Gegevensstructuren en Algoritmen: Practicum 2:

Verslag op het 8-puzzel probleem opgelost met het A zoekalgoritme.*

Mathijs Hubrechtsen, r0664977

01 mei 2017

Inhoudsopgave:

0. Inleiding:

Wat volgt is mijn verslag op het 8-puzzel probleem opgelost met het A* zoekalgoritme gebruik makend van de hamming en manhattan prioriteitsfuncties. De bedoeling van dit verslag is op de gegeven vragen te beantwoorden, dit is ook de structuur dat het verslag zal volgen. Als ik verwijs naar bronnen zijn deze terug te vinden achteraan het document op pagina 14. Wanneer ik code bespreek zal ik ook de relevante code er bij plaatsen. De niet relevante hulpfuncties van die code zullen er niet bijstaan, maar die zijn terug te vinden in de bijlage op pagina 11. In de relevante code zullen er heel vaak verwijzingen zijn naar het veld this.tiles, dit is de interne voorstelling van het spelbord. Die variabele zal zich meestal bovenaan de relevante code bevinden zodat het duidelijk is over wat deze variabele gaat. Bij vraag 1 heb ik de gemiddelde tijd genomen van 25 testen op elke puzzel. Elke individuele test voor eenzelfde puzzel vindt exact hetzelfde aantal verplaatsingen, zoals verwacht.

1. Vraag 1: *Geef voor elk van de volgende puzzels het minimumaantal verplaatsingen om de doeltoestand te bereiken. Geef ook de uitvoeringstijd van het A* algoritme voor de Hamming en de Manhattan prioriteitsfuncties.*

Puzzel	Aantal verplaatsingen	Hamming (s)	Manhattan (s)
puzzle28.txt	28	2	0
puzzle30.txt	30	4	0
puzzle32.txt	32	N/A	1
puzzle34.txt	34	N/A	0
puzzle36.txt	36	N/A	5
puzzle38.txt	38	N/A	8
puzzle40.txt	40	N/A	1
puzzle42.txt	42	N/A	16

/ N/A = oplossing kan niet gevonden worden binnen een redelijke tijd (< 5 minuten) */*

2. Vraag 2: *Wat is de complexiteit (aantal array accesses) van de prioriteitsfuncties hamming en manhattan, in functie van de bord grootte N?*

   a. Hamming:

C_1

```
1.  private int[][] tiles;
2.  public int hamming() {
3.    int counter = 0, result = 0;
4.    for (int i = 0; i < this.tiles.length; i++)
5.      for (int j = 0; j < this.tiles.length; j++) {
6.        counter += 1;
7.        if ((this.tiles[i][j] != counter) &&
              (this.tiles[i][j] != 0))
8.          result += 1;
9.    }
10.   return result;
11. }
```

Het is duidelijk dat er bij de hamming prioriteitsfunctie maar 2 array accesses worden gedaan, allebei in de check van de if -statement (lijn 7, C_1). Deze if -statement wordt opgeroepen in elke iteratie van de binnenste for-lus, deze wordt N keer uitgevoerd. Dit wordt op zich dan ook nog uitgevoerd in elke iteratie van de buitenste for-lus die ook N keer wordt uitgevoerd. We moeten wel rekening houden met het feit dat in java het tweede de deel van de check in de if-statement niet wordt uitgevoerd als het eerste deel al onwaar is. Dit betekent dat we een gevalsonderscheid zullen moeten opmaken voor de if-statement. Het tweede deel wordt oftewel in totaal 1 keer uitgevoerd oftewel wordt het uitgevoerd bij elke iteratie. Wiskundig neergeschreven is dit gelijk aan:

$$T_{Hamming}(N) = \frac{(\mathbf{N} * (\mathbf{N} * \mathbf{1}) + \mathbf{1}) + (\mathbf{N} * (\mathbf{N} * \mathbf{2}))}{\mathbf{2}} = \frac{\mathbf{3N^2}}{\mathbf{2}} + \frac{\mathbf{1}}{\mathbf{2}}.$$

De complexiteit van de hamming prioriteitsfunctie is dus ~1,5N^2.

b. Manhattan:

C_2

```
1. private int[][] tiles;
2. public int manhattan() {
3.   int result = 0;
4.   for (int i = 0; i < this.tiles.length; i++)
5.     for (int j = 0; j < this.tiles.length; j++)
6.      if (this.tiles[i][j] != 0)
7.        result += getDistance(tiles[i][j], i, j);
8.   return result;
9. }
```

