# Een analyse van de complexiteit van de veralgemening van het 8-puzzel probleem

Ferre Van der Vieren - 0851455

30 april 2021

Prof. Dutré

## 1 Inleiding

Het bekende 8-puzzel probleem in de computerwetenschappen, geïntroduceerd door Noyes Palmer Chapman, kan veralgemeend worden tot een NxN rooster, waarbij we geïnteresseerd zijn in het minimaal aantal benodigde zetten om tot een oplossing te komen (indien die bestaat), alsook de sequentie ervan. Hierbij wordt de ruimte- en tijdscomplexiteit van het gebruikte algoritme van uiterst belang voor grotere waarden van $N$. We zullen in dit verslag het gedrag analyseren van het gebruik van het A* zoekalgoritme uit de grafentheorie voor het oplossen van het probleem, alsook de invloed van verschillende prioriteitsfuncties en andere mogelijke ontwerpparameters.

Dit verslag werd geschreven als practicum voor het opleidingsonderdeel Gegevensstructuren & Algoritmen aan de KU Leuven. Alle code werd geschreven in Java, en is beschikbaar gesteld aan de lezer zodat deze de gevonden conclusies zelf kan natrekken. De interface van de twee belangrijkste klassen zijn toegevoegd als appendix; in de tekst wordt er dan ook regelmatig naar bepaalde functies verwezen. Een elementaire kennis van datastructuren en complexiteitsanalyse wordt verondersteld.

## 2 Empirische resultaten voor het A* algoritme

We implementeren het A* algoritme met behulp van een prioriteitsrij waarbij de toestand met de minimale prioriteit als eerste staat. We maken hierbij gebruik van de abstracte datastructuur *PriorityQueue* in de *java.util* package. De toestanden zijn in dit geval de verschillende mogelijke, geldige spelborden. De ordeningsrelatie waar de prioriteitsrij gebruik van maakt hangt af van de implementatie; er zijn meerdere mogelijkheden. We zullen er twee uitvoerig bespreken: de Hamming en Manhattan prioriteitsfunctie.

Vooraleer de complexiteit van beide prioriteitsfuncties te analyseren zullen we eerst enkele empirische resultaten bekijken, om zo een algemeen begrip te krijgen van de uitvoeringstijd[1]. Beschouw de acht puzzels die in bijlage van dit verslag zijn toegevoegd. We meten de uitvoeringstijd van het A* algoritme voor de Hamming en Manhattan prioriteitsfuncties, alsook het minimaal aantal verplaatsingen benodigd. Dit laatste hangt af van het soort puzzel, en niet van het specifiek gebruikte algoritme, en zal dus gelijk zijn voor beide implementaties. De uitvoeringstijd zal echter wel verschillen: we meten het gemiddelde over meerdere iteraties voor de kleinere puzzels om zo de invloed van onnauwkeurigheden in minieme tijdsmetingen te beperken.

| Puzzel | Minimaal aantal verplaatsingen | Hamming uitvoeringstijd (s) | Manhattan uitvoeringstijd (s) |
|---|---|---|---|
| $puzzle28.txt$ | 28 | 1.347 | 0.064 |
| $puzzle30.txt$ | 30 | 2.261 | 0.129 |
| $puzzle32.txt$ | 32 | $OutOfMemoryError$ | 1.293 |
| $puzzle34.txt$ | 34 | $OutOfMemoryError$ | 0.430 |
| $puzzle36.txt$ | 36 | $OutOfMemoryError$ | 2.052 |
| $puzzle38.txt$ | 38 | $OutOfMemoryError$ | 2.624 |
| $puzzle40.txt$ | 40 | $OutOfMemoryError$ | 1.577 |
| $puzzle42.txt$ | 42 | $OutOfMemoryError$ | 5.438 |

Tabel 1: Empirische resultaten van het A* algoritme

We merken op dat de Hamming prioriteitsfunctie al snel een onacceptabele uitvoeringstijd krijgt voor grotere puzzels[2]. Het is dan ook duidelijk dat het A* algoritme gebruik makend van de Manhattan distance optimaler is.

[1] Tijdsmetingen hangen steeds af van systeem tot systeem, maar deze resultaten geven wel een grote orde benadering.

[2] We gebruiken hier een maximum heap size van 1024MB.

