

# 7


## Breadth First Search Algoritme



### Inleiding

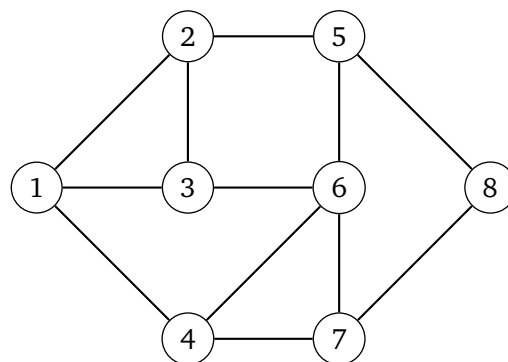
Download het bestand `week07-Grafen-BFS-opgave.zip` van Toledo. Importeer dit bestand in je IDE.

In deze oefenzitting leer je het Breadth First Search Algoritme (BFS) te implementeren in Java.

### Oefening 7.1


 In de cursustekst wordt BFS visueel uitgelegd. De knopen worden één voor één ingekleurd en de reeds bezochte knopen worden opgelijst in een boomstructuur. Gebruik deze methode om onderstaande oefening op te lossen. Zo krijg je inzicht hoe BFS werkt.

- a) Stel de verbindingsmatrix op van de getekende graaf. 
- b) Zoek het pad van het eerste naar het laatste knooppunt dat zo weinig mogelijk knooppunten telt door gebruik te maken van het breadth-first algoritme. 



**Figuur 7.1** Niet-gerichte graaf met 8 knooppunten



## Oefening 7.2

 Los dezelfde oefening nu op zoals uitgelegd in het schema op bladzijde 11 van de theorie. Teken de ancestors-matrix  $A$  en de queue  $Q$ . Simuleer het algoritme op papier. Dit zal je helpen om in wat volgt het algoritme in Java te implementeren.

## Oefening 7.3



Bestudeer de klasse `Graph` die je in IntelliJ importeerde. Een graaf heeft slechts één instantieveranderlijke, namelijk de verbindingsmatrix.

- De methode `isGeldigeVerbindingsmatrix(int[][] matrix)` controleert of `matrix` een geldige verbindingsmatrix is. Welke voorwaarden worden getest? 
- De instantieveranderlijke is een matrix van booleans, niet van integers. Waarom? 

## Oefening 7.4



We beginnen nu aan de implementatie van het BFS-algoritme. Houd de oplossing van oefening 7.3 bij de hand.

Bij BFS start je in een gegeven knoop. Je kijkt welke knopen allemaal verbonden zijn met deze startknoop. Dan neem je de eerste van die knopen en lijst je op wie met deze tweede knoop verbonden is enz. totdat je de eindknoop gevonden hebt. Dit resulteert in een array, de ancestors, waar je voor elke knoop kan aflezen wie zijn ‘voorganger’ is, of ‘ouder’ in de boomstructuur in afbeelding 1.8. In het schema blz. 11 is dit de array  $A$ .

Schrijf de methode `int[] findAncestors(int start, int destination)`. Invoer zijn start- en eindknoop. Uitvoer is de ancestors-array  $A$ , een array van integers.

De indices van  $A$  refereren naar (het nummer van) een knoop, maar zijn niet gelijk. Als je de voorganger van knoop met nummer  $i$  wil kennen, moet je in  $A$  de inhoud van het element met index  $i - 1$  opvragen. We behouden dit verschil omdat we in mensentaal niet spreken over het ‘nulde’ element, maar wel van het ‘eerste’ element.

We initialiseerden reeds alle elementen van  $A$  op `infty`. Zolang de waarde van een element op `infty` blijft staan, is de bijhorende knoop nog niet bezocht.

Om bij te houden welke knoop er onderzocht moet worden, gebruiken we de queue  $Q$ . In Java bestaat de Queue-datastructuur. Zoek op <https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html> hoe je elementen toevoegt en uitleest uit  $Q$ .

