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 < this.tiles.length; i++){
                for (int j = 0; j < this.tiles[i].length; 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 $\sim 2 \cdot N^2$.
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

isSolvable

Borden in geheugen

Worst-case tijdscomplexiteit

Betere prioriteitsfuncties

Tijd, geheugen of algoritme?

Efficiënt algoritme