1

Breadth First Search Algoritme

Inleiding

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

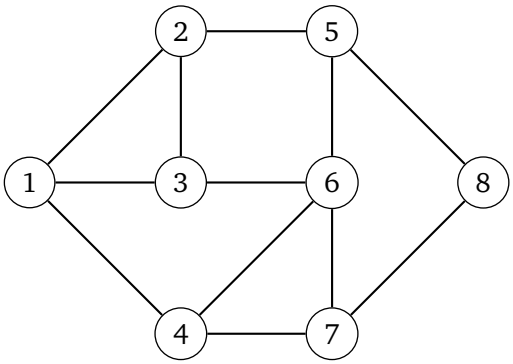
In deze oefenzitting leer je het Breadth First Search Algoritme (BFS) te implementeren in Java.

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.

2 Floyd

Inleiding

Download het bestand `BG_Les9.zip` van Toledo. Importeer dit bestand in Eclipse.

In deze oefenzitting leer je het algoritme van Floyd te implementeren in Java.

De eerste oefening is op papier. Hiermee leer je het algoritme goed begrijpen. Verder kan je deze papieren versie gebruiken om je zelfgeschreven code te testen. Houd de oplossing dus goed bij!

Oefening 2.1

Maak deze oefening op papier.

Een graaf bestaat uit 6 knooppunten en wordt gegeven door zijn gewichtenmatrix

$$D^{(0)} = \begin{bmatrix} 0 & 7 & 3 & \infty & 5 & \infty \\ 2 & 0 & \infty & 12 & \infty & \infty \\ \infty & 3 & 0 & \infty & 1 & \infty \\ \infty & \infty & \infty & 0 & \infty & 6 \\ 5 & \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & 1 & \infty & 0 \end{bmatrix}$$

1. Teken het netwerk.
2. Gebruik de methode van Floyd om de kortste paden tussen de verschillende punten van het netwerk te berekenen.

Oefening 2.2

Bestudeer de klasse `WeightedGraph` die je in Eclipse importeerde.

1. In tegenstelling tot de klasse `Graph` die we schreven bij BFS, is de instantieveranderlijke nu een matrix van `double` en niet van `boolean`. Waarom?

2. Wanneer is een gewichtenmatrix geldig?

Oefening 2.3

Schrijf de methode `int[][] findDistances()`. Deze methode berekent de pointermatrix P.

- De (lege) pointermatrix wordt gedeclareerd.
- Er wordt een kloon van de gewichtenmatrix gemaakt. Dit is een kopie die verwijst naar een andere geheugenplaats. Deze kloon is de D-matrix van de cursustekst.
- Voeg de knopen één voor één toe als tussenstation en pas de D-matrix aan zoals aangegeven in formule (2.3). Update de pointermatrix.

Verwachte uitvoer:

```
p_matrix:
0 0 4 0 4
5 0 0 0 4
5 5 0 0 4
5 5 0 0 0
0 1 4 1 0
```

Oefening 2.4

Schrijf de methode `ArrayList<Integer> getShortestPath(int i, int j, int[][] path)`. Deze methode berekent uit de pointermatrix `path` het kortste pad tussen knoop `i` en knoop `j`.

In deze oefening geldt dezelfde opmerking over de nummering van de knopen als we maakten in oefening 1.4. Het knooppunt dat laatst toegevoegd werd als tussenstation bij de berekening van het kortste pad tussen knooppunten `i` en `j`, is `path[i - 1][j - 1]`.

(Deel van) de verwachte uitvoer:

```
Kortste paden:
Er is geen pad van 1 naar 1
Kortste pad van 1 naar 2 lengte = 0 via : [1, 2]
Kortste pad van 1 naar 3 lengte = 0 via : [1, 4, 3]
Kortste pad van 1 naar 4 lengte = 0 via : [1, 4]
Kortste pad van 1 naar 5 lengte = 0 via : [1, 4, 5]

Kortste pad van 2 naar 1 lengte = 0 via : [2, 4, 5, 1]
Er is geen pad van 2 naar 2
Kortste pad van 2 naar 3 lengte = 0 via : [2, 3]
```

```
Kortste pad van 2 naar 4 lengte = 0 via : [2, 4]
Kortste pad van 2 naar 5 lengte = 0 via : [2, 4, 5]

Kortste pad van 3 naar 1 lengte = 0 via : [3, 4, 5, 1]
Kortste pad van 3 naar 2 lengte = 0 via : [3, 4, 5, 1, 2]
Er is geen pad van 3 naar 3
Kortste pad van 3 naar 4 lengte = 0 via : [3, 4]
Kortste pad van 3 naar 5 lengte = 0 via : [3, 4, 5]
```