C_3

```
1. public int getDistance(int value, int i, int j) {
2.   int[] defaultPosition =
3.         convertLineToIJ(getDefaultPosition(value));
4.   return Math.abs(i - defaultPosition[0]) +
5.          Math.abs(j - defaultPosition[1]);
6. }
```

Het aantal array accesses is een beetje moeilijker om te berekenen bij de manhattan prioriteitsfunctie dan bij de hamming prioriteitsfunctie. Dit is omdat mijn implementatie van de manhattan functie veel gebruik maakt van hulpfuncties. Om het aantal array accesses te berekenen moet er opnieuw aandacht besteed worden aan de binnenste for lus. Hier wordt ten eerste opnieuw een check uitgevoerd in de if-statement (lijn 6, C_2), indien die check voldaan is wordt de eerste hulpfunctie, getDistance opgeroepen (lijn 7, C_2). Het aantal array accesses en de code van de hulpfuncties van getDistance zijn te vinden in de bijlage achteraan dit document. Met die data kunnen we opnieuw de wiskundige uitwerking maken:

$$T_{Manhattan}(N) = N * (N * (1 + T_{getDistance}(N))) - T_{getDistance}(N)$$
$$= N * (N * (1 + T_{getDefaultPosition}(N) + T_{convertLineToIJ}(N) + 2) - T_{getDefaultPosition}(N) + T_{convertLineToIJ}(N) + 2$$
$$= N * (N * (1 + 0 + 2 + 2) - 0 + 2) + 2 = 7N^2 + 2$$

/ Voor N > 0. Anders is $T_{Manhattan}(N) = 0$. */*

De complexiteit van de manhattan prioriteitsfunctie, voor N > 0, is dus gelijk aan ~$7N^2$ wat meer is dan de complexiteit van de hamming prioriteitsfunctie. Beide functies stijgen dus wel met een kwadratisch verloop. Dit is zoals verwacht want beide functies moeten namelijk de hele N x N-array overlopen, wat als resultaat minstens een veelterm van de tweede graad heeft.

3. Vraag 3: *Leg uit hoe je isSolvable hebt geïmplementeerd. Wat is de complexiteit (aantal array accesses) in functie van de bord grootte N?*

a. Implementatie:

C_4

```
1.  private int[][] tiles;
2.  public boolean isSolvable() {
3.    int[][] tiles = getResetBoard();
4.    int[] tilesMap = getTilesMap(tiles);
5.    double numeratorResult = 1, denominatorResult = 1;
6.    for (int j = 1; j < (tiles.length * tiles.length); j++)
7.      for (int i = 1; i < j; i++) {
8.        numeratorResult *= (tilesMap[j] - tilesMap[i])
9.        denominatorResult *= (j - i);
10.     }
11.     if ((numeratorResult == Double.POSITIVE_INFINITY)
           && (denominatorResult == Double.POSITIVE_INFINITY))
12.       return true;
13.   return (numeratorResult / denominatorResult) >= 0;
14. }
```

Mijn implementatie van isSolvable verloopt in 2 delen. Het eerste deel maakt 2 nieuwe datastructuren aan die later nodig zullen zijn. Het tweede deel is de implementatie van de gegeven formule.

Het eerste deel begint met de allereerste methode die opgeroepen wordt (lijn 3, C_4), deze methode, getResetBoard, maakt met behulp van legale bewegingen een bord waarin de lege (het nul vakje) tegel zich rechts onderaan bevindt. Dit is nodig omdat de rest van isSolvable ervan uitgaat dat de lege tegel daar effectief staat. De methode getResetBoard verandert niets aan this.tiles (de interne voorstelling van het bord), het geeft een kopie van this.tiles waarin de lege tegel op de juiste plaats staat terug. Dit resultaat wordt bijgehouden in de lokale variabele: tiles. Dit betekent dat tiles en this.tiles dus niet aan elkaar gelijk zijn, tiles is de kopie, this.tiles is de interne voorstelling. Het zijn andere objecten. Na getResetBoard wordt getTilesMap opgeroepen (lijn 4, C_4), deze methode geeft een array terug waarin de index gelijk is aan de waarde, en op de plaats van die index staat de positie van die waarde in tiles. Dit zorgt ervoor dat de complexiteit van het volgende deel veel lager is.

