Verslag Practicum 2:

Arnout Coenegrachts, r0665757

30 april 2020

1 Vraag 1:

In tabel 1 staat het minimum aantal verplaatsingen dat nodig is om de eindtoestand te bereiken voor zowel de Hamming en de Manhattan prioriteitsfuncties. De tijdsduur is opgenomen via System.nanoTime() van Java. Voor de puzzels 28 en 30 kon het algoritme de oplossing vinden voor beide prioriteitsfuncties, maar voor de andere puzzels kon er geen oplossing gevonden worden via de Hamming prioriteitsfunctie, omwille van een OutOfMemoryError, zelfs nadat het maximaal aantal geheugen dat Java gebruiken mag verhoogd was. Een screenshot van de volledige error kan gevonden worden in figuur 3 in de appendix.

Table 1: Het minimum aantal verplaatsingen en de hoeveelheid tijd nodig om de doeltoestand te bereiken.

Puzzel	Aantal verplaatsingen	Hamming (s)	Manhattan (s)
puzzle28.txt	28	2.483175936	0.182055304
puzzle30.txt	30	4.99957399	0.306797523
puzzle32.txt	32	/	1.94863768
puzzle34.txt	34	/	0.998084759
puzzle36.txt	36	/	13.15391548
puzzle38.txt	38	/	3.266948254
puzzle40.txt	40	/	2.133228536
puzzle42.txt	42	/	33.544258168

Uit de tabel is het duidelijk te zien dat de Manhattan prioriteitsfunctie veel efficiënter is dan de Hamming prioriteitsfunctie. Manhattan lost dezelfde puzzels veel sneller op dan Hamming, en kan al de gegeven puzzels oplossen zonder het volledige beschikbare geheugen in te nemen. Dit viel dan ook te verwachten, omdat de Hamming prioriteitsfunctie enkel maar verlaagt wanneer er een getal op de correcte plaats wordt gezet, terwijl de Manhattan prioriteitsfunctie dit ook doet wanneer een getal dichter bij zijn correcte plaats komt. Hierdoor kan Manhattan beter de volgende stap kiezen.

2 Vraag 2:

De code die de Hamming prioriteitsfunctie uitwerkt wordt gegeven in figuur 1a. Deze code heeft een nested loop met daarin één array access, in de vorm van " *tegel = tiles[i][j];* ". De complexiteit hiervan is dus die van een nested loop, namelijk $\sim N^2$.
De code die de Manhattan prioriteitsfunctie uitwerkt wordt gegeven in figuur 1b. Deze code

```java
// return number of blocks out of place
public int hamming()
{
    int[][] tiles = this.getTiles();
    int n = this.getSize();
    int count = 0;
    int tegel;
    int solutionTile;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            tegel = tiles[i][j];
            if (i == n-1 && j == n-1) {
                solutionTile = 0;
            }
            else {
                solutionTile = n*i+j+1;
            }
            if (tegel !=0 && tegel != solutionTile) {
                count++;
            }
        }
    }
    return count;
}
```

(a) Hamming

```java
// return sum of Manhattan distances between blocks and goal
public int manhattan()
{
    int n = this.getSize();
    int[][] tiles = this.getTiles();
    int I;
    int J;
    int count = 0;
    int tegel;
    int solutionTile;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            tegel = tiles[i][j];
            if (i == n-1 && j == n-1) {
                solutionTile = 0;
            }
            else {
                solutionTile = n*i+j+1;
            }
            if (tegel !=0 && tegel != solutionTile) {
                I = (tegel-1)/n; //use integer division to find the row coordinate
                J = tegel - 1 - I*n; //reverse n*i+j+1 to find the column coordinate
                count = count + Math.abs(i - I) + Math.abs(j - J);
            }
        }
    }
    return count;
}
```

(b) Manhattan

Figure 1: De code die de Hamming en de Manhattan prioriteitsfunctie berekent.

heeft ook een nested loop met daarin slechts één array acces, in de vorm van " *tegel = tiles[i][j];* ". De complexiteit hiervan is dus hetzelfde als die van de Hamming prioriteit, namelijk $\sim N^2$.

3 Vraag 3:

De code van mijn isSolvable() implementatie kan je zien in figuur 2 op de volgende pagina. De code van alle hulpfuncties staan in de appendix.