# 3 Complexiteitsanalyse van Hamming en Manhattan prioriteitsfuncties

Nu we een algemeen beeld hebben van de uitvoeringstijden van beide prioriteitsfuncties kunnen we ze nader bespreken en analyseren.

## 3.1 Hamming

De Hamming prioriteitsfunctie ordent toestanden op basis van het aantal tegels in de verkeerde positie plus het aantal verplaatsingen nodig om deze toestand te bereiken vanuit de startpositie. Het aantal verplaatsingen houden we bij als een instantieveld doorheen het A* algoritme, dus deze hoeft niet berekend te worden. De complexiteit van de *hamming*() functie hangt dus integraal af van het bepalen van het aantal tegels in de verkeerde positie. Dit kan men eenvoudig implementeren door middel van een geneste lus. Indien we rij-linearisatie veronderstellen bekomen we de volgende mogelijke implementatie[3]:

```
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++) {
      int value = tiles[i][j];
      if (value != (N * i) + j + 1 && value != 0)
         counter++;
```

We gaan dus elke cel af en controleren of deze - indien verschillend van nul - op de juiste positie staat. Hiervoor is er in de binnenste lus één array access nodig. De functie zal bijgevolg een complexiteit van $\sim N^2$ hebben. Er hoeft geen onderscheid gemaakt te worden tussen worst- en best-case scenario's.

## 3.2 Manhattan

De Manhattan prioriteitsfunctie zal de prioriteitsrij ordenen volgens de som van de (horizontale en verticale) afstanden van de tegels naar de doelpositie van de waarde van de desbetreffende tegel, plus het aantal verplaatsingen om deze toestand te bereiken vanuit de initiële toestand. Zoals eerder gesteld kunnen we het aantal verplaatsingen steeds bijhouden als een variabele, dus resteert er enkel de berekening van de afstand. We zullen - analoog als hierboven - alle tegels overlopen en hiervan de doelpositie bepalen om daarna de absolute waarden op te tellen van het verschil tussen de huidige rij/kolom tot de doelrij/doelkolom.

```
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++) {
      int value = tiles[i][j];
      ...
```

[3]Hier wordt een voorbeeld weergegeven van hoe men de *hamming*() functie zou kunnen implementeren ter illustratie. Er zijn natuurlijk meerdere mogelijkheden.

```
if (correctRow != currentRow || correctColumn !=
    currentColumn)
    totalValue += Math.abs(currentRow - correctRow)
    + Math.abs(currentColumn - correctColumn);
```

Opnieuw bekomen we een geneste lus die alle elementen zal afgaan in het NxN rooster, en ook één array access waarmee iets complexere bewerkingen uitgevoerd worden waarvan de invloed verwaarloosbaar is. We bekomen dus opnieuw als resultaat $\sim N^2$, zonder worst- of best-case scenario's.

Merk op dat zowel de *hamming*() als de *manhattan*() functie beide geneste lussen gebruiken, en dus zoals gesteld een gelijkaardige complexiteit hebben. Dit betekent echter niet dat het algemeen A* algoritme gebaseerd op *hamming*() ook even efficiënt zal zijn als de Manhattan variant; we hebben dit al duidelijk ondervonden in de vorige sectie. De verklaring hiervoor is dat de Hammingwaarde enkel handig is voor de laatste fasen van het algoritme: indien we een willekeurige puzzel beschouwen is de kans laag dat we slechte opties snel kunnen elimineren door middel van de prioriteitslijst. Hierdoor zal de heap-structuur in de queue te groot worden, wat voor een onacceptabele uitvoeringstijd zorgt. De Manhattan prioriteitsfunctie geeft dus meer informatie over de huidige bordconfiguratie, en is daarom gunstiger.

# 4 Implementatie en complexiteit van $isSolvable()$

## 4.1 Implementatie

Niet elke willekeurige NxN schuifpuzzel heeft een oplossing; we moeten dus eerst nagaan of de puzzel in kwestie wel degelijk oplosbaar is. We kunnen de voorwaarde 'oplosbaar' definiëren als voldoen aan de ongelijkheid

$$\frac{\prod_{i<j}(p(b,j) - p(b,i))}{\prod_{i<j}(j-i)} \geq 0 \tag{1}$$

met $p(b,i)$ de functie die gegeven een bord $b$ en tegel $i$, de positie teruggeeft. De positie van een tegel definiëren we als de waarde van die tegel in de opgeloste eindpositie van het bord $b$. We kunnen deze ongelijkheid echter nog vereenvoudigen. De noemer is namelijk steeds positief en zal dus niks bijdragen aan de uitkomst, aangezien we enkel willen bepalen of de breuk positief of negatief. We herschrijven (1) dus als