Vervolgens start het tweede deel, de implementatie van de formule:

$$oplosbaar(b) = \frac{\prod_{i<j} p(b,j) - p(b,i)}{\prod_{i<j} j - i} \geq 0. \quad (1)$$

Eerst worden het resultaat van de teller (numerator) en noemer (denominator) geïnitialiseerd op 1 (lijn 4, C_4). Dan loopt de methode met een for-lus over alle mogelijke waarden in de puzzel. Deze is gelijk aan de verzameling:

$$V_1 = \{x \in \mathbb{Z} \mid (x > 0) \wedge (x < N^2)\}.$$

/ Met N = de bord grootte van de N x N-array. */*

In de tweede for-lus itereert de methode over dezelfde verzameling maar met 1 aanpassing: i < j, oftewel:

$$V_2 = \{x \in V_1 \mid x < j\}.$$

/* Met j ∈ V_1 */

Met deze 2 for-lussen wordt dus de iteratie van de 2 $\prod$ tekens gerealiseerd. Nu moeten nog beide de teller en noemer uitgewerkt worden. De noemer is heel makkelijk, deze is gewoon j – i, dit wordt daarna maal het vorige resultaat gedaan. Omdat j nooit kleiner is dan i, is er geweten dat de noemer nooit kleiner of gelijk aan nul zal zijn. Er zal dus nooit een deling door nul gebeuren. In de teller wordt in de plaats van het verschil tussen de waarden, het verschil van de posities genomen. Dit kan wel kleiner dan of gelijk aan nul zijn. Op deze stap wordt er ook gebruik gemaakt van de tilesMap zodat we niet de hele interne voorstelling van this.tiles moeten overlopen.

Na de uitwerking van het algoritme zouden numerator en denominator dus theoretisch het juiste resultaat bevatten. Maar hier wordt geen rekening gehouden met de getal voorstelling in de computer. Ik heb bewust gekozen voor het double type zodat er nooit overflow gebeurt van een positief getal naar een negatief getal. Dit is van belang omdat de producten heel groot worden, en dit dus regelmatig voorkomt. Maar zelfs het double type loopt op een gegeven moment over, dit is de reden waarom dat de if-statement (lijn 10, C_4) noodzakelijk is. Aangezien oneindig gedeeld door oneindig niet gedefinieerd is, wordt er zonder die check steeds false teruggegeven (lijn 12, C_4). Maar we weten dat op dit moment de variabele niet echt “oneindig” zijn, maar gewoon heel groot. Dus kunnen we stellen dat die deling sowieso een getal geeft dat groter of gelijk is aan nul. (Beide getallen zijn positief) In dit geval moet er dus true teuggegeven worden. In alle andere gevallen kunnen we gewoon lijn 12 uitvoeren.

b. Complexiteit:

De complexiteit van isSolvable is niet zo simpel te bepalen. Dit is vooral door de getResetBoard methode. Ik zal eerst de complexiteit van het algoritme bespreken van het tweede deel van het algoritme, de implementatie van de mathematische formule (1) zonder getResetBoard.

1. Deel 2: Implementatie mathematische formule (1):

De berekening voor de complexiteit begint opnieuw vanuit de binnenste for-lus (lijn 6, C_4). Het is duidelijk dat enige array accesses die van de map zijn (lijn 8, C_4). Er zijn wel 2 belangrijke observaties, ten eerste wordt er niet geïtereerd over {1, …, N}, maar over {1, …, $N^2 - 1$} en ten tweede, moet er voor de binnenste lus een gevalsonderscheid opgesteld worden.

Het worst-case geval gebeurt wanneer het element helemaal achteraan in de verzameling zit (i + 1 = = j) en j ook helemaal achteraan zit (j = = $N^2 - 1$), oftewel:

$$T_{innerFor,\ WC}(N) = (N^2 - 2) * 2 = 2N^2 - 4.$$

Het best-case geval gebeurt wanneer het element helemaal vooraan in de verzameling zit, (i = = 1) en j ook helemaal vooraan zit (j = = 1), oftewel:

$$T_{innerFor,\ BC}(N) = 0.$$

De complexiteit van dit algoritme is gelijk aan het average case, deze is gewoon het gemiddelde van het worst en best-case:

$$T_{innerFor}(N) = \frac{\mathbf{T_{innerFor,WC}(N) + T_{innerFor,BC}(N)}}{\mathbf{2}} = \frac{\mathbf{2N^2 - 4}}{\mathbf{2}} = N^2 - 2.$$

Vervolgens kan de rest van deel 2 uit rekenen:

$$T_{deel2}(N) = (N^2 - 1) * (N^2 - 2) = N^4 - 3N^2 - 2.$$

/ $(N^2 - 2) = T_{innerFor}(N)$ **/*

2. Deel 1: Datastructuren:

De uitwerking van deel 1, de datastructuren, zal ik opnieuw opsplitsen in 2 sub-delen: getResetBoard en getTilesMap, de optelling van deze twee sub-delen zorgt voor de complexiteit van deel 1.