In het bestand `BreadthFirstSearchUI` vind je in de `main`-methode de verbindingsmatrix van het uitgewerkte voorbeeld in de cursustekst (figuur 1.5)

Verwachte uitvoer voor het voorbeeld uit de theorie:

Ancestors van 1 naar 7:

0 1 2 1 4 4 5

Ancestors van 7 naar 1:

infty infty infty infty infty infty 0

## Oefening 7.5



Schrijf nu de methode `List<Integer> findPath(int start, int destination)` die het gezochte pad berekent tussen start en destination.

Bepaal het pad van achter naar voren. Dit wil zeggen dat je start bij de eindknoop, zijn voorganger zoekt en zo opbouwt totdat je de startknoop vindt.

Verwachte uitvoer voor het voorbeeld uit de theorie:

Kortste pad van 1 naar 7 is 4 knopen lang en bestaat uit volgende knopen : [1, 4, 5, 7]

Er is geen pad van 7 naar 1

## Oefening 7.6



Voeg in de main methode de verbindingsmatrix van de graaf uit de eerste oefeningen (figuur 7.1 op pagina 41) toe en test je code met enkele verschillende start- en eindpunten in deze graaf.



# Oplossingen

**Oplossing 7.1**  $1 \rightarrow 2 \rightarrow 5 \rightarrow 8$  of  $1 \rightarrow 4 \rightarrow 7 \rightarrow 8$

## Oplossing 7.3

1. Een verbindingsmatrix is correct indien
  - het aantal rijen gelijk is aan het aantal kolommen
  - de diagonaal van linksboven naar rechtsonder overal nul is
  - alle elementen gelijk zijn aan 0 of 1
2. De waarden van de verbindingsmatrix kunnen slechts gelijk zijn aan 0 en 1. Het vraagt minder geheugenruimte en het werkt efficiënter als de elementen als een boolean voorgesteld worden.

## Oplossing 7.4

**Listing 2** findAncestors(int,int)

```
private boolean rechtstreekseVerbinding(int van, int tot) {
    //System.out.println("verbinding van "+van+" tot "+tot+"?");
    return this.getVerbindingsMatrix()[van - 1][tot - 1];
}

private int[] findAncestors(int start, int destination) {
    int aantalKnopen = this.getAantalKnopen();
    int[] ancestors = new int[aantalKnopen];
    initArray(ancestors, infty);

    Queue<Integer> queue = new LinkedList<>();
    queue.add(start);
    ancestors[start - 1] = 0;

    int huidig = queue.remove();
    while (huidig != destination) {
        //System.out.println("huidig = "+huidig);
        //zoek alle nog niet bezochte knooppunten vanuit huidig
        for (int i = 1; i <= aantalKnopen; i++) {
            if (rechtstreekseVerbinding(huidig, i) && (ancestors[i - 1] == infty)) {
                //System.out.println("ja");
                //voeg knoop i toe aan queue
                queue.add(i);

                //duid aan dat huidig de ouder is van i in ancestormatrix
                ancestors[i - 1] = huidig;
            }
        }
    }
}
```

## Oplossingen

```
//voorst element van queue wordt nieuwe huidige knoop
if (!queue.isEmpty()) {
    huidig = queue.remove(); //of .poll() wat geen exception gooit
} else {
    //queue is leeg, stop maar
    break;
}

}
return ancestors;
}

public boolean[][] getVerbindingsMatrix() {
    return verbindingsMatrix;
}

private void initArray(int[] array, int value) {
    for (int i = 0; i < array.length; i++)
        array[i] = value;
}
```

## Oplossing 7.5

### Listing 3 findPath

```
public List<Integer> findPath(int start, int destination) {
    if (start <= 0 || start > this.getAantalKnopen() || destination <= 0 ||
        destination > this.getAantalKnopen())
        throw new IllegalArgumentException();

    int[] ancestors = this.findAncestors(start, destination);
    List<Integer> path = new LinkedList<>();

    int ouder = ancestors[destination - 1];
    while (ouder != 0 && ouder != infty) {
        path.add(0, destination);
        destination = ouder;
        ouder = ancestors[destination - 1];
    }
    if (ouder == 0) {
        path.add(0, destination);
    }
    return path;
}
```