Oefening 2.5

Schrijf de methode `int berekenLengte(ArrayList<Integer> pad)`. Invoer is `pad`: een opeenvolging van knopen die resulteert in het kortste pad tussen eerste en laatste element van `pad`. Uitvoer is de lengte van dit pad. Deze wordt berekend met behulp van de instantievariabele `gewichtenmatrix`. (Deel van) de verwachte uitvoer:

```
Kortste paden:
Er is geen pad van 1 naar 1
Kortste pad van 1 naar 2 lengte = 1 via : [1, 2]
Kortste pad van 1 naar 3 lengte = 3 via : [1, 4, 3]
Kortste pad van 1 naar 4 lengte = 1 via : [1, 4]
Kortste pad van 1 naar 5 lengte = 4 via : [1, 4, 5]

Kortste pad van 2 naar 1 lengte = 8 via : [2, 4, 5, 1]
Er is geen pad van 2 naar 2
Kortste pad van 2 naar 3 lengte = 3 via : [2, 3]
Kortste pad van 2 naar 4 lengte = 2 via : [2, 4]
Kortste pad van 2 naar 5 lengte = 5 via : [2, 4, 5]

Kortste pad van 3 naar 1 lengte = 10 via : [3, 4, 5, 1]
Kortste pad van 3 naar 2 lengte = 11 via : [3, 4, 5, 1, 2]
Er is geen pad van 3 naar 3
Kortste pad van 3 naar 4 lengte = 4 via : [3, 4]
Kortste pad van 3 naar 5 lengte = 7 via : [3, 4, 5]

Kortste pad van 4 naar 1 lengte = 6 via : [4, 5, 1]
Kortste pad van 4 naar 2 lengte = 7 via : [4, 5, 1, 2]
Kortste pad van 4 naar 3 lengte = 2 via : [4, 3]
Er is geen pad van 4 naar 4
Kortste pad van 4 naar 5 lengte = 3 via : [4, 5]

Kortste pad van 5 naar 1 lengte = 3 via : [5, 1]
Kortste pad van 5 naar 2 lengte = 4 via : [5, 1, 2]
```

```
Kortste pad van 5 naar 3 lengte = 6 via : [5, 1, 4, 3]
Kortste pad van 5 naar 4 lengte = 4 via : [5, 1, 4]
Er is geen pad van 5 naar 5
```

3

Algoritme van Dijkstra

Inleiding

Download het bestand `BG_Les10.zip` van Toledo. Importeer dit bestand in Eclipse.

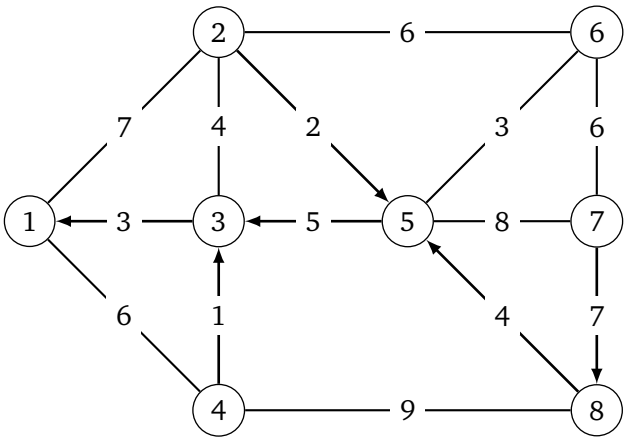
In deze oefenzitting leer je het algoritme van Dijkstra te implementeren in Java.

De eerste oefening is op papier. Hiermee leer je het algoritme goed begrijpen. Verder kan je deze papieren versie gebruiken om je zelfgeschreven code te testen. Houd de oplossing dus goed bij!

Oefening 3.1

Figuur 3.1 toont een netwerk. De aangegeven getallen bij elke verbinding zijn het aantal km tussen beide knooppunten. Stel een lijst op met de kortste afstanden vanuit *knooppunt 3* naar elk ander punt. Geef ook telkens de kortste route aan. Gebruik de methode van Dijkstra.

