1

Breadth First Search Algoritme

Inleiding

Download het bestand `BG_Les8.zip` van Toledo. Importeer dit bestand in Eclipse.

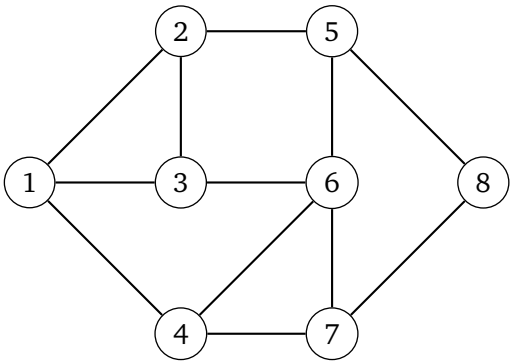
In deze oefenzitting leer je het Breadth First Search Algoritme (BFS) te implementeren in Java.

Oefening 1.1

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 1.1 Niet-gerichte graaf met 8 knooppunten

Oefening 1.2

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 1.3

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

a) De methode `isGeldigeVerbindingsmatrix(int[][] matrix)` controleert of `matrix` een geldige verbindingsmatrix is. Welke voorwaarden worden getest?

b) De instantieveranderlijke is een matrix van booleans, niet van integers. Waarom?

Oefening 1.4

We beginnen nu aan de implementatie van het BFS-algoritme. Houd de oplossing van oefening 1.3 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 1.5

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 1.6

Voeg in de `main` methode de verbindingsmatrix van de graaf uit de eerste oefeningen (figuur 1.1) toe en test je code met enkele verschillende start- en eindpunten in deze graaf.