Implementatie en analyse van het A* algoritme bij het 8-puzzel probleem

Tom Van Eyck - r0636114

22 december 2016

Samenvatting

In dit verslag wort het A* algoritme, geïmplementeerd om het 8-puzzel probleem op te lossen, beschreven. De nadruk ligt op de analyse van dit algoritme: mogelijke optimalisaties, tijdscomplexiteit en dergelijke met enkele concrete resultaten voor een paar voorbeeldpuzzels.

Inhoudsopgave

Lijst van tabellen

1 Inleiding

Het 8-puzzel[1] probleem is een puzzel die al sinds lange tijd bestaat en de inspiratie is geweest voor miljoenen andere schuifpuzzels. Iemand die deze puzzel wil oplossen probeert dat zo snel mogelijk te doen, maar dat lukt niet altijd. Als men de kortste "weg"naar de oplossing dan probeert te zoeken, staat men al snel terug stil en vraagt men zich af: hoe begin ik hier aan?

Dat is waar algoritmes die de kortste weg vinden van pas komen. Een van de bekendste en vaak gebruikte, is A* (A star). Dit algoritme werkt met heuristische waardes die men toekent aan stappen om het benodigde werk om die stap uit te voeren weer te geven. Dit algoritme moet maar een klein beetje worden aangepast om gebruikt te worden om een 8-puzzel op te lossen.

2 Methode

Het A*-algoritme werkt door het initiële bord in een prioriteitsrij te stoppen, dan het bord met de laagste prioriteit uit die rij te halen en de buren ervan terug in de rij te stoppen. Het algoritme blijft dit herhalen tot het bord met de laagste prioriteit in de prioriteitsrij de oplossing voor initiële bord is.

De prioriteitsfuncties die hiervoor gebruikt worden zijn de hamming- en manhattanfuncties. De eerste werkt door voor elk getal na te gaan of het op de juiste plaats staat. Zoniet telt hij 1 op bij de som, welke op het einde dus het aantal getallen op de foute plaats weergeeft. De manhattanfunctie werkt door voor alle foute getallen i.p.v. 1 op te tellen bij de som, de afstand volgens geldige verplaatsingen naar de doelpositie op te tellen bij de som. Deze functie bevat dus meer informatie. Bij deze sommen wordt ook nog het aantal stappen nodig om de huidige bordpositie te bereiken bijgeteld. Het bord met de kleinste som komt dan als eerste in de prioriteitsrij te staan.

Om de oplosbaarheid van een bord te berekenen maken we gebruik van permutaties. Als we het lege vakje in de puzzel naar rechtsbeneden verschuiven met geldige verschuivingen, en dan van links naar rechts en van boven naar onder alle getallen achter elkaar zetten, krijgen we een permutatie van de oplossing. Als de permutatie bestaat uit een even aantal verschuivingen, dan kunnen we besluiten dat de puzzel oplosbaar is.[2]

[1] Het 8-puzzel probleem heeft veel varianten, waarvan de meest voorkomende het 15-puzzel- en het 21-puzzelprobleem zijn.

[2] De pariteit van een permutatie wordt soms ook aangeduid met het teken van de permutatie. Is deze positief, dan is de permutatie even en bijgevolg de puzzel oplosbaar.

3 Resultaten

Puzzel	Aantal verplaatsingen	Uitvoeringstijd	
		Hamming	Manhattan
Puzzel 28	28	1,533	0,044
Puzzel 30	30	3,375	0,055
Puzzel 32	32	/	2,170
Puzzel 34	34	/	0,472
Puzzel 36	36	/	4,272
Puzzel 38	38	/	4,842
Puzzel 40	40	/	2,245
Puzzel 42	42	/	18,950

Tabel 1: Aantal verplaatsingen om de uitkomst te bekomen met de uitvoeringstijd van het algoritme voor zowel de hamming- als manhattanvariant.[3]

4 Bespreking

4.1 De complexiteit van de prioriteitsfuncties

De complexiteit van de prioriteitsfuncties is makkelijk te vinden: De prioriteitsfunctie moet voor elk vakje beslissen of het al dan niet verkeerd staat in de puzzel (en voor de manhattanprioriteitsfunctie de afstand tot de juiste positie berekenen).
Het algoritme moet dus, voor een bord met de grootte N, N x N arrayelementen opvragen. Dit is gelijk aan een tijdscomplexiteit van $\sim N^2$.

4.2 Controle op oplosbaarheid

Het bord wordt gecontroleerd door gebruik te maken van het teken van permutaties. In deze implementatie wordt eerst het lege vakje met geldige verschuivingen naar de rechterbenedenhoek verschoven. Hierna wordt een voor een, van links naar rechts, over alle getallen gelopen en slaat het algoritme de locatie voor dat getal in het bord op in een rij.

Dan wordt elk getal vanaf 1 tot en met het grootste getal ingevuld in de volgende formule:

$$sgn(p) = \frac{\prod_{i<j} p(j) - p(i)}{\prod_{i<j} j - i} \tag{4.1}$$

Dit berekent het teken van de permutatie. Is dit teken positief, dan is de permutatie even en is bijgevolg de puzzel op te lossen.

[3] Vanaf puzzel 32 werd er voor de hammingvariant `java.lang.OutOfMemoryError` gegooid, waardoor de puzzel niet opgelost kon worden.