1. Pas het algoritme grafisch toe.
2. Gebruik de tabelmethode.



Figuur 3.1 Netwerk bij oefening 3.1

Oefening 3.2

In tegenstelling tot het algoritme van Floyd en BFS, is het algoritme van Dijkstra zoals je het vindt in de cursus niet 'reeds klaar voor Java-gebruik'. In deze eerste oefening maken we daarom de vertaling van het algoritme naar een alternatieve tabelmethode die geïmplementeerd kan worden in Java.

Lees het document 'oefeningAlternatieveTabelMethodeVoorDijkstra.pdf'. Los nu oefening 3.1 nogmaals op papier op, maar nu met deze methode.

Oefening 3.3

Je bent eindelijk klaar met het papierwerk. Je kan nu beginnen met het opzetten van de klasse `Graph`.

Schrijf de methode `int[][] initMatrixDijkstra(int vanKnoop)`. Deze methode modelleert de gewichtenmatrix zoals uitgelegd in stap 1 en stap 2 van het document 'oefeningAlternatieveTabelMethodeVoorDijkstra.pdf'. Let op de indices die je gebruikt!

- Er wordt een `int[][]` aangemaakt die evenveel kolommen telt als de gewichtenmatrix, maar één extra rij.
- Om aan te geven dat de elementen van de eerste rij (index 0) leeg zijn, stellen we ze gelijk aan `Integer.MAX_VALUE`.
- De overige elementen krijgen de corresponderende waarde van de gewichtenmatrix waarbij `infinity` vervangen wordt door 0.
- Zet in de kolom die overeenkomt met de startknoop nullen.

Verwachte uitvoer bij het voorbeeld uit de cursustekst, waar een extra knoop 9 toegevoegd is cfr. main methode `UIDijkstra`:

```
Initiele matrix:

0       inf     inf     inf     inf     inf     inf     inf     inf
0       5       9       0       0       0       0       0       0
0       0       3       8       10      11      0       0       0
0       3       0       2       0       0       7       0       0
0       8       2       0       0       3       7       0       0
0       10      0       0       0       1       0       8       0
0       0       0       3       1       0       5       10      0
0       0       7       7       0       0       0       12      0
0       0       0       0       8       10      12      0       0
0       0       0       0       0       0       0       0       0
```

Oefening 3.4

Schrijf de methode `int[][] Dijkstra(int vanKnoop)`. Deze methode implementeert het algoritme van Dijkstra. Ze geeft de aangepaste gewichtenmatrix terug, met extra rij bovenaan met de kleinste afstanden.

Verwachte uitvoer:

```
Resulterende matrix:

0 5 8 10  14  13  15  22  inf
0 5 0 0 0 0 0 0 0
0 0 3 0 0 0 0 0 0
0 0 0 2 0 0 7 0 0
0 0 0 0 0 3 0 0 0
0 0 0 0 0 0 0 8 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

Oefening 3.5

Schrijf de methode `ArrayList<Integer> vindPad(int vanKnoop, int naarKnoop, int[][] res)` die het kortste pad van `vanKnoop` naar `naarKnoop` berekent. De veranderlijke `int[][] res` is de aangepaste gewichtenmatrix zoals die teruggegeven wordt door de methode `Dijkstra`.

Verwachte uitvoer:

```
Kortste afstand van 1 naar 2 = 5
via [1, 2]
Kortste afstand van 1 naar 3 = 8
via [1, 2, 3]
Kortste afstand van 1 naar 4 = 10
via [1, 2, 3, 4]
Kortste afstand van 1 naar 5 = 14
via [1, 2, 3, 4, 6, 5]
Kortste afstand van 1 naar 6 = 13
via [1, 2, 3, 4, 6]
Kortste afstand van 1 naar 7 = 15
via [1, 2, 3, 7]
Kortste afstand van 1 naar 8 = 22
via [1, 2, 3, 4, 6, 5, 8]
Er is geen pad van 1 naar 9
```

Oplossingen

Oplossing 1.1 $1 \rightarrow 2 \rightarrow 5 \rightarrow 8$ of $1 \rightarrow 4 \rightarrow 7 \rightarrow 8$

Oplossing 1.3

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 1.4

Listing 1 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;
      }
    }
```

```
    //voorste 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 1.5

Listing 2 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;
}
```

