Schakelprogramma: Master in de toegepaste informatica

Gegevensstructuren en algoritmen

Practicum 2

Dries Janse
r0627054
30 april 2019

Inhoud

1. Inleiding

Dit verslag is geschreven naar aanleiding van het tweede practicum van het opleidingsonderdeel: Gegevensstructuren en algoritmen. Dit als onderdeel van het schakelprogramma: Master in de toegepaste informatica.

In dit practicum wordt mijn oplossing van de solver voor het 8-puzzel probleem besproken. Hierbij wordt gebruik gemaakt van het A* algoritme. In dit verslag leg ik de nadruk op de efficiëntie van de geïmplementeerde methoden en prioriteitsfuncties.

2. Puzzel oplossingen

Onderstaande tabel geeft voor elk van de opgesomde puzzels het minimum aantal verplaatsingen om de doelstand te bereiken weer. Ook geeft het de uitvoeringstijd, in seconden, van het A* algoritme voor de Hamming en de Manhattan prioriteitsfuncties. De vraagtekens in onderstaande tabel geven weer dat de oplossing niet binnen een redelijke tijd (<5 minuten) gevonden kon worden.

Puzzel	*Aantal verplaatsingen*	*Hamming (s)*	*Manhattan (s)*
puzzel28.txt	28	1.151	0.052
puzzel30.txt	30	2.600	0.088
puzzel32.txt	32	?	1.618
puzzel34.txt	34	?	0.371
puzzel36.txt	36	?	6.443
puzzel38.txt	38	?	7.132
puzzel40.txt	40	?	1.369
puzzel42.txt	42	?	26.979

Tabel 1: Puzzels met het aantal verplaatsingen en de uitvoertijd per prioriteitsfunctie.

3. Complexiteit prioriteitsfuncties

In dit onderdeel bespreek ik de complexiteit van de gebruikte implementaties van de Hamming en Manhattan prioriteitsfuncties. De complexiteit van deze algoritmes wordt uitgedrukt in het aantal array accessen voor willekeurige N x N borden (voor elke N groter dan 1).

3.1 Complexiteit Hamming prioriteitsfunctie

```
public int hamming() {
	int expectedValue = 1;
	int nbrIncorrect = 0;
	for (int row = 0; row < this.getSize(); row++) {
		for (int column = 0; column < this.getSize(); column++) {
			int val = this.getValue(row, column);
			if(val != 0 && val != expectedValue) {
				nbrIncorrect++;
			}
			expectedValue++;
		}
	}
	return nbrIncorrect;
}

private int getValue(int row, int column) {
	return this.getTiles()[row][column];
}
```

Code implementatie 1: Java implementatie van de Hamming prioriteitsfunctie.

In de bovenstaande code maak ik gebruik van een getValue(int row, int column) hulpfunctie, dewelke gebruik maakt van één array access. In de eigenlijke Hamming prioriteitsfunctie zien we onmiddellijk dat er maar op exact één plaats gebruik gemaakt wordt van deze array access. Deze wordt opgeroepen in twee for-loops, die elk N keer worden uitgevoerd. Hieruit kunnen we besluiten dat de tijdscomplexiteit $\sim N^2$ is.

3.1 Complexiteit Manhattan prioriteitsfunctie

```
public int manhattan() {
	int totalDistance = 0;
	for (int row = 0; row < this.getSize(); row++) {
		for (int column = 0; column < this.getSize(); column++) {
			int val = this.getValue(row, column);
			if(val != 0 ) {
				totalDistance += this.getDistance(row, column, val);
			}
		}
	}
	return totalDistance;
}

private int getDistance(int row, int column, int value) {
	int[] pos = this.getXYPosition(value);
	return Math.abs(row - pos[0]) + Math.abs(column - pos[1]);
}

private int[] getXYPosition(int value) {
	int row =    (int) Math.ceil((double) value/ (double) this.getSize()) -1;
	int column = (int) (value -1) % this.getSize();
	return new int[] {row,column};
}
```

Code implementatie 2: Java implementatie van de Manhattan prioriteitsfunctie.

Net zoals bij de implementatie van Hamming prioriteitsfunctie zien we hier één array access. Het resultaat van deze array access wordt opgeslagen in de variabele: 'val'. De andere hulpfuncties gebruiken geen array accessen uit de 2D tiles array. De tijdscomplexiteit komt dus ook hier neer op $\sim N^2$ array accessen uit de tiles array.