$$\prod_{i<j}(p(b,j) - p(b,i)) \geq 0 \tag{2}$$

Voordat we vergelijking (2) kunnen toepassen als criterium voor oplosbaarheid moet de lege positie van het desbetreffende bord wel helemaal rechts onderaan staan. Dit kan eenvoudig geïmplementeerd worden door middel van geldige verschuivingen die de nulpositie systematisch naar rechtsonder beweegt. We

hebben dus twee delen in onze implementatie van $isSolvable()$. We kijken eerst na of het lege vakje helemaal rechtsonder staat, zoniet verschuiven we deze totdat de correcte positie bereikt wordt. Dan volstaat enkel de ongelijkheid in (2) uit te rekenen om de oplosbaarheid te bepalen. We implementeren $isSolvable()$ dus door middel van een if-structuur.

## 4.2 Complexiteit verplaatsing lege positie

In het eerste deel van de functie $isSolvable()$ willen we de lege positie rechtsonder in het spelbord plaatsen. De coördinaten van het nulvak kunnen bijgehouden worden als instantieveld; men kan dus in nagenoeg constante tijd nakijken of de lege positie correct staat. Indien deze niet correct staat zal de nul eerst helemaal naar rechts in de matrix geplaatst worden, om vervolgens naar onder te zakken. Deze verplaatsingen zijn in feite 'swaps'; de plaats van de lege positie wordt verwisseld met de waarde van de onder- of rechtertegel. We zullen dus de waarde van deze tegel naast de nulpositie moeten opslaan en plaatsen op de lege positie, om dan de nul te plaatsen op de nu vrijgekomen tegel. Dergelijke verplaatsingen zullen maximum $2(N-1)$ plaatsvinden; worst-case staat de lege positie namelijk linksboven. De complexiteit voor het brengen van de nul in de juiste positie is dus lineair.

Het verplaatsen van de nul is echter niet de enige noodzakelijke bewerking. De originele puzzel moet namelijk ook onveranderd blijven na het berekenen van de oplosbaarheid, aangezien we het algoritme willen toepassen op de mogelijke oplosbare puzzel in zijn originele staat, en niet op de puzzel die bekomen wordt na de verschuivingen. Om dit te bereiken zijn er twee mogelijkheden; ofwel maken we initieel een kopie van het spelbord en werken hiermee verder in de functie $isSolvable()$, ofwel houden we de initiële nulpositie bij en voeren we, na het berekenen van (2), dezelfde verschuivingen maar in omgekeerde volgorde terug uit. Indien we de eerste optie verkiezen zullen we een nieuw bord moeten initialiseren, en dus alle tegels overlopen, waardoor we op een complexiteit van $\sim N^2$ uitkomen. We merken hier op dat - in elk geval waar de nul nog niet rechtsonder staat - we een kwadratische tijdscomplexiteit hebben; ook al verplaatsen we de nul maar met enkele tegels, we zullen altijd eerst het bord volledig kopiëren. Een betere optie[4] is dus de uitgevoerde bewerkingen in omgekeerde volgorde terug te maken: zo voeren we maximaal $4(N-1)$ verplaatsingen uit. Deze methode heeft meerdere voordelen; we verkrijgen niet enkel een lineaire tijd, maar we voeren ook minder bewerkingen uit indien het bord al initieel gunstig geordend is. In het beste scenario[5] hebben we maar twee verplaatsingen in totaal uit te voeren, in het slechtste geval $4(N-1)$ en gemiddeld dus $2(N-1)$. Elke verplaatsing resulteert in drie array accesses: we vragen eerst de waarde van de desbetreffende buurtegel van de lege positie op, daarna plaatsen we deze waarde

[4] In de code is deze optie ook gehanteerd.

[5] Het beste scenario algemeen is natuurlijk dat de nul al op de juiste plaatst staat. Hier gaat het over het beste scenario indien de lege positie nog moet verplaatst worden.

op de lege positie en dan plaatsen we een nul op de oorspronkelijke buurtegel (nieuwe lege positie). We bekomen dus gemiddeld een complexiteit van $\sim 6N$ om de puzzel eerst in correcte vorm te brengen om (2) te berekenen en dan terug in de oorspronkelijke staat te brengen.

