Heuristieken verslag: Rush Hour probleem

Nicol Heijtbrink 10580611
Nicole Fereira Silverio 10521933
Sander de Wijs 10582134

December 2016

1 Inleiding

Het spel Rush Hour bestaat uit een bord van 6 bij 6 vakjes waarop auto's en vrachtwagens zijn geplaatst waarbij een auto bezet 2 vakjes en een vrachtwagen 3. Één van de auto's is rood gekleurd en om het spel te winnen moet deze rode auto het bord kunnen verlaten via een gedefinieerde uitgang. Dit wordt bereikt door alle auto's die de route blokkeren te verplaatsen. De uitgang is altijd in dezelfde rij als de rode auto. Het doel is om algoritmes te programmeren die dit probleem op kunnen lossen in zo min mogelijk stappen en idealiter in zo min mogelijk iteraties en tijd. We bekijken in totaal 7 verschillende borden; 3 van 6x6, 3 van 9x9 en 1 van 12x12.

Figuur 1: Bord 1 en bord 7. Alle 7 borden dienen opgelost te worden.[2]

Een legale zet is het verplaatsen van een auto in de richting van zijn oriëntatie zolang er geen andere auto in de weg staat. Een verticaal geplaatste auto kan dus omhoog en omlaag verschuiven en een horizontale auto kan naar links en rechts verschuiven. De auto kan nooit van oriëntatie veranderen. Elke verschuiving wordt gezien als één zet.

Tabel 1: Toestandsruimtegrootte per bord, berekend met formule 1.

Bord (dimensie)	Toestandsruimtegrootte
Bord 1 (6)	15.689,00
Bord 2 (6)	244.140.629,00
Bord 3 (6)	244.140.629,00
Bord 4 (9)	69.001.951.985,00
Bord 5 (9)	18.014.398.509.599.600,00
Bord 6 (9)	18.014.398.515.246.800,00
Bord 7 (12)	144.209.936.106.509.000.000.000.000.000,00

De grootte van de toestandsruimte van RushHour is het aantal unieke configuraties waarin het bord kan staan. Het eerste bord heeft 36 vakjes, met daarop 6 auto's en 3 vrachtwagens, die in totaal 21 vakjes bezetten. Van elke auto kan gezegd worden dat het op 5 verschillende manieren in een lege rij kan staan. Voor een vrachtwagen zijn dat er 4. Met de volgende formule, waar n het aantal auto's is, m het aantal vrachtwagens en 9 de dimensie van het bord, kan een grove toestandsruimtegrootte per bord berekend worden:

$$(x-1)^n * (x-2)^m \tag{1}$$

Deze formule toegepast op alle borden geeft de resultaten die te zien zijn in tabel 1. Dit is echter verre van nauwkeurig, gezien deze formule geen rekening houdt met de regel dat auto's elkaar niet mogen kruisen. Het geeft wel aan hoe sterk de toestandsruimte toe neemt bij het vergroten van een bord en het aantal voertuigen op een bord. Ook houdt de formule dus geen rekening met de bewegingsvrijheid op het bord. De bewegingsvrijheid kan uitgedrukt worden in het aantal vrije vakjes op een bord. Echter niet elk vrije vakje kan ook per se benut worden aangezien wanneer er geen voertuig met zijn kopse kant aan grenst er geen legale zet gedaan worden om het vrije vakje te bezetten. Dit verminderd dus de feitelijke bewegingsvrijheid.

Het winnen van het spel kan dus eindigen in verschillende configuraties. En er leiden verschillende wegen naar zo'n winnnende configuratie. Het probleem wat we in dit verslag zullen proberen op te lossen is wat de kortste weg is naar zo'n configuratie.

