chapman in de jaren '70 van de 19e eeuw. Het wordt gespeeld op een 3×3 rooster met 8 vierkante tegels, genummerd van 1 tot 8 en een lege ruimte. Het doel is de tegels te herordenen zodat ze op volgorde staan. Je mag tegels horizontaal en verticaal verschuiven naar de lege ruimte. Hieronder tonen we een sequentie van geldige verplaatsingen van een initiële bordconfiguratie (links) tot het doel (rechts).

```
   1  3       1     3       1  2  3       1  2  3       1  2  3
4  2  5   =>  4  2  5   =>  4     5   =>  4  5     =>  4  5  6
7  8  6       7  8  6       7  8  6       7  8  6       7  8

initieel                                                 doel
```

Bij de oplossing staat de lege tegel (het nulvakje) dus rechts onderaan.

[1] Dit practicum is gebaseerd op een programmeeropdracht gelinkt aan het boek Algorithms 4th edition (Robert Sedgewick, Kevin Wayne), https://algs4.cs.princeton.edu/home/

1.2 Algoritme

We beschrijven hier een oplossing die een algemene methodologie in artificiële intelligentie illustreert, het A* zoekalgoritme. We definiëren een toestand in het spel als de configuratie van het bord, het aantal verplaatsingen om die configuratie te bekomen, en de vorige toestand. Plaats eerst de initiële toestand (de initiële configuratie, 0 verplaatsingen en `null` als vorige toestand) in een prioriteitsrij. Verwijder dan de toestand met de minimum prioriteit uit de prioriteitsrij en voeg alle naburige toestanden (diegene die met 1 verplaatsing bereikt kunnen worden) toe. Herhaal deze procedure tot de minimum toestand de doeltoestand is.

Het succes van deze aanpak hangt af van de keuze van de prioriteitsfunctie voor een toestand. We beschouwen twee prioriteitsfuncties:

1. Hamming prioriteitsfunctie. Het aantal tegels in de verkeerde positie, plus het aantal verplaatsingen om deze toestand te bereiken vanuit de initiële toestand. Intuïtief gezien, zal een toestand met een klein aantal verkeerde tegels dicht bij de doeltoestand liggen, en we verkiezen een toestand die met zo weinig mogelijk verplaatsingen bereikt kan worden.

2. Manhattan prioriteitsfunctie. De som van de afstanden (som van de verticale en horizontale afstand) van de tegels naar hun doelpositie, plus het aantal verplaatsingen om deze toestand te bereiken vanuit de initiële toestand.

Bijvoorbeeld, de Hamming en Manhattan prioriteiten van de toestand hieronder zijn respectievelijk 5 en 10. Merk op dat we de lege ruimte (de lege tegel) niet meetellen in de berekening van de Hamming of Manhattan prioriteiten.

```
8  1  3     1  2  3     1  2  3  4  5  6  7  8    1  2  3  4  5  6  7  8
4     2     4  5  6     ----------------------    ----------------------
7  6  5     7  8        1  1  0  0  1  1  0  1    1  2  0  0  2  2  0  3

initieel     doel       Hamming = 5 + 0           Manhattan = 10 + 0
```

We maken hierbij een belangrijke observatie: willen we een puzzel van een bepaalde toestand in de prioriteitsrij oplossen, dan is het totaal aantal verplaatsingen (inclusief diegene die reeds gedaan zijn) minstens gelijk aan de prioriteit, zowel voor de Hamming als de Manhattan prioriteitsfunctie. (Voor de Hamming prioriteit klopt dit omdat elke verkeerde tegel minstens 1 verplaatsing moet doen om zijn doelpositie te bereiken. Voor de Manhattan prioriteit klopt dit omdat elke verkeerde tegel zijn Manhattan afstand tot de doelpositie moet afleggen.)

Bijgevolg, van zodra we een toestand uit de prioriteitsrij halen, hebben we niet alleen een sequentie van verplaatsingen van de initiële bordconfiguratie tot het bord horend bij de toestand, maar is die sequentie ook de kortste. (Uitdaging voor de wiskundigen onder jullie: bewijs dit.)