### 4.3 Complexiteit berekening ongelijkheid

Voor het berekenen van de ongelijkheid (2) moet de 2D matrix representatie ($int[][]$) via rij-linearisatie overgebracht worden tot een 1D representatie zodat we eenvoudig de functie $p()$ kunnen berekenen. Hiervoor definiëren we een functie $getStream()$ die alle tegels zal overlopen en dus een complexiteit van $\sim N^2$ heeft. We zullen dan (2) berekenen door middel van twee lussen: een buitenste lus die over de $N^2 - 1$ mogelijke waarden (exclusief nul) van de tegels itereert, en een binnenste lus die itereert over alle strikt positieve natuurlijke getallen kleiner dan de huidige waarde van de lusvariabele van de buitenste lus.

```
for (int j = 1; j < N * N; j++)
   for (int i = 1; i < j; i++)
      result *= (p(b, j) - p(b, i))
```

Voor het berekenen van $p(b, j)$ en $p(b, i)$ voor een waarde $j$ en $i$ zijn er twee array accesses nodig. Aangezien dat $\sum_{p=1}^{n} p = \frac{n(n+1)}{2}$ bekomen we

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

en dus een complexiteit van $\sim N^4$. Merk op dat de tijd nodig voor het brengen van de nul in de juiste positie verwaarloosbaar is voor grote $N$. Merk ook op dat het exact aantal array accesses kan verschillen van implementatie tot implementatie en van ontwerpkeuzes. Men kan bijvoorbeeld een rij-linearisatie als instantieveld bijhouden die berekend wordt in de constructor van het object om zo niet eerst $N^2$ bewerkingen te moeten uitvoeren voor het berekenen van (2). We kunnen echter wel concluderen dat de worst-case complexiteit van $isSolvable()$ $\sim N^4$ is.

## 5 Aantal bordposities worst-case

Bij elke iteratie van het A* algoritme kunnen er hoogstens vier bordtoestanden toegevoegd worden aan de prioriteitsrij, aangezien een lege positie maximaal vier buurtegels heeft (minder als het zich op de rand van het bord bevindt). Dit wordt recursief herhaald; we verwijderen opnieuw het bord vooraan in de prioriteitsrij en voegen de buren toe. Merk op dat een bordconfiguratie nooit meerdere keren in de prioriteitsrij zit; men zal dus dikwijls niet vier nieuwe borden kunnen toevoegen. Het aantal mogelijke configuraties voor een NxN puzzel is simpelweg een permutatie van het aantal tegels, dus $(N^2)!$. Niet elke mogelijke rangschikking van de tegels is echter bereikbaar vanuit de initiële positie; een

onbereikbare bordconfiguratie zal nooit toegevoegd worden aan de prioriteitsrij, aangezien deze nooit verkregen kan worden door middel van geldige zetten.

In sectie 4 hebben we reeds gezien dat een negatieve waarde voor $p(b,j) - p(b,i)$ de oplosbaarheid van een bord omdraait; een oplosbaar bord wordt onoplosbaar en een onoplosbaar bord wordt oplosbaar. Hier maakt oplosbaarheid niet uit, maar het principe blijft wel gelden. Oplosbaarheid is namelijk simpelweg het criterium dat de oplossing van de puzzel (alle getallen van klein naar groot en de nul rechtsonder) bereikbaar is. We kunnen dus het aantal bereikbare posities hiervan afleiden. Indien we het toestandsteken van $p(b,j) - p(b,i)$ omdraaien, zal een bereikbare positie onbereikbaar worden (cfr. oplosbaar wordt onoplosbaar), ofwel een onbereikbare positie bereikbaar (cfr. onoplosbaar wordt oplosbaar). Beschouw nu de permutaties: voor het eerste vakje hebben we $N$ mogelijke getallen, voor het tweede vakje $N-1$, etc. Voor het voorlaatste vakje hebben we echter niet twee keuzes: voor de ene keuze wordt $p(b,j) - p(b,i)$ negatief, voor de andere positief. We hebben dus maar één optie. Het aantal mogelijke toestanden voor een NxN puzzel is dus gelijk aan de helft van het aantal permutaties. We vinden bijgevolg als resultaat $\frac{(N^2)!}{2}$.