Strikt genomen maakt de getDistance methode gebruik van 2 array accessen van de array die gemaakt wordt door de getXYPosition method. Deze getXYPosition methode maakt op zichzelf ook weer gebruik van 2 array accessen omdat het een array van lengte 2 maakt. Als we deze functies mee in rekening brengen wordt er gebruik gemaakt van $\sim 5N^2$ array accessen.

4. Complexiteit en implementatie van isSolvable

In de opgave van deze methode werd ons onderstaande formule aangereikt:

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

Het resultaat van bovenstaande, groter dan of gelijk aan, operatie geeft aan of de puzzel oplosbaar is. De formule maakt gebruik van de functie p. Deze functie geeft, gegeven een bord en een waarde van een tegel, de positie terug.

In mijn implementatie maak ik gebruik van een “lookup array”. Deze array heeft als index de waarde van de tegel. Op elk van deze indices staan de posities van de waarden. Door gebruik te maken van deze “lookup array” maak ik geen gebruik van een methode p die over alle data zou lopen tot het de gevraagde waarden gevonden heeft.

```
public boolean isSolvable() {
	int[][] emptyTileEnd = this.getEmptyTileAtEndBoard();
	int[] lookupHelper = this.lookupArray(emptyTileEnd);
	double subresult =1;
	for (int j = 1; j < (this.getSize()*this.getSize()); j++) {
		for (int i = 1; i < j; i++) {
			double numerator   = (lookupHelper[j] -lookupHelper[i]);
			double denominator = (j-i);
			subresult *= (numerator/denominator);
		}
	}
	return subresult>=0;
}
```

Code implementatie 3: isSolvable() implementatie.

In bovenstaande implementatie maken we in de eerste twee lijnen code gebruik van twee hulpfuncties waarvan de complexiteit later besproken wordt. In de binnenste for-loop zien we onmiddellijk het gebruik van twee array accessen: lookupHelper[j] en lookupHelper[i]. Hoe vaak wordt dit stuk code uitgevoerd?

- De variabele ‘j’ loopt van 1 tot en met $N^2 - 1$
- De variabele ‘i’ loopt van 1 tot en met $j - 1$

Door dit als een som te schrijven kunnen we gemakkelijk de complexiteit bepalen:

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

Deze berekening geeft weer hoe vaak de statements binnen de tweede for-loop uitgevoerd worden. Aangezien hier twee array accessen instaan moeten we bovenstaande formule vermenigvuldigen met twee. Dit komt neer op:

$$2 * \frac{(N^2 - 2)(N^2 - 1)}{2} = N^4 - N^2 - 2N^2 + 2 = N^4 - 3N^2 + 2$$

Vervolgens bespreken we de complexiteit van de gebruikte hulpfuncties.

```
private int[] lookupArray(int[][] tiles) {
	int[] lookup = new int[tiles.length*tiles.length];
	for (int i = 0; i < tiles.length; i++) {
		for (int j = 0; j < tiles.length; j++) {
			lookup[tiles[i][j]] = (i*tiles.length) + j + 1;
		}
	}
	return lookup;
}
```

Code implementatie 4: lookupArray(int[][] tiles) implementatie.

Bovenstaande lookupArray methode maakt gebruik van N^2 array accessen om de array te initialiseren. Vervolgens gaat deze array gevuld worden met de juiste posities, dit is ook goed voor N^2 array accessen. In totaal geeft dit $2N^2$ array accessen voor bovenstaande methode.

```
private int[][] swap(int frow, int fcol, int srow, int scol, int[][] tiles){
	int temp = tiles[srow][scol];
	tiles[srow][scol] = tiles[frow][fcol];
	tiles[frow][fcol] = temp;
	return tiles;
}

private int[] getPositionOfValue(int value, int[][] tiles) {
	if(tiles == null) {
		throw new IllegalArgumentException("Tiles cannot be null");
	}
	int size = tiles.length;
	for (int row = 0; row < size; row++) {
		for (int column = 0; column < size; column++) {
			if(this.getValue(row, column) == value) {
				return new int[] {row,column};
			}
		}
	}
	return null;
}
```

Code implementatie 5: swap en getPositionValue implementatie.

