

Gegevensstructuren en Algoritmen: Practicum 2

Academiejaar 2016-2017

Inhoudsopgave

1 Gedragscode	1
2 Wat is de bedoeling van dit practicum?	2
3 Technische uitwerking	5
4 Criteria	6
5 Deadline	7
6 Communicatie	7
A Hoe Ant installeren	7
B Hoe Ant gebruiken	8

1 Gedragscode

(Laatste update gedragscode: 5 maart 2015)

De practica worden gequoteerd, en het examenreglement is dan ook van toepassing. Soms is er echter wat onduidelijkheid over wat toegestaan is en niet inzake samenwerking bij opdrachten zoals deze.


De oplossing en/of verslag en/of programmacode die ingediend wordt moet volledig het resultaat zijn van werk dat je zelf gepresteerd hebt. Je mag je werk uiteraard bespreken met andere studenten, in de zin dat je praat over algemene oplossingsmethoden of algoritmen, maar de bespreking mag niet gaan over specifieke code of verslagtekst die je aan het schrijven bent, noch over specifieke resultaten die je wenst in te dienen. Als je het met anderen over je practicum hebt, mag dit er dus NOOIT toe leiden, dat je op om het even welk moment in het bezit bent van een geheel of gedeeltelijke kopie van het opgeloste practicum of verslag van anderen, onafhankelijk van of die code of verslag nu op papier staat of in elektronische vorm beschikbaar is, en onafhankelijk van wie de code of het verslag geschreven heeft (mede-studenten, eventueel uit andere studiejaar, volledige buitenstaanders, internet-bronnen, e.d.). Dit houdt tevens ook in dat er geen enkele geldige reden is om je code of verslag door te geven aan mede-studenten, noch om dit beschikbaar te stellen via publiek bereikbare directories of websites.

Elke student is verantwoordelijk voor de code en het werk dat hij of zij indient. Als tijdens de beoordeling van het practicum er twijfels zijn over het feit of het practicum zelf gemaakt is (bvb. gelijkaardige code, grafieken, of oplossingen met andere practica), zal de student gevraagd worden hiervoor een verklaring te geven. Indien dit de twijfels niet wegwerkt, zal er worden overgegaan tot het melden van een onregelmatigheid, zoals voorzien in het onderwijs- en examenreglement (zie <http://www.kuleuven.be/onderwijs/oer/>).

2 Wat is de bedoeling van dit practicum?

In dit practicum moet je een solver maken voor het 8-puzzel probleem (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 dit practicum is om inzicht te verwerven over de performantie van je geïmplementeerde algoritme, dit correct te analyseren, en om te kunnen redeneren over mogelijke verbeteringen of inherente limitaties.

 **De focus van dit practicum ligt opnieuw op het verslag!**

2.1 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 bord configuratie (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.

2.2 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.)

2.3 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

2.4 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 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.

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

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
```

2.5 Implementatie

Bouw verder op de gegeven files `Solver.java` en `Board.java` 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 gebruiken. Aangezien we onze eigen testen runnen op jouw code mag de interface van deze klassen niet worden gewijzigd! 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
```

Net als bij de vorige assignment, bieden we ook nu weer 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. Wij zullen je inzending testen met maximaal 1 GiB RAM.

2.6 Verslag

Naast het implementeren van de solver, vragen we om onderstaande vragen te beantwoorden. Plaats ook nu weer je verslag in de vorm van een PDF-bestand in de map **report**.

1. Geef voor elk van de volgende puzzels het minimum aantal verplaatsingen om de doeltostand te bereiken. Geef ook de uitvoeringstijd van het A* algoritme voor de Hamming en de Manhattan prioriteitsfuncties. Duid aan wanneer de oplossing niet binnen een redelijke tijd (< 5 minuten) gevonden kan worden.

Puzzel	Aantal verplaatsingen	Hamming (s)	Manhattan (s)
puzzle28.txt	.	.	.
puzzle30.txt	.	.	.
puzzle32.txt	.	.	.
puzzle34.txt	.	.	.
puzzle36.txt	.	.	.
puzzle38.txt	.	.	.
puzzle40.txt	.	.	.
puzzle42.txt	.	.	.

2. Wat is de complexiteit (aantal array accesses) van de prioriteitsfuncties **hamming** en **manhattan**, in functie van de bord grootte N ? Gebruik \sim -notatie en leg kort uit.
3. Leg uit hoe je **isSolvable** hebt geïmplementeerd. Wat is de complexiteit (aantal array accesses) in functie van de bord grootte N ? Geef ook de complexiteit van hulpfuncties indien je die gebruikt. Gebruik \sim -notatie en leg uit.
4. Hoeveel bord posities zitten er worst-case in het geheugen, in functie van de bord grootte N ? Geef een zo laag mogelijke bovengrens en leg uit. Je mag veronderstellen dat een bordconfiguratie nooit meerdere keren in de prioriteitsrij zit.
5. Zijn er betere prioriteitsfunctie(s)? Indien ja, leg uit hoe deze werken. Wat is de verwachte impact op uitvoeringstijd en geheugen verbruik?
6. Indien je willekeurige 4×4 of 5×5 puzzels zou willen oplossen, wat zou je verkiezen: meer toegelaten tijd (bijvoorbeeld $10 \times$ zo lang), meer geheugen (bijvoorbeeld $10 \times$ zo veel), of een betere prioriteitsfunctie? Waarom?
7. Denk je dat er een efficiënt algoritme bestaat wat, zelfs voor grote puzzels, de optimale oplossing binnen praktische tijd kan vinden? Leg uit.