1. De eerste stap in mijn implementatie is een deep copy nemen van de tegels van het bord, via de methode genaamd "deepCopy" (zie figuur 4). Deze methode werkt via een nested loop die over alle individuele tegels gaat en deze een voor een kopiëert naar een andere matrix. Deze stap heeft dus een complexiteit van $\sim N^2$. Deze stap wordt uitgevoerd om te voorkomen dat de tegels van het bord gewijzigd wordt wanneer we de nul tegel naar de rechter onderhoek verplaatsen.

2. De tweede stap is de coordinaten van de 0 tegel te vinden, via de "find" methode (zie figuur 5). Deze bevat ook een nested loop die over alle individuele tegels gaat, maar deze dubbele loop wordt onderbroken wanneer het gezochte getal, in dit geval 0, gevonden is. In het slechtste geval, wanneer de gezochte tegel helemaal rechts onderaan is, is de complexiteit hiervan $\sim N^2$. In het beste geval, wanneer de gezochte tegel links bovenaan is, is de complexiteit ~ 1. We kunnen er van uit gaan dat we gemiddeld maar tot in de helft van de matrix moeten gaan om de tegel te vinden, wat ons een

complexiteit van $\sim N^2/2$ geeft.

```
// is the initial board solvable? Note that the empty tile must
// first be moved to its correct position.
public boolean isSolvable()
{
    int[][] grid = deepCopy(getTiles());
    int n = getSize();
    int[] coord = find(0, grid);
    int[] new_coord;
    while (coord[0] != n-1) {
        new_coord = new int[]{coord[0]+1, coord[1]};
        grid = exchange(coord, new_coord, grid);
        coord = new_coord;
    }
    while (coord[1] != n-1) {
        new_coord = new int[]{coord[0], coord[1]+1};
        grid = exchange(coord, new_coord, grid);
        coord = new_coord;
    }
    float breuk = 1;
    int[] row = toRow(grid);
    int N = row.length;

    for (int j = 1; j < N; j++) {
        for (int i = 1; i < j; i++) {
            breuk = breuk * ( findRow(j, row) - findRow(i, row) ) / ( j - i );
        }
    }
    if (breuk >= 0) {
        return true;
    }
    else
        return false;
}
```

Figure 2: De code die gebruikt werd om te controleren of de initiële bord configuratie oplosbaar is

3. Hierna komen 2 aparte while-lussen, eentje die de nul-tegel naar de onderste rij brengt, en een tweede die de nul-tegel naar de meest rechtste kolom brengt. In beide gevallen wordt de nul-tegel verplaatst via de "exchange" methode, die 4 array accesses gebruikt om 2 tegels van plaats te veranderen. De code hiervan kan je zien in figuur 6. Hoe vaak deze lussen uitgevoerd worden, hangt dus van de invoer af, maar het maximum aantal keer is $N - 1$ per lus. We kunnen er alweer van uitgaan dat de 0 zich ongeveer in de helft van de matrix bevindt, wat geeft dat elke lus $\sim N/2$ keer uitgevoerd moet worden. Dat geeft ons dan een complexiteit van $\sim 2N$ per lus, ofwel $\sim 4N$ in totaal

voor deze stap.

4. De volgende stap zet de NxN matrix om naar een rij van lengte N^2. Dit gebeurt via een methode genaamd "toRow" (zie figuur 7), die een nested loop bevat. De complexiteit hiervan is $\sim N^2$.

5. De laatste stap berekent de functie (1) die weergeeft of de bord configuratie oplosbaar is.

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

Hiervoor werd het productteken buiten de breuk gehaald (zie vergelijking (2)), waardoor deze via een nested loop berekend kan worden.

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

Het exact aantal keer dat de inhoud van deze dubbele lus voorkomt, wordt gegeven door $\sum_{j=1}^{N^2-1}(j-1) = \frac{(N^2-2)\times(N^2-1)}{2}$. Hieruit volgt dat we deze lus $\sim N^4/2$ keer voorkomt. Binnen deze dubbele lus komt 2 keer de methode "findRow" voor (zie figuur 8). Deze methode zoekt de plaats van de opgegeven waarde binnen een rij van lengte N^2 via één lus. Als we aannemen dat de gezochte waarde gemiddeld in de helft van deze rij zit, dan heeft dit telkens een complexiteit van $\sim N^2/2$. De complexiteit van deze stap is dus $\sim N^4/2 \times (N^2/2 + N^2/2) = \sim N^6/2$.

Om de complexiteit van de volledige "isSolvable()" te vinden, moeten we gewoon de som van de complexiteiten van deze stappen nemen. Dit geeft ons $\sim N^6/2 + 5N^2/2 + 4N$. De complexiteit is dan $\sim N^6/2$.