Er kunnen dus, per definitie, nooit meer dan $\frac{(N^2)!}{2}$ bordconfiguraties in de prioriteitsrij zitten. Bemerk dat dit in de praktijk niet zal plaatsvinden; de prioriteitsfunctie zorgt ervoor dat bepaalde mogelijke configuraties met weinig potentie voor een optimale oplossing nooit onderzocht worden. Dit verklaart het belang van een goede prioriteitsfunctie.

# 6 Bewerkingen prioriteitsrij worst-case

De datastructuur van een prioriteitsrij ligt aan de basis van de toepassing van het A* algoritme. Dit wordt geïmplementeerd met behulp van een heap-structuur. Het toevoegen van een element aan een heap-structuur heeft een worst-case complexiteit van $\sim log_2 M$ met M de huidige grootte van de prioriteitsrij. Het toegevoegde element wordt namelijk steeds vergeleken (volgens de geïmplementeerde Manhattan of Hamming comparator) met zijn ouder in de heap, en dit wordt recursief herhaald tot de heap terug in een geldige structuur komt ('swim' operatie). In het slechtste geval wordt er een nieuw bord toegevoegd dat vooraan in de prioriteitsrij moet komen, dus aan de wortel, waardoor men de hele boom zal overlopen.

Bij het verwijderen van een element uit de prioriteitsrij zal de wortel van de heap verwijderd worden en het element rechtsonderaan aan de wortel geplaatst worden. Dit element zal vervolgens recursief vergeleken worden met beide kinderen, en verwisseld worden met het grootste kind indien dat kind groter is dan zijn ouder. Worst-case zullen we dus de hele boom afgaan waarbij we op elk niveau twee vergelijkingen maken. We verkrijgen dus een complexiteit van $\sim 2log_2 M$ met M de huidige grootte van de prioriteitsrij.

De complexiteit is afhankelijk van de actuele grootte $M$ van de prioriteitsrij. Eerder hebben we reeds ontdekt dat er maximaal $\frac{(N^2)!}{2}$ bordconfiguraties in de rij kunnen zitten met N de dimensie van het NxN bord. Het slechtste geval voor insert/delete operaties vindt dus plaatst als de grootte van de heap-structuur maximaal is, namelijk $\frac{(N^2)!}{2}$. We bekomen dus een worst-case complexiteit van $\sim log_2 \frac{N^2!}{2}$ en $\sim 2log_2 \frac{N^2!}{2}$ met N de dimensie van het NxN bord. Hierbij wordt verondersteld dat we elementen in constante tijd kunnen vergelijken met elkaar: de klasse $HammingComparator$ en $ManhattanComparator$, die beide de $Comparator$ Java interface implementeren, moeten dus de functie $compare()$ in constante tijd uitvoeren. Dit kan eenvoudig geïmplementeerd worden door de Hamming en Manhattan waarden van een bord te berekenen bij initialisatie en dit bij te houden als veld van het bord-object, zodat deze waarden eenvoudig kunnen opgevraagd worden met een getter (constante tijd). Het aantal reeds gemaakte zetten houdt men ook bij als instantieveld van de klasse $Board$ (zie Appendix). De $HammingComparator/ManhattanComparator$ zal dan beide waardes opvragen met behulp van een getter. Indien men dit niet doet, en in de functie $compare()$ de overeenkomstige $hamming()$ en $manhattan()$ functies steeds opnieuw moet oproepen bij elke vergelijking, zal de complexiteit drastisch verhogen. Aangezien beide functies een complexiteit van $\sim N^2$ hadden (aantal array accesses), verkrijgen we dan een worst-case complexiteit van $\sim 2N^2 log_2 \frac{N^2!}{2}$. Hier zien we het belang van een goede implementatie; we kunnen de uitvoeringstijd drastisch naar beneden halen door simpelweg de Hamming en Manhattan waardes[6] van elk bord in de rij te bewaren als een instantieveld, en het dus maar één keer bij de initialisatie te bereken.