Merk op dat de parameter N staat voor de hoogte/breedte van het bord. Dus een bord van grootte N bestaat uit $N \times N$ vakjes.

3 Technische uitwerking

1. Voorbereiding:

- (a) 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.
- (b) Schrijf JUnit tests in `UnitTests.java` die automatisch de hulpfuncties zoals `Bord.hamming`, `Bord.manhattan`, `Bord.isSolvable`, enz uitvoert. We raden je ook aan om enkele tests te schrijven die automatisch puzzels oplost en controleert of de oplossing optimaal en correct is.
Schrijf grondig tests voor `Bord.hamming` en `Bord.manhattan` zodat die goed getest zijn. Zo voorkom je dat een foutje hierin zorgt dat je hele programma fout is in het vinden van de optimale oplossing.
- (c) Implementeer `Board.java` en `Solver.java`. De main methode (in `Main.java`) wordt uitgevoerd via het commando:

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

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

- (d) Voer tijdens je implementatie regelmatig `ant test` uit. Dit checkt je oplossing op een aantal veelvoorkomende (maar niet alle!) fouten. Ook je eigen JUnit tests worden dan uitgevoerd (de tests in bestanden waarin het woord “Test” in de bestandsnaam voorkomt worden uitgevoerd). Als je je tests of de main methode via Eclipse wilt uitvoeren, zal je in Eclipse `lib/libpract.jar` moeten toevoegen aan je build path.

2. Het echte werk

- (a) Schrijf het verslag. Zie sectie Wat is de bedoeling van dit practicum?. Denk eraan om tijdens het implementeren je code met Ant te testen!

3. Releases

- (a) Voer `ant test` uit om te controleren op een aantal veelvoorkomende fouten.
- (b) Controleer of op je verslag je naam en studentnummer staat zodat we bij het printen weten welk verslag van wie is.
- (c) Maak een zip file met Ant. Deze opgave bevat een sectie Hoe Ant gebruiken. Ant is verplicht. Zo heeft iedere zip-file dezelfde directory structuur; ervaring leert dat dit niet lukt als studenten de ZIP file met de hand maken. Hierdoor kunnen we sneller verbeteren en dus sneller feedback geven. Als je .zip file duidelijk niet met Ant is gemaakt, zullen we hem niet verbeteren.
- (d) Ant maakt een zip file `build/firstname.lastname.studentnumber.zip`. Vul hier je voornaam, achternaam en studentnummer (inclusief letters ‘r’, ‘s’ of ‘m’) in in de bestandsnaam. Verwissel niet je voornaam en je achternaam.
 - i. Fout: ~~Janssens-Jan-r0123456~~.zip
 - ii. Fout: ~~Jan-Janssens-0123456~~.zip
- (e) Open je ZIP file en controleer of alles er in zit.
- (f) Upload je ZIP file op Toledo. Je hoeft geen papieren verslag in te dienen.
Indien Toledo down is, mail dan een screenshot hiervan en je zip file naar de verantwoordelijke van dit practicum (zie sectie Communicatie).

4 Criteria

Je wordt in de eerste plaats beoordeeld op je verslag. Hiernaast moet je algoritme een optimale oplossing geven door gebruik te maken van geldige verschuivingen.

5 Deadline

De deadline is dinsdag 2 mei 2017 14:00. Laattijdige inzendingen worden niet aanvaard.

6 Communicatie

Stel je vragen via het Toledo forum.

De verantwoordelijke van dit practicum is **willem punt penninckx** apenstaartje **cs** punt **kuleuven** punt **be**. Hier kan je ook naar mailen voor zaken die je niet publiekelijk kan communiceren. Andere practica kunnen andere verantwoordelijken hebben.

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 <http://code.google.com/p/winant/> 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.

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

Mac OS X: ¹

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

```
curl -O http://apache.belnet.be/ant/binaries/apache-ant-1.9.6-bin.zip
unzip apache-ant-1.9.6-bin.zip
sudo mkdir -p /usr/local/
sudo cp -rf apache-ant-1.9.6 /usr/local/apache-ant
```

¹Bron: <http://gauravstomar.blogspot.be/2011/09/installing-or-upgrading-ant-in-mac-osx.html>

```
export PATH=/usr/local/apache-ant/bin:$PATH
echo 'export PATH=/usr/local/apache-ant/bin:$PATH' >> ~/.profile
```

Ant is nu geïnstalleerd. 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.

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. Type “**ant release**”. Ant doet een aantal checks om je tegen een aantal fouten te beschermen. Check dus of Ant geen error gaf.