C_5

```
1.  public int[][] getResetBoard() {
2.    int[][] tiles = deepCopy(this.tiles);
3.    int[] position = getIJPosition(tiles, 0);
5.    if ( (position[0] == tiles.length - 1) &&
           (position[1] == tiles.length - 1) ) return tiles;
5.
6.    int displacement = position[0] -
                  convertLineToIJ(getDefaultPosition(0))[0];
7.    if (displacement < 0) tiles =
8.      moveDirection(tiles, position[0], position[1], "Y", 1,
                      displacement);
9.    else if (displacement > 0) tiles =
10.     moveDirection(tiles, position[0], position[1], "Y", -1,
                      displacement);
11.
12.   position[0] = position[0] - displacement;
13.   displacement = position[1] –
                  convertLineToIJ(getDefaultPosition(0))[1];
14.   if (displacement < 0) tiles =
15.     moveDirection(tiles, position[0], position[1], "X", 1,
                      displacement);
16.   else if (displacement > 0) tiles =
17.     moveDirection(tiles, position[0], position[1], "X", -1,
18.                   displacement);
19.   return tiles;
20. }
```

De methode getResetBoard (zie boven), zoals eerder vermeld, maakt een bord waarin de lege (het nul vakje) tegel zich rechts onderaan bevindt. De complexiteit van deze methode is een lange som van de complexiteit van de hulpfuncties van deze methode. Deze hulpfuncties staan opnieuw uitgewerkt in de bijlage achteraan dit document.

$$T_{getResetBoard}(B) = T_{deepCopy}(N) + T_{getIJPosition}(N) + 1{,}5 + 2 * (1 + 1 + T_{getDefaultPosition}(N) + T_{convertLineToIJ}(N) + T_{moveDirection}(N)) + 1$$

$$= N^2 + \frac{\mathbf{N^2+3}}{\mathbf{2}} + 1{,}5 + 2 * (2 + 2 + \frac{\boldsymbol{N-3}}{\mathbf{2}}) + 1$$

$$= \frac{\mathbf{3N^2}}{\mathbf{2}} + N + 9.$$

/ De 1,5 komt van het gevalonderscheid van de if-statement (lijn 4), de + 1 achteraan de formule komt van de optelling (lijn 12). */*

C_6

```
1.  public int[] getTilesMap(int[][] tiles) {
2.    int[] result = new int[tiles.length*tiles.length];
3.    int count = 0;
4.    for (int i = 0; i < tiles.length; i++)
5.      for (int j = 0; j < tiles.length; j++) {
6.        count += 1;
7.        result[tiles[i][j]] = count;
8.      }
9.    return result;
10. }
```

De complexiteit van getTilesMap (zie boven), is vrij voor de hand liggend. Er wordt enkel een array access gedaan in de binnenste for lus dus:

$$T_{getTilesMap}(N) = N^2$$

Nu kunnen we de complexiteit van getResetBoard en getTilesMap combineren om de complexiteit van deel 1 te krijgen:

$$T_{deel1}(N) = \frac{\mathbf{3N^2}}{\mathbf{2}} + N + 9 + N^2 = \frac{\mathbf{5N^2}}{\mathbf{2}} + N + 9.$$

Met het resultaat van deel 1 kunnen we nu ook eindelijk de complexiteit van isSolvable bepalen:

$$T_{isSolvable}(N) = T_{deel1}(N) + T_{deel2}(N) = \frac{\mathbf{5N^2}}{\mathbf{2}} + N + 9 + N^4 - 3N^2 - 2$$
$$= N^4 - \frac{\mathbf{N^2}}{\mathbf{2}} + N + 7.$$

IsSolvable is dus $\sim N^4$, dit is niet een heel efficiënte complexiteit maar het is wel wat we verwachten omdat we de verzameling $\{1, \ldots, N^2 - 1\}$ ongeveer 2 keer moeten overlopen in de implementatie van de gegeven formule (1).