2 Methoden

2.1 Construeren van spelbord

Alle programma 's die gebruikt worden voor het oplossen van het rush hour probleem zijn geschreven in Python. Elk algoritme begint met het opbouwen van het speelbord met auto's. Voor de auto's is een 'Auto' klasse gemaakt waar voor elke auto en vrachtwagen een object van wordt gemaakt. Bij het aanmaken van een auto object worden de x en y coördinaten, lengte (2 (= auto)

of 3 (= vrachtwagen)), oriëntatie en een unieke id (integer) meegegeven en geïnitialiseerd. Vervolgens wordt van de 'Spel' klasse een object met een speelbord, die een dimensie mee krijgt, gemaakt waar alle auto objecten op geplaatst kunnen worden. Het bord zelf wordt gecreëerd middels een (geïmporteerde) numpy 'zero's' functie die een 2 dimensionale array met nullen maakt. Bij het plaatsten van een auto wordt die 0 overschreven met het nummer van de ID van de auto die op die plaats komt te staan zoals te zien in figuur 2.

```
[[0 0 7 2 2 8]
 [0 0 7 0 0 8]
 [0 0 7 1 1 8]
 [0 0 0 9 3 3]
 [4 5 5 9 0 0]
 [4 0 0 9 6 6]]
```

Figuur 2: De beginstaat van bord 1, zoals weergegeven in een 2D rij.

Verder staan in de spel klasse functies die in elke richting kunnen checken of een auto kán bewegen en functies die de auto ook daadwerkelijk bewegen. Door middel van verschillende functies die de bovengenoemde functies aanroepen zijn de verschillende algoritmes opgebouwd. Daarnaast heeft elk algoritme heeft zijn eigen functies om tot een uiteindelijke oplossing te komen.

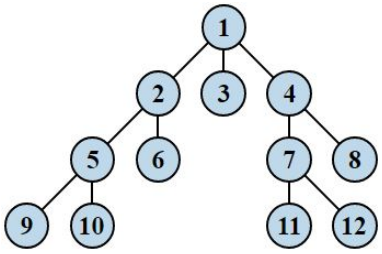
2.2 Simpel algoritme

Het allereerste algoritme bewoog de Auto's altijd wanneer dat mogelijk was, op de volgorde waarin Auto's ingeladen werden. De rode (te bevrijden) auto wordt altijd als eerste ingeladen, dan worden alle auto's van linksboven tot rechts onder ingeladen en als laatste worden de vrachtwagens in dezelfde volgorde als auto's ingeladen. Met behulp van een for loop werden alle auto's afgegaan. Horizontale Auto's werden naar rechts verplaatst, als dat niet mogelijk was bewogen ze naar links en als dat ook niet lukte ging het algoritme naar de volgende auto. Voor verticale Auto's werd eerst gepoogd ze naar beneden te bewegen, lukte dat niet dan probeerde de auto naar beneden te bewegen en als dat ook niet lukte ging het weer door naar de volgende auto. Dit ging door tot de rode auto op de winnende positie staat. Op deze manier werd de winnende positie echter nooit bereikt, omdat het algoritme in oneindige loops terecht kwam. Dit kan gebeuren wanneer de Auto's telkens eenzelfde beweging in dezelfde volgorde maken en dus in feite in een rondje blijven rijden. Dit werd opgelost door elke unieke bord configuratie die gemaakt werd, door het doen van een zet, bij te houden. Dit werd opgeslagen in een set, waarin elke configuratie staat in de vorm van een 'string' die het bord met de Auto's representeert (het omzetten van de 2D rij naar een 'string' wordt met een aparte functie gedaan). Middels dit archief kan

gecontroleerd worden of de zet die gedaan kan worden daadwerkelijk leidt tot een nieuwe bord configuratie. Was dat niet het geval, dan werd de zet ook niet uitgevoerd om oneindige loops voorkomen. Hiermee zou in theorie een oplossing gevonden kunnen worden, maar in praktijk duurt dit erg lang omdat elke in feite willekeurige zet die mogelijk is ook gezet wordt.