In de “swap” methode wordt er gebruik gemaakt van exact 4 array accessen. Bij de “getPositionOfValue” methode is dit iets moeilijker. In het worst-case geval gaat deze alle elementen moeten overlopen. Hier bovenop komen nog eens 2 array accessen om de array te declareren. Dit resulteert dus in $N^2 + 2$ array accessen.

```
public int[][] getEmptyTileAtEndBoard(){
	int[] pos = this.getPositionOfValue(0,this.getTiles());
	int rowMoves = this.getSize() - pos[0] -1;
	int colMoves = this.getSize() - pos[1] -1;
	int[][] currentTiles = this.getDeepCopy(this.getTiles());
	for (int i = 0; i < colMoves; i++) {
		currentTiles = this.swapTiles(pos[0], pos[1]+i, MoveDirection.RIGHT,
		currentTiles);
	}
	for (int i = 0; i < rowMoves; i++) {
		currentTiles = this.swapTiles(pos[0]+i, pos[1]+colMoves,
		MoveDirection.BOTTOM, currentTiles);
	}
	return currentTiles;
}
```

Code implementatie 6: getEmptyTileAtEndBoard() implementatie.

Tot slot de laatste hulpfunctie; getEmptyTileAtEndBoard(). Deze functie gaat het lege (0) vakje verplaatsen naar de rechteronderhoek. Het worst-case geval is hier, als het lege vakje zich in de linkerbovenhoek bevindt. Dan heeft dit $2N - 2$ swaps nodig om het lege vakje te verplaatsen.

Dit resulteert dus in:

$$(\text{"getPositionOfValue"})+(2*\text{"}pos\text{"})+(getDeepCopy)+(\text{"pos"} * \text{"swapTiles"} * (2N - 2))$$

$$= (N^2 + 2) + (2) + (2N^2) + \big(2 * 4 * (2N - 2)\big) = 3N^2 + 16N - 14$$

Nu de complexiteit van alle hulfuncties bepaald zijn, keren we terug naar de isSolvable() methode, code implementatie 3. De totale complexiteit samengesteld:

$$C(getEmptyTileAtEndBoard) + C(lookupArray) + C(isSolvableLoops)$$

$$= (3N^2 + 16N - 14) + (2N^2) + (N^4 - 3N^2 + 2) = \sim N^4$$

Hieruit kan het besluit genomen worden dat de worst-case complexiteit van de isSolvable methode $\sim N^4$ bedraagt.

5. Worst-case geheugen

Het is makkelijk in te zien dat er $N^2!$ bordopstellingen mogelijk zijn, dit is het aantal permutaties. Maar niet al deze bordopstellingen zijn oplosbaar. Er zijn $\frac{N^2!}{2}$ oplosbare bordopstellingen. In het worst-case scenario kunnen er dus $\frac{N^2!}{2}$ verschillende bordopstellingen in het geheugen staan.

6. Worst-case complexiteit bord toevoegen/verwijderen aan prioriteitsrij

Wat is bij mijn implementatie de worst-case complexiteit om een bord configuratie toe te voegen aan de prioriteitsrij?

In mijn oplossing maak ik gebruik van de built in PriorityQueue<> klasse van java, deze is gebaseerd op een priority heap. We weten dat de complexiteit bij het toevoegen van een element aan een prioriteitsrij gelijk is aan $\sim \log_2(N)$. Met andere woorden wil dit zeggen dat er worst-case $\sim \log_2(N)$ borden vergeleken moeten worden. Maar uit de vorige vraag hebben we geleerd dat er worst-case $\frac{N^2!}{2}$ verschillende borden in de prioriteitsrij kunnen zitten. Wat dus neer komt op worst-case $\sim \log_2(\frac{N^2!}{2})$ vergelijkingen.

Net zoals bij de vorige vragen willen we de tijdscomplexiteit uitdrukken in het aantal array accessen. Dit is afhankelijk welke prioriteitsfunctie er gebruik wordt. Om twee borden te vergelijken heb ik eigen klassen geschreven die de Comparator interface implementeert. Een Hamming Comparator die gebruik maakt om van de Hamming prioriteitsfunctie en een Manhattan Comparator die gebruik maakt van de Manhattan prioriteitsfunctie.

```
public class HammingComparator implements Comparator<BoardState> {

	/**
	  * Compare 2 BoardStates using the Hamming priority function of the
	  * current boards with the current boards number of moves.
	  * {@inheritDoc}
	  */
	@Override
	public int compare(BoardState state1, BoardState state2) {
	return (state1.getCurrentBoardHamming() + state1.getNbrMoves()) -
	(state2.getCurrentBoardHamming() + state2.getNbrMoves());
	}

}
```

Code implementatie 7: HammingComparator implementatie.