4. Vraag 4: *Hoeveel bord posities zitten er worst-case in het geheugen, in functie van de bord grootte N?*

Om te weten komen wat het worst-case van het aantal borden het geheugen is, moet de laagste bovengrens van het aantal borden in het geheugen worden gevonden. Dus de supremum van de bovengrenzen. De enige borden in het geheugen zijn de borden in een situatie van de prioriteitsqueue. Er mag verondersteld worden dat een bord nooit meer dan twee keer in het geheugen staat. Ook is er geweten dat niet alle twee borden van eenzelfde grootte dezelfde zijn, oftewel het is niet mogelijk om ze om te vormen tot het ander bord met legale verschuivingen. Bijvoorbeeld $Bord_1^{2x2}$ en $Bord_2^{2x2}$ zijn duidelijk niet dezelfde.

$Bord_1^{2x2}$:

1	2
3	0

$Bord_2^{2x2}$:

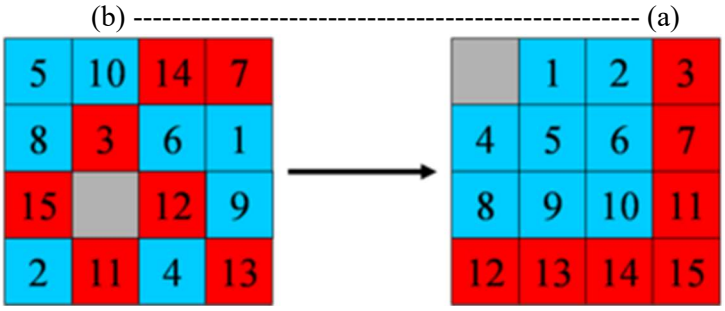
1	3
2	0

Met deze kennis is het mogelijk te stellen dat de enige mogelijke borden in de prioriteitsqueue het initiële bord en zijn permutaties zijn, dit wil zeggen de borden naar waarnaar het initiële bord kan omgevormd worden met legale verschuivingen. Dit is gelijk aan $\frac{N^2!}{2}$.[1] Hieruit volgt dat het aantal borden dat worst-case in het geheugen gelijk is aan $\frac{N^2!}{2}$. De keuze van prioriteitsfunctie zou mogelijk dit supremum nog kunnen verlagen.

5. Vraag 5: *Zijn er betere prioriteitsfunctie(s)? Indien ja, leg uit hoe deze werken.*

Ja, er is bijvoorbeeld de “linear conflict” prioritietsfunctie. Deze prioriteitsfuncties is een uitbreiding op de normale manhattan prioriteitsfunctie. Twee tegels, t_1 en t_2, zijn in lineair conflict met elkaar als en slechts als ze allebei op dezelfde lijn liggen (rij of kolom), hun doelposities ook in die lijn liggen, t_1 rechts ligt van t_2 en de doelpositie van t_1 links ligt van t_2. Als dit gebeurt moeten de tegels minstens 2 extra stappen, plus de manhattan afstand, zetten om hun doelpositie te bereiken. De prioriteitsfunctie voegt dus 2 bij de manhattan afstand in dit geval. We krijgen dan een beter beeld van welke board state een hogere prioriteit krijgt. Deze functie verschilt weinig met manhattan in termen van geheugen verbruik, er worden geen nieuwe objecten aangemaakt. De berekening duurt een klein beetje langer omdat we ook voor lineair conflict moeten uitkijken, maar doordat er een beter beeld geschetst wordt van de board state zal de effectieve totale uitvoeringstijd lager liggen. De priorityqueue zal meer intelligente beslissingen kunnen maken.

