GEGEVENSSTRUCTUREN EN ALGORITMEN: A*-ALGORITME

Liam Volckerick

APRIL 28, 2018

KULEUVEN

Inhoudstafel

Inleiding

In dit practicum werden we geïntroduceerd tot het A*-algoritme en zo ook enkele prioriteitsfuncties. De nadruk van de gemaakte code werd gelegd op het kunnen oplossen van een schuifpuzzel met NxN als dimensie.

A*-implementatie

A*-algoritme is een algoritme dat het kortste pad kan berekenen. Dit kwam zeer handig uit binnen mijn implementatie van het 8-puzzel spel. A* maakt gebruik van een bepaalde functiewaarde 'f'. Laten we zeggen dat 'f' bestaat uit de som van 2 andere elementen, namelijk 'i' en 'j'. 'i' stelt de bewegingskost voor, dat wil zeggen dat 'i' het aantal gemaakte bewegingen bijhoudt. Laten we 'j' nu gelijk stellen aan een benadering van de af te leggen afstand van begin- tot eindpositie. Indien we deze exact zouden berekenen, zal de tijdscomplexiteit echter enorm toenemen aangezien deze uit veel berekeningen bestaat. Daarom hanteer ik in dit practicum een benadering tot 'j' m.b.v. de Hamming en Manhattan prioriteitsfuncties. Elke keer dat mijn A*-algoritme een buurpositie ingaat, hoort onze functiewaarde 'f' in optimale omstandigheden te dalen. Elke buurpositie zal worden geplaatst in een priorityqueue. Dusdanig zal ik een pad creëren van begin- tot eindpositie op een optimale manier. Indien de meest optimale buurpositie niet leidt tot de gewenste eindpositie, zal het algoritme 1 stap terugkeren en de tweede meest gunstige buurpositie aannemen van die vorige stap. Dit zal zich herhalen totdat het optimale pad is gevonden.

In onderstaande tabel kan je enkele metingen aflezen voor enkele bordspellen die zijn opgelost door het A*-algoritme m.b.v. een prioriteitsfunctie. Elke uitvoering gebeurde aan de hand van een 1GiB geheugen. OOM staat voor 'Out Of Memory'. De hoeveelheid borden die werden opgeslagen in de priorityqueue overschreed de limiet van 1GiB geheugen.

Puzzel	**Moves**	**Hamming(s)**	**Manhattan(s)**
28	28	1,107	0,000
30	30	2,734	0,028
32	32	OOM	0,878
34	34	OOM	0,316
36	36	OOM	4,563
38	38	OOM	3,789
40	40	OOM	0,765
42	42	OOM	2,262

Hamming prioriteitsfunctie

Hamming is een prioriteitsfunctie die alle gegeven posities analyseert en een som teruggeeft van alle tegels die zich nog niet op de juiste positie bevinden. Hierbij tel ik het aantal moves erbij op om vanuit initiële toestand de gewenste eindtoestand te bereiken. Hieruit kan ik afleiden dat een bepaalde toestand van het bord, met een lage Hamming kost, dichter gelegen ligt bij de gewenste eindtoestand en deze dus verkozen zal worden als volgende stap. In mijn implementatie, onderzoek ik elke tegel van mijn gegeven bord met dimensie NxN m.b.v. twee for-loops. De eerste itereert over de rijen, de tweede over de kolommen. Wat dus zal uitwijzen op tilde N^2 array accesses omwille van de inner- en outerloop die beiden tot N itereren.

Manhattan prioriteitsfunctie

Manhattan prioriteitsfunctie is een betere benadering ten opzichte van de Hamming prioriteitsfunctie. Deze berekent de som van de afstanden van elke tegel tot hun gewenste eindpositie plus het aantal gemaakte moves tot de huidige staat. Ook deze implementatie is gemaakt met behulp van twee for-loops. De inner- en outerloop itereren beide over de lengte van het bord. Hierdoor zal er steeds N-toegangen zijn gepleegd bij elke iteratie van de outerloop. Dit leidt tot een tilde N^2 array accesses.