De Hamming prioriteitsfunctie gebruikt N^2 array accessen. In bovenstaande compare methode wordt deze tweemaal opgeroepen. Er zijn dus $2N^2$ array accessen per vergelijking. De worst-case complexiteit van het toevoegen van een element, gebruik makende van Hamming prioriteitsfunctie, is:

$$\sim 2N^2 \log_2(\tfrac{N^2!}{2}) = \sim 2N^2 \log_2(N^2!) - 2N^2 = \sim 2N^4 \log_2(N^2) = \sim 4N^4 \log_2(N)$$

```java
public class ManhattanComparator implements Comparator<BoardState> {

	/**
	* Compare 2 BoardStates using the Manhattan priority function of the
	* current board with the current boards number of moves.
	* {@inheritDoc}
	*/
	@Override
	public int compare(BoardState state1, BoardState state2) {
		return (state1.getCurrentBoardManhattan() + state1.getNbrMoves()) -
		(state2.getCurrentBoardManhattan() + state2.getNbrMoves());
	}
}
```

Code implementatie 8: ManhattanComparator implementatie.

Er kan ook gebruik gemaakt worden van de ManhattanComparator, dewelke gelijkaardig is aan HammingComparator, maar deze gebruikt de Manhattan prioriteitsfunctie. Deze prioriteitsfunctie maakt gebruikt van $\sim 5N^2$ array accessen per oproep. Ook hier berekenen we de complexiteit van het toevoegen van een element, gebruik makende van de Manhattan prioriteitsfunctie.

$$\sim 5N^2 \log_2(\frac{N^2!}{2}) = \sim 5N^2 \log_2(N^2!) - 5N^2 = \sim 5N^4 \log_2(N^2) = \sim 10N^4 \log_2(N)$$

Voor het verwijderen van een element uit een prioriteitsrij gaan er worst-case $2\log_2(N)$ vergeleken worden. Het enige verschil met het toevoegen van een element is deze factor 2. Vervolgens geven we de complexiteit van het verwijderen van een element:

- Met Hamming prioriteitsfunctie: $\sim 8N^4 \log_2(N)$
- Met Manhattan prioriteitsfunctie: $\sim 20N^4 \log_2(N)$

7. Betere prioriteitsfuncties

Er kunnen betere prioriteitsfuncties gevonden worden. We denken hier aan heuristiek functies zoals: “Linear Conflict”, “Pattern Database”, “Gasching’s heuristic”, etc.

Graag ga ik dieper in op de berekening van de heuristiek die gebruik maakt van “linear conflict”. Deze heuristiek werd uitgevonden door Hanson, Mayer en Yung in 1992. Het is gebaseerd op de berekening van de Manhattan heuristiek. Twee elementen zijn in conflict met elkaar als ze beide in dezelfde kolom/rij zitten en hun eindpositie is ook in deze kolom/rij. Dan is hun heuristiek de Manhattan heuristiek vermeerderd met twee extra stappen. Bijvoorbeeld, de bovenste rij bevat de tegels (2 1), in deze volgorde. Om deze om te wisselen moet één tegel naar beneden worden verschoven om de andere tegel naar zijn doel positie te laten verschuiven.

Voor deze prioriteitsfunctie moeten we dus meer berekeningen maken dan bij de originele Manhattan functie. Maar doordat de prioriteit accurater berekend is gaan we sneller naar het doel gaan. Hierdoor is er minder geheugen nodig is.

8. Meer geheugen/tijd of betere prioriteitsfunctie?

Er kan beter geopteerd worden voor een betere prioriteitsfunctie. Dit kan zeer snel gegrond worden uit tabel 1. In deze tabel zien we duidelijk dat de Manhattan prioriteitsfunctie veel efficiënter is dan de Hamming prioriteitsfunctie. Bij een minder goede prioriteitsfunctie neemt de geheugen inname en de uitvoer tijd enorm snel toe, zoals bewezen uit de experimenten.

9. Efficiënt algoritme

Ik denk niet dat er een efficiënt algoritme bestaat die het probleem kan oplossen binnen praktische tijd. Het is bewezen dat dit probleem NP-hard is. Hieruit kunnen we stellen dat het makkelijk te controleren is met een oplossing, maar het is moeilijk om een oplossing te vinden.

10. Bronnen

Richard E. Korf and Larry A. Taylor. (1996). Finding Optimal Solutions to the Twenty-Four Puzzle. Artificial Intelligence (AAAI), p.1202-1207