GNA practicum 2

Jeff Mathieu - r0853061

April 30, 2021

Contents

1 Vraag 1

Puzzle	Aantal verplaatsingen	Hamming (time)	Manhattan (time)
puzzle28.txt	28	5m1s	4s
puzzle30.txt	30	$\gg$5m	13s
puzzle32.txt	32	$\gg$5m	3m23s
puzzle34.txt	34	$\gg$5m	2m12s
puzzle36.txt	/	$\gg$5m	$\gg$5m
puzzle38.txt	/	$\gg$5m	$\gg$5m
puzzle40.txt	/	$\gg$5m	$\gg$5m
puzzle42.txt	/	$\gg$5m	$\gg$5m

2 Vraag 2

Wat is de complexiteit (aantal array accesses) van de prioriteitsfuncties hamming en manhattan, in functie van de bord grootte N?
Om deze vraag te beantwoorden gaan we eerst berekenen wat we normaal gezien zouden moeten uitkomen. Daarna gaan we een aantal testen uitvoeren om te zien of onze theoretische waarden kloppen. We zullen eerst natuurlijk moeten kijken naar hoe deze algoritmen nu juist te werk gaan.

2.1 Hoe gaan de algoritmen te werk?

Hamming

Het Hamming algoritme kijkt naar hoeveel tegels van het bord zich in de verkeerde positie bevinden. Om dit te kunnen zien geven we elk vakje een nummer. We starten links boven met 1 en gaan verder naar rechts met twee en zo verder zoals in het volgende voorbeeld:

1 2 3
4 5 6
7 8 0

Het laatste getal, waar de 0 staat, tellen we niet mee omdat bij het tellen van de verkeerd staande tegels tellen we tegel 0 niet mee.
Nu we elke tegel een nummer gegeven hebben die overeenkomt met de tegel die er zou moeten staan kunnen we zien of een tegel juist staat of niet. Het algoritme zal nu over elke tegel gaan en kijken of de tegel die er staat de

juiste nummer heeft.

Manhattan

Het Manhattan algoritme gaat kijken naar hoe ver een tegel van zijn correcte plaats verwijdert is. Dit doet hij voor elke tegel van het bord en telt deze waardes dan bij elkaar op. Om dit te berekenen kijken we naar de x-coördinaat en de y-coördinaat van de tegel en van de plaats waar de tegel naartoe moet. Dan nemen we de absolute waarde van het verschil tussen de x- en y-coördinaten van bijde tegels.

$$Manhattan = |x_{juistetegel} - x_{tegel}| + |y_{juistetegel} - y_{tegel}|$$

2.2 Theoretische complexiteit

Hamming

Als we naar de code kijken van dit algoritme zien we dat voor elk vakje dat bekeken wordt er 2 keer naar een element in de array gekeken wordt. Dit gebeurt omdat we 1 keer moeten controleren of het getal op de juiste plaats staat en 1 keer moeten kijken of het getal op die plaats in het bord geen 0 is. We kijken of het geen 0 is omdat we geen rekening moeten houden met het lege vakje, wat voorgesteld wordt door een vakje met nummer 0. Als we terug vergelijken met de code wat dit wilt zeggen is, hoe vaak wordt het binnenste van de binnenste for-loop uitgevoerd? Voor een eerste iteratie van de eerste for loop zal de binnenste for-loop N keer uitgevoerd worden. Aangezien de eerste for-loop ook N keer uitgevoerd zal worden zal wat binnen de binnenste for-loop staat $N * N$ keer uitgevoerd worden. Omdat we binnen de binnenste for-loop 2 keer in de array gaan kijken zal er $(N*N)*2 = 2N^2$ keer in de array gekeken worden.Als we nu bijvoorbeeld een bord met grootte 3 nemen kunnen we onze theoretisch aantal vergelijken controleren: $(3 * 3) * 2 = 9 * 2 = 18$. Na dit te testen zagen we dat dit inderdaad de juiste formule is aangezien ze het juiste resultaat teruggeeft. Als we deze formule in $\sim$-notatie zetten krijgen we $\sim N^2$

Manhattan

Ook hier zullen we eerst naar de code moeten kijken om de complexiteit te kunnen berekenen. We zien hier weer 2 for-loops in elkaar waarbij ongeveer hetzelfde gebeurt als bij het Hamming algoritme. Er zal dus weer 2 keer in de array gekeken moeten worden. 1 keer voor de nummer van het vakje te weten te komen en 1 keer om te kijken of het vakje niet het lege vakje is

(het 0-vakje). Doordat dit hetzelfde is zullen we dezelfde formule als daarnet uitkomen waardoor hier de complexiteit ook $\sim N^2$ is.

3 Vraag 3

Leg uit hoe je isSolvable hebt geïmplementeerd. Wat is de complexiteit (aantal array accesses) in functie van de bord grootte N?