4 Vraag 4:

Het aantal mogelijke configuraties van een NxN bord is $N^2!$. Dit resultaat is gemakkelijk te zien als je je inbeeldt dat je voor elk vakje zelf de waarde kiest en zo het bord opvult. Voor het eerste vakje heb je N^2 verschillende mogelijkheden. Voor het tweede vakje heb je $N^2 - 1$ verschillende mogelijkheden, want elk vakje moet een verschillende waarde hebben. Het derde vak je heeft dan weer $N^2 - 2$ verschillende mogelijkheden. Hieruit volgt dus het totaal aan $(N^2)!$ verschillende configuraties. Maar niet al deze configuraties zijn oplosbaar. Om te benaderen hoeveel van deze configuraties oplosbaar zijn, kijken we naar vergelijking (1). Deze zegt dat als de waarde hiervan negatief is, dat het bord niet oplosbaar is. Aangezien $i < j$, is de noemer zoiezo positief, waardoor we enkel maar naar de teller moeten kijken. Als i en j op hun correcte plaats staan, dan is de term in het product telkens positief. Als er een permutatie gebeurt is, waardoor er een i met een j verwisseld is $(i, j \neq 0)$, dan gaan er ook negatieve termen voorkomen. Zo één permutatie levert een oneven aantal negatieve termen op, waardoor het hele product negatief is en het bord dus niet oplosbaar. Maar als we een

even aantal van deze permutaties hebben, dan is het hele product positief, en is het bord wel oplosbaar. We gaan er dus van uit dat er $(N^2)!/2$ verschillende mogelijke bordcombinaties zijn die oplosbaar zijn voor een NxN bord.
Ook al wordt telkens de bordconfiguratie met de laagste prioriteit uit de priority queue verwijdert, toch blijven deze nog in het geheugen zitten, want elk bord heeft het vorige bord waaruit het ontstaan is opgeslagen. Hierdoor zitten er in het slechtste geval $(N^2)!/2 - 1$ bordposities in het geheugen, waarbij die -1 veroorzaakt wordt door de eindpositie die het algoritme net zoekt.

5 Vraag 5:

De priority queue die gebruikt wordt in deze implementatie is die van java.util zelf. De documentatiepagina hiervan licht ons in dat deze gebaseerd is op een "heap"-structuur. Volgens het handboek *Algoritms: Fourth edition* van Sedgewick is de complexiteit maximaal $1 + log_2 N$ voor het toevoegen van een element en maximaal $2 \times log_2 N$ voor het verwijderen van een element. In tilde notatie wordt dit voor toevoegen en verwijderen respectievelijk $\sim log_2 N$ en $\sim 2log_2 N$.

6 Vraag 6:

Ik kan geen prioriteitsfunctie die beter werkt dan de Manhattan functie bedenken.

7 Vraag 7:

Een betere prioriteitsfunctie is volgens mij de beste optie. Tijd was geen probleem voor mijn implementatie, omdat altijd ofwel de oplossing vind, ofwel een OutOfMemoryError geeft binnen de vereiste tijd (een voorbeeld van deze tweede optie kan je vinden in figuur 3).
Extra geheugen zou de Hamming kunnen helpen. Maar aangezien Manhattan grotere puzzels met minder geheugen en grotere snelheid dan Hamming kan oplossen, zou een nog betere prioriteitsfunctie meer voordelen hebben dan gewoon meer geheugen.

8 Vraag 8:

Ik denk van niet. Het A* algoritme, of enig ander algoritme dat ook gebaseerd is op een priority queue, is volgens mij het meest efficiënte algorimte voor dit soort probleem. Het gebruik van een priority queue heeft als voordeel dat het een eenvoudige en snelle manier is om de stap met het meeste kans om naar de juiste oplossing te leiden, namelijk die met de laagste prioriteitsscore, als eerste te nemen. Maar voor zeer grote puzzels, zeker als ze veel stappen nodig hebben om de doel positie te bereiken, gaat het onmogelijk zijn om binnen een redelijke tijd doorheen de verschillende opties te gaan, zelfs als er een betere prioriteitsfunctie zou bestaan. Daarnaast gaat de functie om te testen of de puzzel oplosbaar is zeer veel

tijd in beslag nemen, omdat dat $\sim N^6/2$ vereist (omwille van het $\prod_{i<j}^{N^2}$ met daarbinnen 2 zoekfuncties).