Oplossen van een Puzzel

Vooraleer we een puzzel kunnen oplossen hoor ik na te gaan indien deze puzzel wel degelijk oplosbaar is. Gegeven formule: $oplosbaarheid(b) = \prod_{n=1}^{\infty}(p(b,j) - p(b,i) \geq 0$. De elementen i en j itereren over de bordelementen, exclusief de lege ruimte aangeduid met een 0. Indien er zich een groter getal bevindt voor een gegeven tegel, zal dit product veranderen van teken. Zo ook wanneer er een kleiner getal wordt gevonden achter deze tegel. Dus wanneer de hoeveelheid kleinere getallen achter de gegeven tegel en de hoeveelheid grotere getallen voor de gegeven tegel even zijn, dan zal dit product een positief getal uitkomen. Wanneer ze in oneven aantallen voorkomen zal dit product negatief worden. In de oorspronkelijk gegeven formule, deelde ik bovenstaande formule nog met $\prod_{i<j(}j - i)$. Deze complete formule, met teller en noemer, heb ik in mijn programma echter wel gehanteerd, al zou mijn eerst vermelde formule ook hetzelfde resultaat teruggegeven hebben. Die formule, met noemer, is zo gemaakt om overflow te vermijden, aangezien deze getalswaarde enorm groot kan worden. De formule werkt alleen wanneer de lege positie, aangeduid door '0', zich helemaal rechtsonder begeeft in het speelvlak. Om overflow te voorkomen van integers, gebruik ik BigIntegers als teller en noemer.

```
public boolean isSolvable() {
    Board board = getZeroConfiguredBoard(this); //Nkwad
    BigInteger numerator = BigInteger.valueOf(1);
    BigInteger denominator = BigInteger.valueOf(1);
    int i;
    int j;
    for (j = 1; j < board.boardSize * board.tiles[0].length; j++) {////((NxN-1)*(NxN))/2
        for (i = j-1; i > 0; i--) {
            BigInteger numMultiPl = BigInteger.valueOf((getTile(j, board) - getTile(i, board)));
            BigInteger denomMultiPl = BigInteger.valueOf(j-i);

            numerator = numerator.multiply(numMultiPl);
            denominator = denominator.multiply(denomMultiPl);
        }
    }
    System.out.println((numerator.divide(denominator)).intValue());
    return ((numerator.divide(denominator)).intValue() >= 0);
}
```

Figuur 1: isSolvable

Voordat ik de formule op oplosbaarheid bereken, hoor ik de lege positie rechts onderaan in het spelbord te verschuiven. De methode zeroConfiguredBoard geeft me een spelbord terug waar de lege ruimte zich op haar doelpositie bevindt. Hier door ik weer beroep op twee geneste for-loops die elke index nagaan van het spelbord. Ik zoek hiermee de indices waar de waarde ervan 0 teruggeeft. De worst case complexiteit van deze hulpfunctie bedraagt hier N^2. Deze situatie doet zich alleen voor indien de nul positie zich reeds op zijn doelpositie bevindt. Wanneer beide indices gevonden zijn zullen 2 while-loops de nul positie verschuiven. De eerste loop zal het element tot de uiterste rechtse positie verschuiven. De tweede loop zal deze bekomen positie tot de uiterste onderste positie verplaatsen. Waardoor de lege tegel zijn doelpositie heeft bereikt. Wanneer de nul positie zich reeds bevond op zijn doelpositie zullen deze while-loops een kost hebben van 2. Indien de nul positie zich bevindt in de eerste rij en eerste kolom zal de kost voor beide while-loops 4N bedragen, wat nog steeds veel gunstiger is dan worst-case bij de voorgaande geneste for-loops. Worst case zal deze hulpfunctie een kost hebben van N^2 array accesses. In het beste geval is deze 4N+2. Average case van deze hele methode resulteert tot een complexiteit van $(N^2/2)$+2N. Dit is geen tilde notatie.

```
private Board getZeroConfiguredBoard(Board board) {
    int idxRow = 0;
    int idxCol = 0;
    Board tmpBoard = new Board(board.tiles);
    for(int i = 0; i < tmpBoard.boardSize; i++) {//i to N
        for(int j = 0; j < tmpBoard.boardSize; j++) {//jto N
            if(tmpBoard.tiles[i][j] == 0) {
                idxRow = i;
                idxCol = j;
                break;
            }
        }
    }
    while(idxRow != tmpBoard.boardSize-1) {
        //this is not true when we had worst case in first
        //2 forloops. This will be worst case
        //if the first two forloops ended in their first iteration
        //This also holds still with our second while
        //loop underneath for the collumn ordening.
        // Best case: 4 accesses per while loop
        //worst case: 2N accesses per while loop
        tmpBoard.tiles[idxRow][idxCol] = tmpBoard.tiles[idxRow +1][idxCol];
        tmpBoard.tiles[idxRow+1][idxCol] = 0;
        idxRow++;
    }
    while(idxCol != tmpBoard.tiles[0].length - 1) {
        tmpBoard.tiles[idxRow][idxCol] = tmpBoard.tiles[idxRow][idxCol+1];
        tmpBoard.tiles[idxRow][idxCol+1] = 0;
        idxCol++;
    }
    return tmpBoard;
```