3.1 Implementatie

In plaats van de gegeven formule te gebruiken heb ik op een andere manier kunnen zien of een bord oplosbaar was of niet. Dit door het aantal paren die fout staan op te tellen. Wanneer dat getal even is is het bord oplosbaar, als het oneven is is het bord niet oplosbaar. Wat bedoel ik nu precies met dat aantal paren die fout staan? We nemen volgende puzzel als voorbeeld.

7	8	5
4		2
3	6	1

Om het makkelijker te maken zetten we het bord op een rij en laten we het lege vakje eruit. Dan krijgen we *7, 8, 5, 4, 2, 3, 6, 1.* We starten met het eerste getal in de rij, namelijk *7*, we vergelijken dit met alle getallen die erachter komen. Wanneer we een getal tegenkomen dat kleiner is dan *7* noemen we dit een *inversie*. Als we dit doen voor elk getal in de rij komen we een totaal aantal inversies uit. Als dit getal deelbaar is door 2 dan weten we zeker dat deze puzzel opgelost kan worden. Indien niet deelbaar door 2 weten we dat de puzzel niet oplosbaar is.[1]

3.2 Complexiteit

Om te starten zetten we dus alle tegels van het bord in een rij achter mekaar. Hiervoor zal dus al $N*N$ keer in de array gekeken moeten worden. Dit weten we door de dubbele for-loop die we in vorige vraag al bekeken hebben en dat we hier maar 1 keer naar de array kijken binnen de binnenste for-loop.
Voor het tweede deel van de berekening kijken we naar het aantal inversies. Dit gebeurt ongeveer hetzelfde als selection sort uit vorig practicum. Bij de eerste iteratie van de eerste for-loop zullen er dus $N-1$ vergelijkingen gebeuren tussen 2 elementen van de rij. Daardoor zal er dus per vergelijking

[1]COLLIER,A. (2019, 4 oktober). Sliding Puzzle Solvable?. Datawookie Blog. https://datawookie.dev/blog/2019/04/sliding-puzzle-solvable/

2 keer in de nieuwe array gekeken moeten worden. Bij een tweede iteratie zullen er $N-2$ vergelijkingen gebeuren en ook weer 2 keer zo veel array accesses. En zo verder en zo verder. Uiteindelijk komen we zoals bij selection sort op $\frac{N(N-1)}{2}$ vergelijkingen. Dit doen we nu ook maal 2 aangezien er per vergelijking 2 array accesses gebeuren, wat dus op een complexiteit van $2 * \frac{N(N-1)}{2} = N(N-1)$.
Nu we beide complexiteiten hebben moeten we deze optellen om de volledige complexiteit van het algoritme te vinden. $N^2 + N^2 - N = 2N^2 - N$ wat neerkomt op een complexiteit van $\sim 2N^2$.

4 Vraag 4

Hoeveel bordposities zitten er worst-case in het geheugen, in functie van de bord grootte N?
Moest er geen limiet staan op het geheugen zou je bijvoorbeeld een boomstructuur kunnen maken waarin de buren van de buren ... van de initiële bordtoestand opgeslagen zitten. Voor een bord van grootte 3 zullen er bijvoorbeeld $(3*3)!/2 = 9!/2 = 362,880/2 = 181,440$[2] mogelijke bordstructuren zijn. Aangezien we in de boomstructuur alle mogelijke buren berekenen is de kans groot dat praktisch alle mogelijke bordconfiguraties opgeslagen worden in de boomstructuur. Dit is natuurlijk zeer onpraktisch en dus ook de reden dat we geen boomstructuur gebruiken om de puzzels op te lossen.
Als we de meest onpraktische bordconfiguratie opzoeken komen we deze puzzel uit:

8	6	7		6	4	7
2	5	4	of:	8	5	
3		1		3	2	1

Bij deze puzzels zijn er een maximumaantal van 31 moves nodig om tot de eindtoestand te komen. Hierdoor zullen er dus 32 bordposities in de finale rij zitten. Aangezien bij de neighbor rij steeds geleegd wordt zullen we die niet meetellen. Hoe kunnen we dit nu omzetten naar een formule voor een bord met grootte N? We willen dus weten wat de slechtste start positie is. Dit wil zeggen de start positie die het meeste moves nodig heeft om opgelost te worden. Nog een ander voorbeeld is een bord met grootte 15, deze kan altijd opgelost worden in minder dan 81 moves (maximum 80 moves dus).

[2] http://w01fe.com/blog/2009/01/the-hardest-eight-puzzle-instances-take-31-moves-to-solve/

Hierbij zullen er dus maximum 81 borden in het geheugen zitten. We komen aan deze resultaten door te kijken naar het aantal mogelijke borden ($N!/2$) en zo elk bord te evalueren uit de startpositie. Uiteindelijk zal je dus een maximum aantal moves vinden. Dit zal steeds langer en langer beginnen duren, wat logisch is als je naar de formule kijkt.