2.3 Breadth first algoritme

Op basis van het simpele algoritme is een breadth first search (BFS) algoritme gemaakt. BFS is een algoritme dat begint bij de basis van een zoek boom (node 1 in figuur 3). Het zal vervolgens als eerste de kinderen afgaan (2, 3 en 4). Daarna kijkt het naar de kinderen van het eerste kind (5 en 6) en vervolgens naar die van het 2e (geen kinderen) en 3e kind (7 en 8) enzovoorts om zo van elke 'laag' de hele breedte door te zoeken voor het naar een volgende laag gaat. In dit geval is de eerste node de start configuratie van het spelbord en elk kind is een unieke configuratie van het spelbord die vanuit zijn ouder gemaakt kan worden. Een nieuw kind ontstaat wanneer één enkele zet gedaan wordt.
Een ander vergelijkbaar en veelgebruikt algoritme, het depth-first search algoritme, gaat de boom per tak af (in figuur 3 dus: 1, 2, 5, 9). Dit kan ervoor zorgen dat een oplossing met veel meer zetten gevonden wordt. Aangezien het belangrijk is dat een oplossing met zo min mogelijk zetten gevonden wordt is er gekozen voor een implementatie van BFS in plaats van depth-first search.
De oplossing die een BFS algoritme vindt is overigens niet per definitie de enige oplossing, maar het is wel gegarandeerd de snelste oplossing. In dit geval dus de oplossing met het minste aantal mogelijke zetten.



Figuur 3: Een search tree die doorlopen kan worden middels een breadth first algoritme.[1]

2.3.1 Implementatie

Het belangrijkste concept nodig voor een BFS is het gebruik van een queue. In dit geval is de queue klasse van python, met bijkomende functies, geïmporteerd. Het algoritme maakt gebruik van 2 queues; 1 voor de bord staten en 1 voor de auto's. Dit is nodig omdat voor een volgende zet of staat is de configuratie van het bord nodig, maar ook alle auto's met hun positie. In het algoritme

checken verschillende functies of een zet mogelijk is en leidt tot een nieuwe configuratie van het bord, als dit het geval is wordt voor die zet een deepcopy van het bord en een deepcopy van de auto's in de queues gezet. De zet wordt nog niet uitgevoerd, want eerst worden alle mogelijke zetten voor alle auto's op een bord gecontroleerd. Pas als de zet uit de queue gehaald wordt deze ook echt uitgevoerd.
Om verschillende eigenschappen van het algoritme bij te houden zijn er 'dictionaries' aangemaakt (tijdens initialisatie van Spel) die het aantal tot op dat moment uitgevoerde zetten meegeeft aan de bord configuratie als een 'key-value' paar. Tevens is er een 'dictionary' die elk ouder-kind paar die doorlopen wordt bevat om zodra de winnende staat is bereikt het pad wat daartoe geleid heeft te kunnen herleiden.
In een functie waarbij een kind uit de queue gehaald wordt en het hele proces herhaald wordt tot de winnende positie voor de rode auto bereikt is wordt het spel helemaal doorgerekend middels een while loop die elke iteratie checkt of de winnende positie bereikt is. Hierin wordt ook een teller bijgehouden voor het aantal iteraties wat het algoritme nodig heeft om de winnende positie te bereiken.

2.4 A* algoritme

Het volgende algoritme wat weer voortgebouwd was van het BFS algoritme is een A* algoritme. Dit werkt op dezelfde basis maar geeft elke bord kost, en haalt dan op basis van die kost het 'beste' bord als eerste uit de queue. De kost van een bord wordt bepaald aan de hand van bepaalde regels, ook wel heuristieken genoemd. Dit zou de efficiëntie van het algoritme moeten verhogen door sneller een goed bord te vinden, wat dus eerder bij de oplossing komt. Dit leidt tot minder iteraties van het algoritme, maar niet altijd tot het minimale aantal zetten. Omdat grotere en moeilijkere borden erg moeizaam zijn om met een BFS door te zoeken omdat er simpelweg te veel opties zijn om in een aanzienlijke tijd met een reguliere computer te doorzoeken zijn de heuristieken erg belangrijk om een oplossing te kunnen vinden voor ingewikkeldere borden. Ook zorgt dit ervoor dat de grootte van queue niet zo groot wordt dat het geheugen van een reguliere computer niet toereikend is.