1.3 Optimalisatie

Eens je dit algoritme geïmplementeerd hebt, zal je merken dat bepaalde bordconfiguraties meerdere keren in de prioriteitsrij voorkomen. Om dit enigszins te vermijden, kan je naburige bordconfiguraties weigeren als ze dezelfde zijn als de vorige toestand.

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

 vorig        huidig      weiger
```

1.4 Oplosbaarheid

Niet alle initiële bordconfiguraties kunnen tot de doeltoestand leiden. Test daarom eerst de invoer. We leggen hier uit hoe je dat kan doen.

We stellen een bord voor door alle rijen achter elkaar te plaatsen. Bijvoorbeeld de puzzel

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

stellen we voor als $b = 8, 1, 3, 4, 0, 2, 7, 6, 5$. We maken hier een onderscheid tussen de waarde van een tegel, en de positie van een tegel in het bord. De waarde is het getal op de tegel, de positie is de plaats waar de tegel zich in het bord bevindt, waarbij we beginnen te tellen vanaf 1.

We definiëren een functie p die gegeven een bord en een waarde van een tegel, de positie teruggeeft. In ons voorbeeld is dus $p(b, 2) = 6$.

We definiëren een functie *oplosbaar* die gegeven een bord b, teruggeeft of het bord oplosbaar is:

$$oplosbaar(b) = \frac{\prod_{i<j} (p(b,j) - p(b,i))}{\prod_{i<j} (j - i)} \geq 0 \tag{1}$$

Met andere woorden, als de bovenstaande ongelijkheid waar is, dan is de puzzel oplosbaar. Hierbij lopen i en j over geldige waarden van tegels, exclusief het nul vakje. Voor een 3×3 bord, zijn i en j dus waarden uit de verzameling $\{1, 2, 3, 4, 5, 6, 7, 8\}$. $\prod$ is de notatie voor een product (zoals $\sum$ de notatie is voor som).

De functie *oplosbaar* gaat er echter van uit dat in het gegeven bord het nulvakje helemaal rechts onderaan staat. Voordat je dus de functie *oplosbaar* gebruikt, verplaats je de lege tegel (het nulvakje) naar zijn doelpositie. Dit doe je via geldige verschuivingen totdat het rechts onderaan staat.

Een voorbeeld van een onoplosbare puzzel is geven in file `puzzle-impossible3x3.txt`:

```
% cat puzzle-impossible3x3.txt
3
 1  2  3
 4  5  6
 8  7  0

% ant -Dboard=boards/puzzle3x3-impossible.txt
Geen mogelijke oplossing
```

2 Technisch

Vermits we jouw implementatie niet op correctheid nagaan (dat moet je zelf doen), kan je uiteraard zelf kiezen op welke manier je één en ander software-matig aanpakt. We voorzien echter wel enkele startfiles die de opstart wat eenvoudiger kunnen maken, maar gebruik hiervan is uiteraard niet verplicht.

2.1 Implementatie

Bouw verder op de gegeven files `Solver.java of solver.py` en `Board.java of board.py` om een een initiële bord configuratie in te lezen, en om een optimale oplossing uit te printen. Schrijf ook het totaal aantal verplaatsingen uit. Je mag de `PriorityQueue` van Java/Python gebruiken. De invoer bestaat uit de dimensie van het bord N gevolgd door de $N \times N$ initiële configuratie. De lege ruimte wordt aangeduid door 0. Een voorbeeld van mogelijk input is:

```
% cat puzzle04.txt
3
0  1  3
4  2  5
7  8  6
```

We bieden een framework aan op basis van `ant`. Hierover vind je meer informatie in de sectie Hoe Ant gebruiken. In de directory `boards` zitten enkele puzzels die je kan gebruiken om je implementatie te testen. Bijvoorbeeld, een mogelijke output voor `puzzle04.txt` is:

```
   1  3
4  2  5
7  8  6

1     3
4  2  5
7  8  6

1  2  3
4     5
7  8  6

1  2  3
4  5
7  8  6

1  2  3
4  5  6
7  8

Minimum aantal verplaatsingen = 4
```

Je programma moet werken voor willekeurige $N \times N$ borden (voor elke N groter dan 1), zelfs als de uitvoeringstijd daarmee heel groot wordt. Indien je een `OutOfMemoryError` exception krijgt, kan je het maximaal aantal geheugen dat Java mag gebruiken verhogen via:

Linux: `export _JAVA_OPTIONS="-Xmx1024m"`

Windows: `set _JAVA_OPTIONS="-Xmx1024m"`

Voer deze commando's uit in het terminalvenster waar je ook `ant` gebruikt.

2.2 Uitwerking

Voorbereiding:

1. Installeer het programma Ant én voer het eens uit. Doe dit in het begin, zo kom je vlak voor de deadline niet voor verrassingen te staan. Dit document bevat een sectie Hoe Ant installeren.

2. Implementeer `Board.java of board.py` en `Solver.java of solver.py`. Voor java wordt de main methode in `Main.java`) uitgevoerd via het commando:

   ```
   ant run -Dboard=boards/puzzle04.txt
   ```

   In python run je:

```
python main.py boards/puzzle04.txt
```

Dit commando kan je natuurlijk aanpassen om je algoritme te laten uitvoeren op andere puzzels.

Referenties

[1] Rajiv Eranki. Pathfinding using A* (A-star), 2002. `http://web.mit.edu/eranki/www/tutorials/search/`.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2):100–107, July 1968. `http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf`.

[3] Amit Patel. Introduction to A*. `http://www.redblobgames.com/pathfinding/a-star/introduction.html`.

A Hoe Ant installeren

Ant is niet standaard bijgeleverd bij Java en ook niet bij Windows. Je moet Ant dus eerst installeren.

Windows thuis: Het tweede google-resultaat over hoe je Ant installeert onder Windows levert `https://code.google.com/archive/p/winant/downloads` op. Dit is een heel eenvoudige installer. Deze installer vraagt wat de directory is waar JDK is geïnstalleerd; dit is typisch zoiets als `C:\Program Files\Java\jdk1.7.0_17\bin` (afhankelijk van welke JDK je precies hebt).

Linux thuis: Voor Ubuntu en Debian: de installatie is eenvoudigweg "`sudo apt-get install ant`" intypen in een terminalvenster. Voor andere distributies: gebruik je package manager.

PC-labo computerwetenschappen (gebouw 200A): Ant is reeds geïnstalleerd.

LUDIT pc-labo: Ongekend; het is waarschijnlijk veel makkelijker Ant op een eigen machine te installeren.

Mac OS X: [2]

1. Open een terminal
2. Type volgende commando's:

   ```
   /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
   brew install ant
   ```

   Ant is nu geinstalleerd. Als je bij het uitvoeren van "ant" de error krijgt dat je JDK moet installeren, dan moet je dat doen. Je hebt JDK (Java Development Kit) nodig om Java programma's te compileren in het algemeen, dus ook als je Java programma's compileert via Ant.

[2] Bron: http://gauravstomar.blogspot.be/2011/09/installing-or-upgrading-ant-in-mac-osx.html

B Hoe Ant gebruiken

1. Start een terminalvenster (dit werkt ook onder Windows: menu start, dan execute, dan "`cmd`" intypen; zie anders `http://www.google.com/search?q=how+to+open+windows+command`)

2. Navigeer naar de directory waar je bestanden voor dit practicum staan, meer bepaald de directory waar zich `build.xml` in bevindt. Met "`cd`" verander je van directory en met "`ls`" (Unix) of "`dir`" (Windows) toon je de bestanden en directories in de huidige directory.

3. Typ "`ant release`". Ant doet een aantal checks om je tegen een aantal fouten te beschermen. Check dus of Ant geen error gaf.