Figuur 2: getZeroConfiguredBoard

Nadat we deze tegelpositie hebben berekend, zal het programma terugkeren naar de eerstvolgende lijn in de methode isSolvable. Hier wordt de voorgaande lijn gevolgd door een binnenste en buitenste for-loop. Beide loops itereren over elk element uit het spelbord. Met name {1, 2, 3, ..., 7, 8}. Dit spelbord wordt gerepresenteerd door een 1-dimensionele array. J itereert hier tot (NxN-1) en i zal de waarde aannemen van j-1 tot 1. Zie figuur1 voor de code. Wat er zich dus als complexiteit zal voordoen is een som van 1 + 2 + 3 + ... + (NxN-1) = $\frac{(NxN-1)*(NxN-2)}{2} = \frac{(N^2-1)!}{2(N^2-3)!} = \sim \frac{N^4}{2}$

```
private int getTile(int tileValue, Board board) {
    int i;
    int j;
    for(i = 0; i < board.boardSize; i++) {
        for(j = 0; j < board.tiles[0].length; j++) { //N kwad
            if(board.tiles[i][j] == tileValue)  //Nkwad at most, 1 at best
                return i*board.tiles[0].length+1+j;
        }
    }
    return -1;
}
```

Figuur 3: getTile

Telkens we de positie van de waarden j en i willen berekenen, doet het programma beroep op een method genaamd 'getTile'. Deze gaat de positie na van de gegeven waarde in onze 1-dimensionele array. Dit met een worst case complexiteit van N^2omdat beide for-loops binnen deze method genesteld zijn. Elke for-loop gaat van waarde 0 tot de grootte van het aantal kolommen/rijen waarbij deze respectievelijk gelijk zijn aan N. Average case zal leiden tot $N^2/2$. Aangezien we 2 keer 'getTile' oproepen bekomen we een kost van N^2.

Wanneer we terugkeren nadat we de positie van de gegeven waarden hebben berekend, zullen we teller en noemer vermenigvuldigen met de berekende tussenuitkomsten. In de methode isSolvable zullen we uiteindelijk een boolean terugkeren van de vorm (teller/noemer) ≥ 0. Indien deze waar is, is het gegeven spelbord oplosbaar. De volledige complexiteit van isSolvable, met haar hulpfuncties erbij geteld, bedraagt: $\sim \frac{N^6}{2}$.

Geheugen gebruik

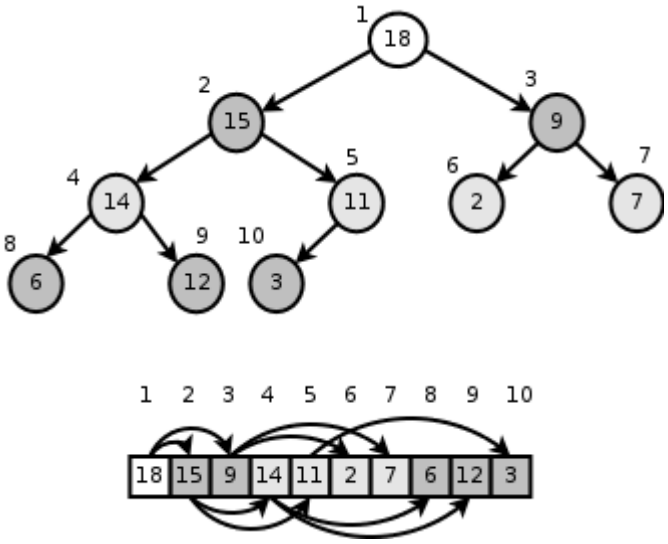
In het slechtste geval zal het A*-algoritme elke puzzel nagaan vooraleer het de oplossing heeft gevonden. Bij een NxN-puzzel zijn er $N^2!$ verschillende bordconfiguraties. Onder deze hele verzameling van alle mogelijke bordconfiguraties is elke bordconfiguratie niet noodzakelijk een geldige permutatie. Een algemeen kenmerk van een oplosbare puzzel is dat het aantal inversies van een NxN-puzzel een even getal blijft, dit ook na elke permutatie. Dit getal is oneven indien de puzzel onmogelijk zou zijn. Sluiten we de oneven mogelijkheden uit, resteert er ons enkel nog $(N^2 + 1)!/2$ andere bordpermutaties vanuit de huidige bordtoestand, door middel van geldige verschuivingen. In het slechtste geval zitter er $\frac{N^2}{2}$ bordconfiguraties in het geheugen.