2.4.1 Implementatie

De basislijn van kosten wordt op 500 gezet zodat een zet niet zo makkelijk negatief uitkomt. Vervolgens wordt per auto gekeken of het aan één of meerdere van de voorwaarden in tabel 2 voldoet. Aan de hand van aan welke voorwaarden het voldoet wordt een kost voor die zet berekend middels de waarden in tabel 2. De zet wordt dan vervolgens in een priorityqueue gezet met zijn kost, auto's en bord. Om dit te verwezenlijken is een klasse PQueueitem aangemaakt die een prioriteit, een lijst met alle auto's en een staat van het bord heeft. Per zet wordt een PQueueitem object aangemaakt en die wordt in de queue gezet. De priorityqueue haalt de objecten met de laagste kost, dus priority, er als eerste

uit om zo voordelige zetten als eerste uit te voeren. Aan elke zet zijn echter nog wel extra kosten verbonden (het aantal zetten op dat moment * 10), ongeacht het type, om zo paden met meer zetten later te vinden dan paden met minder zetten. Het aantal zetten en iteraties worden op dezelfde manier bijgehouden zoals dat bij het BFS algoritme gedaan is.

Tabel 2: Kosten per heuristiek, A* algoritme; De kosten/korting wordt aan/van de totale kosten toegevoegd/afgetrokken.

Heuristiek	Kosten of korting
Rode auto staat op winnende positie	-400
Rode auto	-200
Auto met lengte 3	-200
In de rij van (en voor) rode auto	-100
Auto staat op linkerhelft van het bord	+100

Of dit de juiste heuristieken zijn was niet direct duidelijk. De truc ligt dan ook niet zo zeer in het bedenken van de heuristieken zelf, maar de daaraan verbonden kosten toekennen. Het huidige algoritme maakt gebruik van een absoluut verband; in het geval van meerdere kortingen worden deze bij elkaar op geteld.
Vanuit deze baseline zijn alle verschillende heuristieken nog aangepast van waarde, zowel naar boven als naar beneden. Maar naar trail en error onderzoek bleek de combinatie van de heuristieken in tabel 2 het beste te werken.

2.5 Beam Search

Een toevoeging op het gebruikte A* algoritme is het 'prunen' van de queue; na elke 100 iteraties wordt de slechte (ofwel duurste) 50% borden die op dat moment in de queue zitten eruit gehaald. Dit wordt ook wel een Beam Search algoritme genoemd. Alleen de meest veelbelovende kinderen worden op deze manier bekeken, en de rest wordt weg gegooid. Deze manier van prunen heeft zowel voor- als nadelen. Het voordeel, en ook de reden waarom dit algoritme geïmplementeerd is, is dat er sneller een oplossing gevonden wordt omdat er aanzienlijk minder kinderen doorlopen hoeven te worden. Het andere voordeel is dat er beduidend minder geheugen gebruikt wordt omdat telkens de helft van de queue wordt leeggehaald. Het grote nadeel is echter dat de kans bestaat dat in de reeks weggegooide kinderen de oplossing zit met het minst mogelijk aantal zetten. Door dit algoritme kunnen grotere en moeilijkere borden opgelost worden omdat het geheugen dus beperkt wordt.

2.6 Visualisatie

Voor de BFS, het A* algoritme en het algoritme met tussentijdse pruning wordt het pad van begin tot eind opgeslagen. Dit wordt uitgeschreven in een bestand

wat uitgelezen kan worden door een programma wat dan op dit pad de visualisatie kan uitvoeren. In het pad worden de borden met de Auto's opgeslagen als rijen van cijfers, waarin de cijfers de id van de auto representeren. Een programma Pathmaker.py leest deze cijfers uit en zet ze weer om in 2 dimensionale numpy arrays per bord, die vervolgens meegegeven worden aan het visualisatie programma. De visualisatie zelf maakt gebruik van Tkinter om een canvas te maken, daar het bord in te tekenen en de auto's op te tekenen. Dit gebeurd door voor alle borden in het pad over alle posities in het bord te lopen en de vakjes waar een auto op staat in te kleuren met een kleur uit het kleurenschema wat in het visualisatie programma staat. Door alle borden achter elkaar af te spelen kan dus de visualisatie van het pad getoond worden.

