Gegevensstructuren en algoritmen

Practicum 1: Sorteeralgoritmes

Author:
Stef Tweepenninckx, r0677232

1 mei 2018

Inleiding

Overzicht

Puzzel	Aantal vergelijkingen	Hamming (s)	Manhattan (s)
puzzle28.txt	28	2.435	0.113
puzzle30.txt	30	3.920	0.140
puzzle32.txt	32	>5 min	4.803
puzzle34.txt	34	>5 min	1.093
puzzle36.txt	36	>5 min	14.168
puzzle38.txt	38	>5 min	10.885
puzzle40.txt	40	>5 min	2.490
puzzle42.txt	42	>5 min	8.528

Tabel 1: Resultaten experiment met Hamming en Manhattan prioriteitsfunctie

Tijdscomplexiteit Hamming

De Hamming prioriteitsfunctie kijkt naar het aantal elementen die op de foute plaats staat. We lopen over de hele puzzel en vergelijken de waarde op een bepaalde positie i,j met de verwachte waarde. Als deze 2 waarden niet overeenkomen verhogen we het resultaat.

```
public int hamming() {
        int expected = 1;
        int result = 0;
        for (int i = 0; i < N; i++){
                for (int j = 0; j <N; j++) {
                        if (tiles[i][j] != expected && tiles[i][j] != 0) result++;
                        expected++;
                }
        }
        return result;
}
```

In mijn implementatie van de Hamming functie gebruik ik twee for-lussen die beide van 0 tot N lopen. Bij elke iteratie zijn er maximaal twee array accesses nodig. In dit worst-case geval is de tijdscomplexiteit $\sim 2N^2$.
In het beste geval (de puzzel is al opgelost) wordt de 2^e array access slechts 1 keer uitgevoerd, bij het 0-vakje (dit kan nooit gelijk zijn aan de variabele *expected*). Bij de gewone vakjes is de if-clausule al gefalsifiëerd door de 1^e array access en wordt de 2^e dus niet uitgevoerd. De best-case tijdscomplexiteit is dus $\sim N^2 + 1$.

Tijdscomplexiteit Manhattan

De Manhattan prioriteitsfunctie kijkt naar de totale afstand tussen de elementen en hun juiste locatie. We lopen over de hele puzzel en vergelijken de waarde op een bepaalde positie i,j met de verwachte

waarde. Als deze 2 waarden niet overeenkomen berekenen we de afstand tot de juiste positie en verhogen we het resultaat met deze afstand.

```
public int manhattan() {
        int count = 0;
        int expected = 1;
        for (int i = 0; i<N; i++)
                for (int j=0; j<N; j++){
                        if(tiles[i][j] != expected && tiles[i][j] != 0)
                        {
                                        int expi = Math.floorDiv(tiles[i][j] -1, N);
                                        int expj = Math.floorMod(tiles[i][j] -1, tilesN);
                                        int dist = Math.abs(expi - i) + Math.abs(expj - j);
                                        count = count + dist;
                        }
                        expected++;
                }
        return count;
```

Figuur 1: Implementatie van Manhattan prioriteitsfunctie

Net als bij de Hamming functie, gebruikt mijn Manhattan functie twee for-lussen die van 0 tot N lopen. Bij elke iteratie zijn er maximaal 4 array accesses nodig. De worst-case tijdscomplexiteit is dus $\sim 4N^2$.
In het beste geval worden de binnenste array accesses nooit uitgevoerd en worden enkel bij het 0-vakje beide accesses van de if-clausule uitgevoerd. In het best-case geval is de tijdscomplexiteit dus $\sim N^2 + 1$.