Je zou ook een “pattern database” kunnen gebruiken. Het idee van deze “prioriteitsfunctie” is om in de plaats van een effectieve berekening te doen, zoals bijvoorbeeld in de manhattan functie, we een prioriteitsfunctie ook kunnen beschrijven via het opzoeken van data. Het is een toepassing van de “divide and conquer” strategie. We zullen het oorspronkelijk probleem dus omvormen tot kleinere deelproblemen die gemakkelijker op te lossen zijn. Deze deelproblemen kunnen dan simultaan opgelost worden. Bijvoorbeeld, als we dit toepassen op een puzzel van grootte 15, dan bekomen we de onderstaande figuur[2]:



De bedoeling is om eerst de rode tegels (een pattern) naar hun “target pattern” om te vormen (zie (a)) en daarna de overige puzzel van grootte 8 op te lossen. We doen dit met behulp van de pattern database, deze is een soort van set van alle patterns die kunnen verkregen door middel van permutatie op een specifiek pattern (bijvoorbeeld de rode tegels). Deze techniek kan bij het 15 puzzel probleem een snelheid winst van een factor 2000 over normale prioriteitsfuncties boeken![3] Er zal wel veel meer geheugen nodig zijn om de pattern database op te slaan, ook zal de “berekening” een klein beetje langer kunnen duren. Maar dit wordt goedgemaakt in het feit dat deze prioriteitsfunctie zo veel efficiënter is.

6. Vraag 6: *Indien je willekeurige 4 x 4 of 5 x 5 puzzels zou willen oplossen, wat zou je verkiezen: meer toegelaten tijd (bijvoorbeeld 10x zo lang), meer geheugen (bijvoorbeeld 10x zo veel), of een betere prioriteitsfunctie?*

Ik zou zeker een betere prioriteitsfunctie verkiezen. Zoals aangetoond in vraag 1 maakt dit duidelijk het meeste verschil. Een betere prioriteitsfunctie (manhattan) zorgt ervoor dat problemen opgelost kunnen worden die niet oplosbaar zijn met een slechtere prioriteitsfunctie (hamming), dit kan vastgesteld worden bij puzzle42.txt. Ook zorgt een betere prioriteitsfunctie voor een snelheid winst bij puzzels die de slechte prioriteitsfunctie ook kan oplossen, neem als voorbeeld puzzle30.txt. Daarenboven, als men de toegelaten tijd zou verhogen, dan zal op een gegeven moment het geheugen toch overlopen want dit wordt dan niet verhoogdt. De uitbreiding is dus niet zeer significant. Aan de andere kant, het geheugen verhogen doet ook niet veel want de toegelaten tijd blijft toch dezelfde. Het is dus evident dat een betere prioriteitsfunctie de beste optie is.

7. Vraag 7: *Denk je dat er een efficiënt algoritme bestaat wat, zelfs voor grote puzzels, de optimale oplossing binnen praktische tijd kan vinden?*

Als met efficiënt bedoeld wordt: er bestaat een algoritme dat in polynomiale tijd steeds een correcte oplossing vindt voor grote puzzels, denk ik niet dat er zo een algoritme bestaat. Dit is omdat er in 1986 aangetoond is geweest dat het n-puzzel probleem een NP-Compleet probleem is.[4] Omdat NP-Complete problemen Niet-Deterministisch Polynomiaal Herkenbaar zijn, is het dus duidelijk dat het n-puzzel probleem niet gevonden kan worden in polynomiale tijd. De enige manier dat er wel een algoritme bestaat dat een oplossing kan vinden in polynomiale is als P = NP, wat onwaarschijnlijk is. Maar zelfs als zou dit het geval zijn, dan moet de puzzel nog steeds ingelezen worden en deze operatie is van $\sim N^2$ tijd, wat ook al niet ontzettend efficiënt is. Zeker bij heel grote N zal deze operatie redelijk lang duren. Er zijn natuurlijk wel algoritmes die efficiënter zijn dan andere algoritmes voor dit probleem, maar dat wil niet zeggen dat die algoritmes heel "efficiënt" zijn. Het IDA* algoritme is een voorbeeld van een efficiënter algoritme voor dit probleem.

8. Bijlage:

C_7

```
1. private int[][] tiles;
```

Hiervan wordt gebruik gemaakt in veel hulpfuncties, het is de interne voorstelling van het spelbord.

C_8

```
1. public int[] convertLineToIJ(int position) {
2.   int[] result = {-1, -1};
3.   result[0] = (position - 1)/this.tiles.length;
4.   result[1] = (position - 1)%this.tiles.length;
5.   return result;
6. }
```

De complexiteit van deze hulpfuncties is makkelijk te bepalen, er zijn geen lussen en slechts 2 array accesses. Dus $T_{convertLineToIJ}(N) = 2$.

C_9

```
1. public int getDefaultPosition(int value) {
2.   if (value == 0)
3.     return(this.tiles.length * this.tiles.length);
4.   return value;
5. }
```

Deze methode doet geen enkele array accesses, dus $T_{getDefaultPosition}(N) = 0$.