Priorityqueue

In mijn implementatie maak ik gebruik van de priorityqueue die Java zelf aanbiedt. Deze priorityqueue werkt volgens het principe van First In First Out. Om alles netjes te sorteren binnen deze priorityqueue maak ik gebruik van een comparator. Deze zal de gevormde buren van het huidige bordspel rangschikken volgens de laagste kost. Hierna zullen deze borden worden toegevoegd aan de priorityqueue. Waarna de borden binnen de priorityqueue gerangschikt worden op basis van hun prioriteit. De priorityqueue van Java maakt gebruik van een binary heap-datastructuur. Deze representeert een array waar een bepaalde positie in deze array gegarandeerd groter/gelijk is aan twee andere indices in deze array. Deze twee andere indices zijn respectievelijk ook groter of gelijk aan twee andere indices uit diezelfde array enz. Deze binary heap wordt voorgesteld als een compleet geordende binaire boom onder de vorm van een array. De hoogte van dergelijke binaire boom bedraagt lg(n). N representeert hier het aantal elementen in deze array. Dit is makkelijk aan te tonen doordat de hoogte van de boom verhoogd met 1 telkens N wordt verhoogd tot een macht van 2.

Aangezien in mijn implementatie het berekenen van de heuristische kosten in haast constante tijd gebeurt (Zie optimalisaties), hoor ik hier geen rekening mee te houden wanneer ik deze zou toevoegen aan de priorityqueue. Een element toevoegen zou een complexiteit van 1+lg(n) bedragen. Dit is met uitsluiting van de complexiteit van het KxK-bordspel dat zal worden toegevoegd. Deze operatie op de prioriteitsrij houdt in dat we een pad zullen volgen doorheen de binaire boom. Vertrekkende van de top volgen we een pad tot de bodem van deze boom. Zoals eerder vermeld bedraagt de hoogte van deze binaire boom niet meer dan lg(n). De complexiteit voor het toevoegen van een KxK-bordspel in een n-lange prioriteitsrij bedraagt: $\sim 1 + \lg(n) + k^2$.

Iets verwijderen uit deze prioriteitsrij heeft een complexiteit van $\sim 2\lg(n)$. Dit is te verklaren door dat het verwijderen van een element steeds twee vergelijkingen maakt voor elke knoop op het afgelegde pad in de binaire boom. Eén vergelijking gaat na welke van de kinderen van de huidige knoop de grootste is. De andere vergelijking gaat na of dat grootste kind moet worden gepromoveerd. Dit houdt in of deze knoop hoort te stijgen in de binaire boom.



Figuur 4: Binary Heap

Optimalisaties

Een reeds toegepaste optimalisatie binnen mijn algoritme houdt in dat de Manhattan kost niet steeds wordt berekend telkens het wordt toegevoegd aan de priorityqueue binnen de Solver. We berekenen deze kost eenmalig per object dat zal worden toegevoegd. Dusdanig creëren we een constante complexiteit voor het berekenen van de Manhattan kost. We geven deze kost door als een lokale variabele en geven de opgeslagen waarde terug. Hierdoor is mijn tijdscomplexiteit met een factor 20 gedaald. Hetzelfde gebeurt voor het gebruik van de Hamming prioriteitsfunctie.

Statisch additieve patroon databank

Er bestaan reeds enkele prioriteitsfuncties die gunstiger zouden werken dan Manhattan/Hamming. Zo is een patroon databank een zeer efficiënte manier om het 8-puzzel probleem op te lossen van een NxN-spelbord. Een patroon databank bestaat uit de verzameling van alle patronen die bestaan binnen een geselecteerd deel van het NxN-spelbord en hun heuristische kost, die nodig is om de doelposities te bereiken vanuit elke mogelijke permutatie van de tegels eigen tot die deelverzameling. Anders gezegd, het slaat een verzameling op van kleinere problemen die zullen leiden tot de oplossing van het gevraagde probleem. Het werkt als een soort tabel waar we de heuristische waarde van het huidige bordspel in kunnen opzoeken. Deze databank opmaken is echter wel zeer kostelijk aangezien het elke permutatie van een bepaalde deelgroep zal analyseren en zijn heuristische kost ervan zal berekenen. Een patroon op zichzelf is een partiële specificatie van een staat van een bordspel waar onopgemerkte tegels worden erkend als niets. Een doelpatroon is een partiële specificatie van een doelbordspel.