isSolvable

In het eerste deel van isSolvable() localiseer ik het lege 0-vakje en verplaats het naar de juiste i locatie en de juiste j locatie: $i = N - 1$, $j = N - 1$.
In het tweede deel bereken ik de formule zelf met behulp van twee for-loops. Bij elke iteratie bereken ik de teller en noemer en vermenigvuldig het deeltal met het voorgaande resultaat. Op deze manier kan je de productfunctie berekenen. Op het einde return je of het resultaat groter of gelijk is dan 0 en het bord dus al dan niet oplosbaar is.

```
// is the initial board solvable? Note that the empty tile must
// first be moved to its correct position.
public boolean isSolvable() {
        Board temp = new Board(this.tiles);

        //locate empty space and move to correct position
        int[] location = temp.emptySpaceLocation();

        //correct location in NxN board == [N-1][N-1]
        //move empty space to correct i position
        for (int i = location[1]; i < temp.tiles[0].length-1; i++){
                temp = move(temp, location, new int[] {location[0], location[1] + 1});
                location[1] += 1;
        }

        //move empty space to correct j position
        for (int j = location[0]; j < temp.tiles[1].length-1; j++){
                temp = move(temp, location, new int[] {location[0] + 1, location[1]});
                location[0] +=1;
        }


        //calculate formula
        double numerator;
        double denominator;
        double result = 1;
        double elements = Math.pow(this.tiles[0].length,2) - 1;
        for (int j = 1; j <= elements; j++){
                for (int i = 1; i < j; i++){
                        numerator = p(temp, j) - p(temp, i);
                        denominator = j - i;
                        result = result * numerator/denominator;
                }
        }
        //result must be smaller or equal to 0
        return (result >= 0);
}
```

In mijn implementatie gebruik ik 1 hulpfunctie *move* om de blokjes te verschuiven. In het gemiddelde geval (0-vakje staat in het midden) wordt deze methode N keer uitgevoerd met 2 array accesses, dus een gemiddelde tijdscomplexiteit van $\sim 2N$.

Borden in geheugen

Elk bord met grootte $N \times N$ heeft maximaal $N^2!$ permutaties/verschillende bordconfiguraties. Niet alle permutaties zijn bereikbaar vanuit de initiële configuratie, slechts de helft kan bereikt worden met legale verschuivingen. Het maximaal aantal bordconfiguraties in het geheugen is dus gelijk aan $\frac{N^2!}{2}$. De keuze van prioriteitsfunctie zou dit eventueel nog kunnen verlagen.

Worst-case tijdscomplexiteit

De built-in priorityqueue van Java die ik gebruikt heb, heeft een complexiteit van $O(log(N))$ om een element aan de rij toe te voegen. In de vorige vraag hebben we vastgesteld dat er maximaal $\frac{N^2!}{2}$ bordconfiguraties in het geheugen zitten. Dit impliceert dat het volgende toe te voegen bord moet worden vergeleken met $log(\frac{N^2!}{2})$ borden. De gekozen prioriteitsfunctie (Hamming of Manhattan) bepaalt mee de complexiteit van de vergelijking. De worst-case complexiteit is dan:

$$log(\frac{N^2!}{2}) * 1.5N^2 * 2 \tag{1}$$
$$=3log(\frac{N^2!}{2}) * N^2 \tag{2}$$
$$=3N^2 * (log(N^2!) - log(2)) \tag{3}$$
$$=3N^2 * (log(N^2!) - 1) \tag{4}$$
$$=3N^2 * log(N^2!) - 3N^2 \tag{5}$$

Met behulp van Stirling's benadering $log_2(n!) \approx nlog_2(n) - log_2(e)n + O(log_2(n))$ bekomen we:

$$3N^2 * log(N^2!) \tag{6}$$
$$=\sim 3N^2 * N^2 log(N^2) \tag{7}$$
$$=\sim 6N^4 log(N) \tag{8}$$

Het verwijderen van een element met de laagste priority uit de priorityqueue gebeurt in constante tijd, het staat immers vooraan in de queue. Elk ander element gebeurt in lineaire tijd.

Betere prioriteitsfuncties

Tijd, geheugen of algoritme?

Een betere prioriteitsfunctie heeft naar mijn mening het meeste invloed. Dit kunnen we bijvoorbeeld zien bij de experimenten in vraag 1: met de Hamming functie zijn 6 van de 8 borden niet oplosbaar binnen de 5 minuten, terwijl de Manhattan functie een maximale tijd van 14.2 seconden heeft. Dit toont de invloed van een goed algoritme op de efficiëntie.
Dat meer tijd niet de beste manier is kunnen we op dezelfde manier aantonen: de langste tijd met Manhattan is 14.2 seconden, hetzelfde bord met Hamming is meer dan 5 minuten. 10 keer zo veel tijd heeft dus amper invloed op een probleem met deze complexiteit.
Meer geheugen is ook niet de oplossing, aangezien een betere prioriteitsfunctie impliceert dat er minder borden in de priorityqueue zitten en dus ook minder geheugen nodig heeft.

Efficiënt algoritme