De complexiteit van dit algoritme bestaat uit verschillende delen:

- *De lege plaats zoeken.* Hiervoor moet over alle elementen in het bord gelopen worden tot de lege plaats gevonden is. Voor een bord met grootte N moeten er gemiddeld $\frac{N^2}{2}$ elementen bekeken worden.
- *De lege plaats verschuiven.* Dit geeft in het slechtste geval (de lege plaats in de linkerbovenhoek) $N-1$ verschuivingen naar beneden en $N-1$ verschuivingen naar rechts. In het beste geval staat de lege plaats al juist, en moet er dus niet verschoven worden. Dit komt neer op een gemiddelde van $N-1$ aantal verschuivingen, met 3 aantal rijtoegangen per verschuiving. Zo krijgen we $3N-3$ aantal rijtoegangen in totaal.
- *Het opslaan van de plaats voor elk element.* Het hele bord wordt hiervoor afgelopen, en de locatie wordt voor elk element (behalve het lege) in een nieuwe rij gestoken. Dit geeft een totaal van $2N^2-1$ aantal rijtoegangen.
- *De formule invullen.* Het aanal keer dat de formule ingevuld kan worden (met $i < j$) is gelijk aan $(N-2)+(N-3)+\cdots+2+1$, oftewel $\frac{N(N-1)}{2}$. Per keer dat de formule wordt ingevuld, wordt er twee keer een element opgehaald uit de rij met posities. In totaal zijn er voor de formule dus $N(N-1)$ aantal rijtoegangen nodig.

Om de totale tijdscomplexiteit van dit algoritme te berekenen, volstaat het om al de voorgenoemde delen op te tellen:

$$\frac{N^2}{2} + 3N - 3 + 2N^2 - 1 + N(N-1) = 3N^2 + \frac{N^2}{2} + 2N - 4 \tag{4.2a}$$

$$\Updownarrow \tag{4.2b}$$

$$\sim 3N^2 \tag{4.2c}$$

4.3 Geheugengebruik

Om de oplossing te berekenen wordt er gebruikt gemaakt van een queue waarin borden worden opgeslagen. Het aantal geheugen dat dit inneemt loopt natuurlijk heel snel op, dus is het belangrijk om te weten hoeveel borden er worst case in het geheugen zitten zodat we optimalisaties kunnen toepassen waar nodig.

Voor een puzzel met grootte N zijn er vanzelfsprekend $N^2!$ aantal bordtoestanden. Natuurlijk zijn niet al die toestanden oplosbaar. Door het feit dat de permutaties even of oneven zijn (hetzelfde als het teken, zie sectie 4.2), en hiermee de oplosbaarheid bepaald kan worden, is af te leiden dat de helft van alle mogelijke toestanden oplosbaar zijn. Bijgevolg kunnen er in het slechtste geval maximum $\frac{N^2!}{2}$ aantal borden aanwezig zijn in de prioriteitsrij.

4.4 Prioriteitsfuncties

De prioriteitsfunctie bij een algoritme zoals hier gebriukt wordt is een van de belangrijkste aspecten van het algoritme. Deze functie bepaalt namelijk welke stappen minder werk vergen en welke stappen het snelst leiden naar de oplossing.

De hiergebruikte prioriteitsfuncties (hamming en manhattan) zijn goed genoeg voor de meeste puzzels, maar soms gaan ze niet snel genoeg. Een verbetering van de manhattanfunctie heet *linear conflict*. Deze optimalisatie voegt bij de manhattanfunctie 2 toe aan de som als er twee elementen al in hun doelrij/-kolom staan, maar verwisseld zijn t.o.v. hun doelpositie.

Bij een tweede verbetering maakt men gebruik van een patroondatabase. Hier gaat men voor bepaalde patronen op voorhand berekenen hoeveel zetten er nodig zijn om tot de oplossing te komen en deze opslaan in een database. Hierdoor neemt het minder tijd in beslag om de optimale volgende stap te berekenen.

4.5 Grotere puzzels

Wanneer men grotere puzzels wilt oplossen (bijvoorbeeld met een grootte van 4 of 5), is het logisch dat er iets moet veranderen aan het algoritme. Voor grotere puzzels is het mogelijk om meer tijd te voorzien, maar het geheugen, dat nog steeds beperkt is, gaat dat niet toelaten. We kunnen i.p.v. meer tijd meer geheugen voorzien, maar daar gaat de tijd weer een obstakel vormen.

Een mogelijkheid zou zijn om zowel meer tijd als geheugen toe te laten. Dit is wel degelijk een oplossing voor puzzels met beperkte grootte. Willen we echter puzzels oplossen die veel groter zijn, dan zitten we al snel aan uren uitvoeringstijd en het is niet zo moeilijk om te bedenken dat bij zeer grote puzzels de uitvoeringstijd naar de jaren gaat. In dat laatste geval gaat het ook fysisch onmogelijk zijn om genoeg werkgeheugen te voorzien.

Om toch in een redelijke tijd tot een oplossing te komen is het efficiënter om het algoritme te optimaliseren. Hiervoor kan men de methodes uit de vorige paragraaf gebruiken, maar er bestaan nog tal wan andere optimalisaties zoals het weigeren van al geziene borden in de prioriteitsrij, het cachen van de hamming en manhattan waardes enz.

Nu kunnen we ons de vraag stellen: is dat zeer efficiënte algoritme uit de vorige paragraaf in staat zeer grote puzzels in een redeijke tijd op te lossen? Aangezien er altijd minstens een keer over alle elementen in elk bord gelopen moet worden, en het aantal elementen per bord een grootteorde van N^2 hebben, krijg je voor grote N een NP-probleem, wat op dit moment nog niet praktisch op te lossen is.