5	10	14	7
8	3	6	1
15	0	12	9
2	11	4	13

→

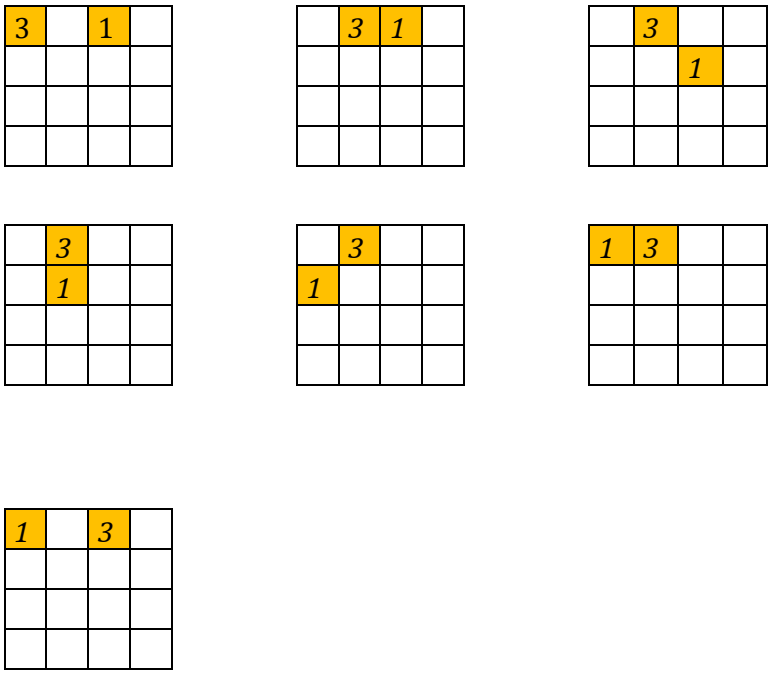
1	*2*	*3*	*4*
5	*6*	*7*	*8*
9	*10*	*11*	*12*
13	*14*	*15*	*0*

Hierboven heb je een voorbeeld van meerdere patroon databanken gecombineerd met elkaar. Hiermee vormen we statisch additieve patroon databanken. Het is belangrijk dat de patronen niet overlappen, anders zou de som van de heuristische kosten niet meer kloppen. De Manhattan kost is een speciaal geval van dit algoritme, waar elk patroon bestaat uit 1 tegel. De geelgekleurde tegels stellen patroon 1 voor. Patroon 2 zijn alle wit-kleurige tegels, patroon 3 zijn de groenkleurige tegels. Stel dat patroon 1 een heuristische kost heeft van 20, patroon 2 een kost van 15 en patroon 3 een kost van 22, dan zou de uiteindelijke kost 57 bedragen. De afzonderlijke heuristische kost van elk patroon afzonderlijk wordt berekend ongeacht de conflicten met de andere patronen. Het telt alleen haar eigen conflicten. Telkens we een bepaalde bordconfiguratie bekomen zal dus de kost tot haar doelconfiguratie berekend worden voor elk van deze patronen. De term ‘statisch’ luidt naar het feit dat de patroongroepen doorheen het oplossingsproces niet veranderen. De bovenstaande verdeling toont een 6-6-3 partitionering, wat qua snelheid en geheugenverbruik de beste verhouding teruggeeft. Een 7-8 partitionering is het snelste, maar vraagt echter een grotere geheugeninname.

De heuristieken van deze priorityfunctie zijn veel kostelijker om na te gaan, dit doordat we een zeer grote databank zullen doorzoeken. Echter wordt deze kost gecompenseerd door de enorme vermindering van het aantal gegenereerde knopen in onze priorityqueue. Door deze toepassing kunnen we de oplossingstijden met een factor 1000 doen verminderen.