3 Resultaten

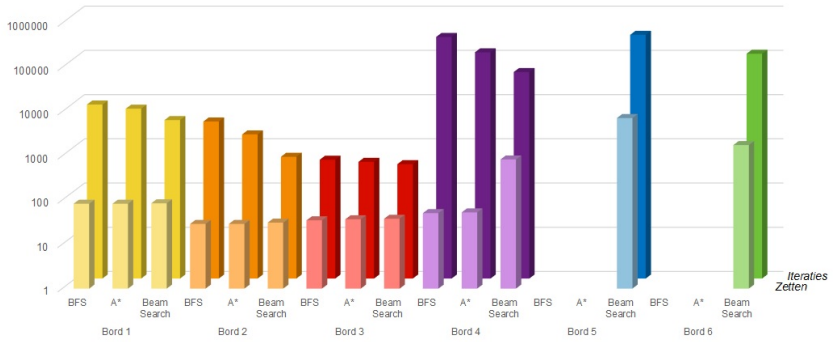
Er zijn geen resultaten behaald met het simpele algoritme. Met het BFS algoritme en A* algoritme zijn voor de eerste vier borden resultaten behaald, maar bij de borden vijf tot en met zeven werd er geen oplossing gevonden. Met het Beam Search algoritme is er voor bord vijf en zes nog een resultaat behaald. De oplossing van bord zeven is niet gevonden. In tabel 3 staan het aantal zetten en iteraties per algoritme per bord en in figuur 4 is een visuele weergave van de resultaten uit tabel 3 te zien.

Uit tabel 3 is af te lezen dat het Beam Search algoritme meer zetten nodig heeft om tot een oplossing te komen dan het BFS algoritme en A* algoritme. Echter is het aantal iteraties dat het Beam Search algoritme nodig heeft aanzienlijk lager dan het aantal iteraties bij zowel het BFS algoritme als het A* algoritme.

Afhankelijk van welk bord getest werd, was het aantal zetten bij het A* algoritme gelijk aan of iets hoger dan bij het BFS algoritme. Hier was echter ook een verbetering te zien in het aantal iteraties dat het algoritme doorliep.

Tabel 3: Het aantal zetten en iteraties per algoritme per bord

Bord (dimensie)		Zetten	Iteraties
Bord 1 (6)	BFS	83	8600
	A*	83	6927
	Beam	115	3830
Bord 2 (6)	BFS	29	3551
	A*	29	1816
	Beam	31	567
Bord 3 (6)	BFS	35	486
	A*	37	435
	Beam	38	386
Bord 4 (9)	BFS	51	290406
	A*	53	130287
	Beam	832	46661
Bord 5 (9)	BFS	-	-
	A*	-	-
	Beam	7195	324411
Bord 6 (9)	BFS	-	-
	A*	-	-
	Beam	1795	120935
Bord 7 (12)	BFS	-	-
	A*	-	-
	Beam	-	-



Figuur 4: Het aantal zetten en iteraties per algoritme per bord.[2]