5 Vraag 5

Wat is bij jouw oplossing de worst-case tijdscomplexiteit om een bordconfiguratie toe te voegen aan de prioriteitsrij? En om een bordconfiguratie uit de prioriteitsrij te nemen?
Voor het toevoegen van een bord aan de prioriteitsrij zal elk element van de rij in een nieuwe rij gestoken worden. Dit zal neerkomen op een complexiteit van $\sim n\log(n)$. Om een element van de prioriteitsrij te halen zoeken we eerst naar het bord met de kleinste hamming of manhattan waarde. Dut zak iij neerkomen op een complexiteit van $\sim n\log(n)$.[3]

6 Vraag 6

Zijn er betere prioriteitsfunctie(s)?
Uit een artikel van MIT[4] blijkt dat wanneer we een puzzel willen oplossen we steeds de manhattan prioriteitsfunctie als optionele manier uitkomen. Er wordt gezegd dat de tijd die nodig is om de puzzel op te lossen enkel kleiner kan worden door een efficiëntere manhattan functie te gebruiken. Hoe gesofisticeerder de manhattan functie hoe sneller je je resultaat hebt. Dus om op de vraag te antwoorden: nee, er bestaan (momenteel) geen betere prioriteitsfuncties. Ik zeg hier momenteel omdat voorlopig er nog geen betere manier gevonden is om de prioriteit van een bord te berekenen. Dit probleem is ook een *millenium probleem*, dit is een van de 7 problemen die begin 2000 opgestelt zijn. Wanneer je er een zou kunnen oplossen word je beloond met *1 miljoen euro*. Dit probleem wordt ook wel het *P vs NP* probleem genoemd.

[3]https://stackoverflow.com/questions/46627634/java-priorityqueue-time-complexity-of-this-code/46627745

[4]KUNKLE,D.R. (2001, 8 oktober). Solving the 8 Puzzle in a Minimum Number of Moves: An Application of the A* Algorithm. https://web.mit.edu/6.034/wwwbob/EightPuzzle.pdf

Hiermee wordt bedoeld dat het makkelijk is om een oplossing te vinden maar hoe vinden we de kortste oplossing. Vandaar dat er momenteel nog geen betere prioriteitsfunctie is dan de manhattan functie die we nu gebruiken.

7 Vraag 7

Indien je willekeurige 4×4 of 5×5 puzzels zou willen oplossen, wat zou je verkiezen: meer toegelaten tijd (bijvoorbeeld 10× zo lang), meer geheugen (bijvoorbeeld 10× zo veel), of een betere prioriteitsfunctie? Waarom?
Ik zou liever voor meer geheugen gaan. Tijdens het maken van het algoritme zat ik heel lang vast bij de solver. Dit kwam doordat elke keer hij een puzzel wou oplossen hij telkens te weinig geheugen had. Vandaar zou ik liever meer geheugen gebuiken waardoor er makkelijker een bijna volledige boomstructuur te creeëren om zo een oplossing te vinden.
Een betere prioriteitsfunctie zou natuulijk ook zeer handig zijn. Want door een betere prioriteitsfunctie te gebruiken zal de tijd die nodig is en de plaats die nodig is verminderen. Dit zal ook het vorige probleem oplossen. Maar zoals we in vorige vraag besproken hebben bestaat er momenteel nog geen betere prioriteitsfunctie.
Meer oplossingstijd zou ik eerder niet kiezen aangezien we moeten zoeken naar een manier om algoritmen juist sneller te laten werken in plaats van trager. Natuurlijk in sommige gevallen is het belangrijker om minder geheugen te gebruiken dan een sneller algoritme te maken dat meer geheugen gebruikt. Maar in dit geval is het niet erg om langer te wachten om een oplossing te vinden.

8 Vraag 8

Denk je dat er een efficiënt algoritme bestaat wat, zelfs voor grote puzzels, de optimale oplossing binnen praktische tijd kan vinden?
Moest er een manier zijn om een volledige boomstructuur te maken, hoe groot hij ook zal moeten zijn, zonder dat er veel tijd en geheugen voor nodig is zou dit een betere manier zijn denk ik. Zo zouden we naar de eind-bladeren van de boom kijken en controleren of dit de eindtoestand van de puzzel is. Als dit is nemen we het pad naar boven om zo de oplossing van te puzzel

te hebben. Stel er zijn meerdere juiste eindtoestanden nemen we de kortste weg naar boven, wat de kortste oplossing is. Het probleem hiermee is dat zo een grote boomstructuur opstellen zeer veel tijd vraagt. Hierdoor zal het dus waarschijnlijk niet sneller zijn dan de methode die we nu gebruiken. Daarom denk ik dat er geen efficiënter algoritme bestaat om een puzzel op te lossen. Dit is ook weer hetzelfde NP probleem als in vraag 6.