Lineaire conflicten

Dit was de eerste verbetering ten opzichte van de Manhattan prioriteitsfunctie. Neem aan dat in het doelbordspel, tegel 1 zich links bevindt van tegel 2 in de bovenste rij. Maar in sommige gevallen, bevindt tegel 2 zich links van tegel 1 in de bovenste rij. De Manhattan afstand berekent de aantal geldige verplaatsingen van deze twee tegels in de bovenste rij om hun doelpositie te bereiken. Dit houdt helaas geen rekening met het feit dat een van deze twee tegels bij de verplaatsingen, zich niet meer in de bovenste rij zal bevinden. Indien we een tegel niet uit de bovenste rij verplaatsen, zou er geen geldige verplaatsing zijn om als dusdanig beide tegels van positie te veranderen in de bovenste rij. Hieruit kunnen we dus afleiden dat er voor deze twee tegels, twee extra moves moeten worden bijgeteld bij hun totale Manhattan kost. Deze prioriteitsfunctie gaat elke tegel na die zich reeds al in de juiste rij of kolom bevindt, maar in omgekeerde volgorde. Hierna wordt het aantal verplaatsingen berekend om elk van deze tegels te verschuiven naar hun doelpositie. Deze som wordt nog opgeteld met de Manhattan kost voor elke tegel afzonderlijk. Hierdoor bekomen we een accuratere heuristiek.



Figuur 5: Lineaire conflicten

Hierboven is er een simpel voorbeeld geïllustreerd waar tegels 1 en 3 in lineair conflict staan met elkaar. Beide tegels bevinden zich op de correcte rij, alleen staan ze voorlopig nog in omgekeerde volgorde. Deze prioriteitsfunctie zal in eerste instantie de kost berekenen om beide tegels te verwisselen van positie. In de beginsituatie bedraagt de kost 6 voor het lineair conflict. De Manhattan kost voor beide tegels zou 4 bedragen. Dit brengt onze heuristische kost tot een totaal van $6 + 4 = 10$. Omwille van de manhattan kost bovenop de lineaire conflictische kost, zorgt dit voor meer rekenwerk telkens we een nieuwe buur toevoegen aan de priorityqueue. Deze kost is dan ook weer afhankelijk hoe je de code hebt geschreven. In mijn geval zal het berekenen van de kost in haast lineaire tijd worden berekend. Dit komt omdat ik bij het genereren van de buren, daar de kost al laat berekenen en deze toeken aan het object dat ik teruggeef. Hierdoor zal de kost niet steeds opnieuw moeten berekend worden bij het maken van mijn priorityqueue. Omwille van deze betere benadering (in vergelijking met Manhattan) tot de exacte heuristische kost, zullen het aantal gegenereerde buren van een spelbord heel wat dalen. Conclusie voor lineair conflict: rekenwerk verhoogt, geheugeninname vermindert.

Indien mogelijk zou een betere prioriteitsfunctie leiden tot snellere oplossingen, aangezien het vinden van de paden naar de oplossing in snellere tijd gevonden zal worden. Wellicht zijn er enkele nadelen zoals hogere rekenkosten en grotere geheugeninname. Als tweede keuze zou een groter geheugen zeker te pas komen. Ondanks dat we Manhattan zouden blijven gebruiken, voor grotere NxN-puzzels zullen het aantal borden dat in de priorityqueue geplaatst worden enorm toenemen en zal er meer geheugen nodig zijn. De oplossingstijd is zeer uiteenlopend en zou dus tot zeker enkele dagen kunnen duren, maar ze zal uiteindelijk toch wel tot een oplossing komen.

Een efficiënt algoritme vinden voor een NxN-puzzel

Indien er een algoritme bestaat dat de meest optimale oplossing vindt voor grotere puzzels binnen aanvaardbare tijd, dan zou het gevonden resultaat de meest optimale oplossing zijn. Hoe weten we nu of deze in vergelijking met een andere optimale oplossing wel de meest optimale is uit die verzameling van optimale oplossingen? We zullen dus elke andere optimale oplossing ook moeten berekenen. Dit leidt tot een probleemgrootte van de oorspronkelijke probleemgrootte. Het is dus zeer onwaarschijnlijk dat er een algoritme bestaat dat een probleem A in polynomiale tijd kan oplossen. Wat er zich hier voordoet is dat een probleem A in polynomiale tijd tot A gereduceerd zal worden. Dus is dit probleem NP-moeilijk. Theoretisch gezien zijn de mogelijke bordconfiguraties eindig voor een NxN-bordspel. Dus door de hele zoekboom handmatig te doorzoeken zouden we zo wel de meest optimale oplossing kunnen bekomen. Maar het lijkt zeer onwaarschijnlijk te zijn om met een algoritme dit probleem in polynomiale tijd te kunnen uitwerken.