Indien men dus constante vergelijkingstijd verondersteld, bekomen we een worst-case complexiteit van $\sim log_2 \frac{N^2!}{2}$ voor het toevoegen van een bordconfiguratie en $\sim 2log_2 \frac{N^2!}{2}$ voor het verwijderen van een bordconfiguratie met N de dimensie van het NxN bord. We kunnen deze formules nu vereenvoudigen. Volgens de benadering van Stirling weten we dat $log_2 N! \sim N log_2 N$, dus

$$\begin{aligned} log_2 \frac{N^2!}{2} &= log_2 N^2! - log_2 2 \sim log_2 N^2! \\ &\sim N^2 log_2 N^2 = 2N^2 log_2 N \end{aligned} \tag{4}$$

$$\begin{aligned} 2log_2 \frac{N^2!}{2} &= 2log_2 N^2! - 2log_2 2 \sim 2log_2 N^2! \\ &\sim 2N^2 log_2 N^2 = 4N^2 log_2 N \end{aligned} \tag{5}$$

[6]Met de Hamming en Manhattan waardes bedoelen we hier de Hamming en Manhattan prioriteitsfunctie toegepast op het gegeven bord, uitgesloten van het aantal eerdere zetten. Het aantal gemaakte zetten wordt namelijk behouden als instantieveld, en kan opgevraagd worden door een getter (zie Appendix). De HammingComparator en ManhattanComparator zullen dan zowel de Hamming/Manhattan als de reeds gemaakte zetten opvragen mbv van een getter.

# 7 Optimalere prioriteitsfuncties

We hebben al gemerkt dat de invloed van een goede prioriteitsfunctie van groot belang is voor de efficiëntie van het A* algoritme (Tabel 1). De Hamming en Manhattan functies zijn al uitvoerig besproken, waarbij we kunnen stellen dat de Manhattan functie optimaler is omdat deze meer informatie geeft over de potentie van een bordconfiguratie. Dit is dan ook het ultieme doel van een goede prioriteitsfunctie; zoveel mogelijk informatie over een bord kunnen geven zodat suboptimale zetten niet berekend moeten worden.

Men kan meer geavanceerde prioriteitsfuncties implementeren die bij bepaalde bordconfiguraties efficiënter zullen werken. Zo zou men een database kunnen opstellen op basis van patronen (bv. het aantal zetten nodig voor elke mogelijke configuratie van negen tegels en de nulpositie) zodat een grote puzzel onderverdeeld kan worden in kleinere secties. Men kan dan de prioriteit van een configuratie berekenen door beroep te doen op deze database. Dit vergt uiteraard extra geheugen, maar kan de uitvoeringstijd verbeteren. Ook kunnen we optimalisaties aanbrengen aan de eerder besproken prioriteitsfuncties door bepaalde terugkerende patronen efficiënter af te handelen in de code. Dit zou de uitvoeringstijd kunnen verlagen zonder extra geheugen te vergen.

# 8 Afweging tussen geheugen, uitvoeringstijd en prioriteitsfunctie

Als afsluitende voorbeeld stellen we volgende vraag: indien we willekeurige 4x4 of 5x5 puzzels willen oplossen, wat is optimaal? Meer toegelaten geheugen, meer tijd of een betere prioriteitsfunctie? Een groter geheugen zal ervoor zorgen dat we grotere puzzels kunnen oplossen zonder de limiet te overschrijden (zoals het geval was in Tabel 1). Meer tijd zorgt ervoor dat puzzels die de geheugenlimiet niet overschrijden maar wel lang duren acceptabel worden. Een betere prioriteitsfunctie combineert echter beide voordelen: er is minder geheugen noodzakelijk én de uitvoeringstijd wordt lager. We verklaren dit intuïtief.

Indien we een betere prioriteitsfunctie hebben zullen configuraties die niet leiden tot een optimale oplossing niet verder uitgewerkt worden, en zullen dus de borden verkregen door de *neighbors*() functie op dat bord niet toegevoegd worden aan de rij. Dit zorgt er dus voor dat er minder onnodige borden in de prioriteitsrij zullen zitten, en dus dat het geheugengebruik daalt. Dit zal ook resulteren in een betere uitvoeringstijd: indien de prioriteitsrij kleiner is, vergen de bewerkingen op deze prioriteitsrij ook minder tijd (zie hoofdstuk 6). Bovendien zullen er bij een betere prioriteitsfunctie in het algemeen minder nieuwe bordconfiguraties aangemaakt en behandeld worden. We kunnen namelijk beter bepalen of een gegeven bord tot een optimale oplossing zou leiden. De empirische resultaten in hoofdstuk 2 bevestigen dit ook; de Manhattan functie heeft

voor elk bord een betere uitvoeringstijd en zal voor grote borden niet zo snel de geheugenlimiet overschrijden.