4 Conclusie

Tijdens het runnen van het eerste algoritme vielen een aantal dingen op, voornamelijk dat een bord met minder voertuigen aanzienlijk meer zetten vereist om tot een oplossing te komen dan een bord met meer voertuigen.
Dit suggereert dat er een optimum aantal auto's is wat voor de moeilijkheid van een bord zorgt. Moeilijkheid van een bord wordt gekwantificeerd met het minste aantal zetten die moet plaatsvinden om tot een oplossing te komen.
Een laag aantal voertuigen zorgt voor weinig blokkades en dus een snel te bevrijden weg. Een hoog aantal voertuigen zorgt voor veel blokkades van de te bevrijden weg, wat het bord moeilijk maakt. Maar door een hoog aantal voertuigen zijn er ook minder mogelijke legale zetten, wat het juist weer makkelijker maakt.
Uit de resultaten is te concluderen dat het A* algoritme en het Beam Search algoritme niet per se de kortste oplossing vinden, aangezien uit de resultaten blijkt dat er met die algoritmes regelmatig oplossingen komen met meer zetten dan het BFS algoritme, wat gegarandeerd altijd de kortste sequentie van zetten tot een oplossing vindt. Echter zijn het A* algoritme en het Beam Search algoritme aanzienlijk sneller wat betreft het aantal iteraties dan het BFS algoritme. Het A* algoritme is in staat een antwoord met het laagste mogelijke aantal zetten te vinden. Dit is echter niet in alle gevallen gelukt, wegens het snelle groeien van de queue wat uiteindelijk te veel geheugen in beslag neemt voor een reguliere computer. Ook valt dit te wijten aan de kostenfunctie, waar de kosten te arbitrair gekozen zijn. De verhouding tussen de kosten zijn middels trial-and-error verbeterd, maar de daadwerkelijke waarden zijn magic numbers. De enige 'admissible' heuristiek van de gebruikte A* kostenfunctie is de kosten die verbonden zijn aan elke zet.
Een nadeel van de Beam Search is dat, omdat je de hoogste kosten weggooit, het moeilijker is om over een lokaal maximum te komen, waardoor het algoritme makkelijker in een lokaal minimum terecht komt. Dit zou verklaard kunnen worden door het feit dat het Beam Search algoritme verder bouwt op het A* algoritme en dus niet de optimale heuristieken gebruikt voor de kostenfunctie. Dit maakt dat het niet te garanderen is dat het laagste aantal zetten gevonden wordt met dit algoritme.
De laatste 3 gebruikte algoritmes zijn goed in het vinden van oplossingen binnen een kleine toestandsruimtegrootte. Maar al snel zijn de algoritmes niet toereikend meer, omdat ze veel geheugen vereisen van de computer waarop ze uitgevoerd worden.

4.0.1 Suggesties vervolgonderzoek

De huidige kostenfunctie past een opstapeling van kosten op een absolute manier toe middels optelling van verschillende kosten. De vraag is echter of het niet beter is om elke korting die van toepassing is op eenzelfde zet te vermenigvuldigen met elkaar, wat zou leiden tot een relatief verband tussen te verschillende kosten.

De kostenfuncties die in het A* algoritme en in Beam search gebruikt worden magic numbers, dit betekend dat de getallen arbitrair gekozen zijn. Dit is niet optimaal en zou verbeterd kunnen worden door ze allemaal op verschillende manieren aan te passen om zo een optimum te vinden tussen de verschillende heuristieken. In dit proces zouden de heuristieken ook eerst per stuk bekeken kunnen worden.
Daarnaast zouden er nog andere, in dit verslag niet gebruikte, heuristieken toegepast kunnen worden, zoals:

- Een beweging alleen korting geven wanneer die een vergroting van de bewegingsvrijheid van een andere auto teweegbrengt.
- De rode auto als startpunt gebruiken waarvan je de omliggende ruimte probeert vrij te maken. Dus een auto aangrenzend aan de rode auto bewegen, als de volgende blokkerende auto bewegen etc.

Of deze aanpassingen inderdaad een beter algoritme opleveren zou in een vervolgonderzoek bekeken kunnen worden.

5 Referenties

Referenties

[1] Breadth-firstTree. https://commons.wikimedia.org/wiki/File:Breadth-first$_t ree.svg$.
$Accessed : 20 - 12 - 2016$.

[2] Rushhour12x12. http://heuristieken.nl/wiki/index.php?title=Rush$_H our$.
$Accessed : 22 - 11 - 2016$.