C_{10}

```
1. public int[][] deepCopy(int[][] array) {
2.   int[][] result = new int[array.length][array[0].length];
3.   for (int i = 0; i < array.length; i++)
4.     for (int j = 0; j < array.length; j++)
5.       result[i][j] = array[i][j];
6.       return result;
7. }
```

Heel evidente berekening voor deepCopy: $T_{deepCopy}(N) = N * N = N^2$.

C_{11}

```
1.  public int[] getIJPosition(int[][] tiles, int value) {
2.    int[] result = {-1, -1};
3.    for (int i = 0; i < this.tiles.length; i++)
4.      for (int j = 0; j < this.tiles.length; j++)
5.        if (tiles[i][j] == value) {
6.          result[0] = i;
7.          result[1] = j;
8.          return result;
9.        }
10.   return result;
11. }
```

De complexiteit van deze hulpfunctie (zie boven) maakt gebruik van een gevalonderscheid:

Worst Case:

$$T_{getIJPosition,\ WC}(N) = N * (N) + 2 = N^2 + 2.$$

Best Case:

$$T_{getIJPosition,\ BC}(N) = 1.$$

Average Case:

$$T_{getIJPosition}(N) = \frac{\mathbf{T_{getIJPosition,WC}(N) + T_{getIJPosition}(N)}}{\mathbf{2}} = \frac{\mathbf{N^2+3}}{\mathbf{2}}.$$

C_{12}

```
1.  private int[][] moveDirection(int[][] tiles, int i1, int
      j1, String XorY, int direction, int displacement) {
2.    int i = i1, j = j1;
3.    while (displacement != 0) {
4.      if (XorY == "X") {
5.        tiles[i][j] = tiles[i][j + direction];
6.        tiles[i][j + direction] = 0;
7.        j += direction;
8.      }
        else if (XorY == "Y") {
9.        tiles[i][j] = tiles[i + direction][j];
10.       tiles[i + direction][j] = 0;
11.       i += direction;
12.     }
13.     displacement += direction;
14.   }
15.   return tiles;
16. }
```

Een ietwat meer complexe methode. Er moet hier rekening worden gehouden met het feit dat deze methode een belangrijke hulpfunctie in getResetBoard is. Zo is er geweten wanneer distance maximaal (en minimaal) gelijk is.
Het maximum is:

$$distance_{max} = +/- ((tiles.length - 1) - 0) = +/- (N - 1).$$

Voor een minimale distance moet er een kleine omweg genomen worden. In de plaats van te kijken naar distance in moveDirection, moet er gekeken worden hoe moveDirection wordt opgeroepen. Er is geweten dat het oftewel opgeroepen wordt met |distance| <= $|distance_{max}|$, oftewel wordt het niet opgeroepen, wanneer distance nul is. Deze "niet" oproep zal genomen worden als minimum.

$$distance_{min} = 0.$$

Nu kunnen we de effectieve complexiteit bepalen:

$$T_{moveDirection}(N) = \frac{\mathbf{T_{moveDirection,WC}(N) + T_{moveDirection,BC}(N)}}{\mathbf{2}} = \frac{\mathbf{(N-1) * 3+0}}{\mathbf{2}} = \frac{\mathbf{N-3}}{\mathbf{2}}.$$

9. Bronnen:

1. Weisstein, Eric W. "Odd Permutation." From *MathWorld*--A Wolfram Web resource: http://mathworld.wolfram.com/OddPermutation.html

2. "Pattern Database." Heuristicswiki. N.p., n.d. http://heuristicswiki.wikispaces.com/pattern+database

3. Korf, Richard E., and Ariel Felner. Disjoint Pattern Database Heuristics. Rep. N.p.: Elsevier, n.d. Disjoint Pattern Database Heuristics - ScienceDirect. http://www.sciencedirect.com/science/article/pii/S0004370201000923

4. Ratner, Daniel, and Manfred Warmuth. *FINDING A SHORTEST SOLUTION FOR THE N XN EXTENSION OF THE H-PUZZLE IS INTRACTABLE.* Rep. N.p.: n.p., n.d. *FINDING A SHORTEST SOLUTION FOR THE N XN EXTENSION OF THE H-PUZZLE IS INTRACTABLE.* University of California Santa Cruz, 1986. http://www.aaai.org/Papers/AAAI/1986/AAAI86-027.pdf