# 9 Een efficiënter algoritme

Het is belangrijk om op te merken dat een NxN schuifpuzzel tot de complexiteitsklasse NP behoort. Een heel efficiënt algoritme zal er dus niet bestaan vanwege de aard van het probleem. Men kan echter wel de term 'oplossing' breder definiëren. Indien we niet de kortste oplossing van een puzzel willen, maar enkel een geldige oplossing, kan men een algemeen algoritme opstellen dat in $O(N^3)$ loopt.[7]

We kunnen ook nog geavanceerdere optimalisaties aanbrengen aan onze huidige code. Zo is het wiskundig bewezen dat de functie $isSolvable()$ in kwadratische uitvoeringstijd berekend kan worden[8]. Bovendien kunnen we ook complexere varianten van het A* algoritme gebruiken. Ook zouden we enkel met Manhattan waardes kunnen werken, en dus bij de creatie van een nieuw bord niet langer de Hamming waarde berekenen. We kunnen dan niet meer zo eenvoudig wisselen tussen de implementatie met de Hamming prioriteitsfunctie en Manhattan, dus hebben we dit niet ingevoerd in de code voor dit verslag, maar het zou een eenvoudige optimalisatie kunnen zijn.

# 10 Conclusie

In dit verslag hebben we het bekende A* algoritme toegepast voor het oplossen van algemene NxN schuifpuzzels. Eerst is er empirisch vastgesteld dat de prioriteitsfunctie die de prioriteitsrij ordent hierbij van groot belang is; de Manhattan prioriteitsfunctie was veel efficiënter dan de Hamming functie omdat het, gegeven een willekeurig bord, meer informatie geeft over de huidige configuratie en dus veel suboptimale zetten niet verder uitwerkt. Daarna hebben we een specifieke implementatie van de functie $isSolvable()$ geanalyseerd (meerdere implementaties zijn mogelijk), en gesteld dat dit - door de lussen voor het berekenen van de ongelijkheid - een complexiteit van $\sim N^4$ heeft. Ten slotte hebben we de complexiteit en invloed van het gebruik van een prioriteitsrij verder besproken: hoe groter de huidige prioriteitsrij, hoe langer de bewerkingen op de rij zullen duren. De worst-case complexiteit voor de bewerkingen op de rij wordt dus bereikt wanneer de rij het maximaal aantal borden bevat. Een grote prioriteitsrij zorgt dus niet enkel voor meer geheugengebruik, maar ook voor een minder efficiënte uitvoeringstijd. Hierbij kan dan de link gelegd worden naar het begin van het verslag, waarbij we gemerkt hebben dat de optimalere Manhattan prioriteitsfunctie niet alleen minder geheugen gebruikt, maar ook efficiënter is voor de puzzels waarbij de geheugenlimiet niet overschreden wordt.

[7] https://cseweb.ucsd.edu/~ccalabro/essays/15_puzzle.pdf
[8] https://cseweb.ucsd.edu/~ccalabro/essays/15_puzzle.pdf

# Appendices

## A User interface klasse Board.java

```
public class Board {s

    private int[][] tiles;
    private final int N;
    private int[] zeroCoordinates;

    private int moves;
    private Board previousBoard;
    private int hammingPriority;
    private int manhattanPriority;

    public int getHammingPriority();
    public int getManhattanPriority();
    public int getMoves();
    public int[][] getTiles();
    public int getValue(int x, int y);
    public int[] getZeroCoordinates();
    public Board getPreviousBoard();
    private void setValue(int x, int y, int value);
    private void setZeroCoordinates(int x, int y);

    public Board(int[][] tiles);
    public Board(int[][] tiles, Board previousBoard);

    public int hamming();
    public int manhattan();
    public Collection<Board> neighbors();

    public boolean isSolvable();
    private List<Integer> getStream();
    private boolean isZerroBottomRight();

    public String toString();
    @Override
    public boolean equals(Object y);
    @Override
    public int hashcode();
```

## B User interface klasse Solver.java

```
public class Solver {

    public List<Board> SolutionBoards;
    public Solver(Board initial, PriorityFunc priority);

    static class HammingComparator implements
        Comparator<Board> {
         publlic int compare(Board board1, Board board2);
    }
    static class ManhattanComparator implements
        Comparator<Board> {
         publlic int compare(Board board1, Board board2);
    }

    public List<Board> solution();
```