9 Appendix

```
laptop-Arnout-2:gna-practicum2-2019-2020 coenegrachtsbollenmacbookair$ ant run -Dboard=boards/puzzle32.txt
Picked up _JAVA_OPTIONS: -Xmx1024m
Buildfile: /Users/coenegrachtsbollenmacbookair/Documents/School/Universiteit/2019-2020/Semester_2/GnA/Verslag-2/gna-practicum2-2019-2020/build.xml

clean:
   [delete] Deleting directory /Users/coenegrachtsbollenmacbookair/Documents/School/Universiteit/2019-2020/Semester_2/GnA/Verslag-2/gna-practicum2-2019-2020/build

check-libpract-exist:

compile:
    [mkdir] Created dir: /Users/coenegrachtsbollenmacbookair/Documents/School/Universiteit/2019-2020/Semester_2/GnA/Verslag-2/gna-practicum2-2019-2020/build/classes
    [javac] Compiling 6 source files to /Users/coenegrachtsbollenmacbookair/Documents/School/Universiteit/2019-2020/Semester_2/GnA/Verslag-2/gna-practicum2-2019-2020/build/classes
    [javac] Note: /Users/coenegrachtsbollenmacbookair/Documents/School/Universiteit/2019-2020/Semester_2/GnA/Verslag-2/gna-practicum2-2019-2020/src/gna/UnitTests.java uses or overrides a deprecated API.
    [javac] Note: Recompile with -Xlint:deprecation for details.

jar:
    [mkdir] Created dir: /Users/coenegrachtsbollenmacbookair/Documents/School/Universiteit/2019-2020/Semester_2/GnA/Verslag-2/gna-practicum2-2019-2020/build/jar
      [jar] Building jar: /Users/coenegrachtsbollenmacbookair/Documents/School/Universiteit/2019-2020/Semester_2/GnA/Verslag-2/gna-practicum2-2019-2020/build/jar/practicum.jar

run:
     [java] Picked up _JAVA_OPTIONS: -Xmx1024m
     [java] Exception in thread "main" java.lang.OutOfMemoryError: Java heap space: failed reallocation of scalar replaced objects
     [java] Java Result: 1

BUILD SUCCESSFUL
Total time: 2 minutes 15 seconds
laptop-Arnout-2:gna-practicum2-2019-2020 coenegrachtsbollenmacbookair$
```

Figure 3: De OutOfMemoryError die verkregen werd bij het uitvoeren van het programma om puzzel 32 op te lossen met behulp van de Hamming prioriteitsfunctie.

```
private int[][] deepCopy(int[][] tiles) {
    int N = tiles.length;
    int[][] newTiles = new int[N][N];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            newTiles[i][j] = tiles[i][j];
        }
    }
    return newTiles;
}
```

Figure 4: *deepCopy*: De code om een deep copy te maken van de tegel matrix

```
public static int[] find(int tile, int[][] tiles) {
    int[] coord = new int[2];
    int n = tiles.length;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (tiles[i][j] == tile) {
                coord[0] = i;
                coord[1] = j;
                return coord;
            }
        }
    }
    return coord;
}
```

Figure 5: *find*: De code om een gegeven getal te zoeken in de tegel matrix

```
private int[][] exchange(int[] coord1, int[] coord2, int[][] tiles) {
    int tile1 = tiles[ coord1[0] ][ coord1[1] ];
    int tile2 = tiles[ coord2[0] ][ coord2[1] ];
    int n = tiles.length;
    if (tile1 == 0 || tile2 == 0) {
        tiles[ coord1[0] ][ coord1[1] ] = tile2;
        tiles[ coord2[0] ][ coord2[1] ] = tile1;
    }
    return tiles;
}
```

Figure 6: *exchange*: De code om 2 tegel van plaats te wisselen, zolang een van de 2 tegels de nul-tegel is.

```
private int[] toRow(int[][] tiles) {
    int n = tiles.length;
    int size = n*n;
    int[] row = new int[size];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            row[n*i+j] = tiles[i][j];
        }
    }
    return row;
}
```

Figure 7: *toRow*: De code om de tegel matrix om te vormen naar 1 rij van lengte N^2.

```
private int findRow(int tile, int[] row) {
    int coord = -1;
    int n = row.length;
    for (int i = 0; i < n; i++) {
        if (row[i] == tile) {
            coord = i;
            return coord;
        }
    }
    return coord;
}
```

Figure 8: *findRow*: De code om een gegeven getal te zoeken in de rij-representatie van de tegel matrix.