Oplossing 2.1 Gewichtenmatrix $D^{(6)}$ en bijhorende pointermatrix P:

$$D^{(6)} = \begin{bmatrix} 0 & 6 & 3 & 7 & 4 & 6 \\ 2 & 0 & 5 & 9 & 6 & 8 \\ 5 & 3 & 0 & 4 & 1 & 3 \\ \infty & \infty & \infty & 0 & \infty & 6 \\ 5 & 11 & 8 & 3 & 0 & 2 \\ \infty & \infty & \infty & 1 & \infty & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 3 & 0 & 6 & 3 & 5 \\ 0 & 0 & 1 & 6 & 3 & 5 \\ 2 & 0 & 0 & 6 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 1 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Enkele paden: $2 \to 1 \to 3 \to 5 \to 6$ met lengte 8; $5 \to 1 \to 3 \to 2 \to 4$ met lengte 3

Oplossing 2.2

1. Bij het algoritme van Floyd berekenen we paden met minste gewicht. We hebben daarom de gewichtenmatrix nodig. De elementen van die gewichtenmatrix zijn niet meer binair (0 of 1), maar kunnen alle positieve waarden aannemen.
2. Een gewichtenmatrix is geldig als er elementen in zitten en als hij vierkant is.

Oplossing 2.3

Listing 3 findDistances

```
public int[][] findDistances() {
  int aantal = this.gewichtenMatrix.length;
  int[][] P = new int[aantal][aantal];
  //double[][] D = this.gewichtenMatrix.clone(); fout = shallow clone
  //http://stackoverflow.com/questions/9106131/how-to-clone-a-multidimensional-array-in-java
  //of manuele versie in de nieuwe opgave op toledo
  //argument voor deze clone: is gezien in OOP
  double[][] D = this.gewichtenMatrix.clone();
  for (int i = 0; i < D.length; i++) {
      D[i] = D[i].clone();
  }

  for (int k = 0; k < aantal; k++) {
    for (int i = 0; i < aantal; i++) {
      for (int j = 0; j < aantal; j++) {
        if (D[i][k] + D[k][j] < D[i][j]) {
          D[i][j] = D[i][k] + D[k][j];
          P[i][j] = k + 1;
        }
      }
    }
  }

  return P;
}
```

Oplossing 2.4

Listing 4 getShortestPath

```
public List<Integer> getShortestPath(int van, int tot, int[][] P) {
  List<Integer> pad = new ArrayList<>();
  if (van == tot) {
    return pad;
  } else {
    int via = P[van - 1][tot - 1];
    if (via == 0){
      pad.add(van);
      pad.add(tot);
    } else {
      pad = getShortestPath(van, via, P);
      pad.remove(pad.size() - 1); //anders dubbel
      pad.addAll(getShortestPath(via, tot, P));
    }
  }

  return pad;
}
```

Oplossing 2.5

Listing 5 berekenLengte

```
public int berekenLengte(List<Integer> pad) {
  int som = 0;
  int aantalKnopen = pad.size();
  int huidigeKnoop, volgendeKnoop;

  for (int i = 0; i < aantalKnopen - 1; i++) {
    huidigeKnoop = pad.get(i);
    volgendeKnoop = pad.get(i + 1);
    som += this.gewichtenMatrix[huidigeKnoop - 1][volgendeKnoop - 1];
  }

  return som;
}
```

Oplossing 3.1

Stad	Kortste afstand vanuit 3	Route
1	3	$3 \rightarrow 1$
2	4	$3 \rightarrow 2$
3	0	
4	9	$3 \rightarrow 1 \rightarrow 4$
5	6	$3 \rightarrow 2 \rightarrow 5$
6	9	$3 \rightarrow 2 \rightarrow 5 \rightarrow 6$
7	14	$3 \rightarrow 2 \rightarrow 5 \rightarrow 7$
8	18	$3 \rightarrow 1 \rightarrow 4 \rightarrow 8$

Oplossing 3.4

Listing 6 Dijkstra

```
public int[][] Dijkstra(int vanKnoop) {
  int[][] res = initMatrixDijkstra(vanKnoop);

  System.out.println("Initiele matrix: \n");
  printIntMatrix(res);

  boolean ok = false;
  while (!ok) {
    int indexKleinsteJ = 0;
    int indexKleinsteI = 0;
    int kleinste = Integer.MAX_VALUE;

    for (int i = 0; i < this.gewichtenMatrix.length; i++) {
      if (res[0][i] != Integer.MAX_VALUE) {
        // doorzoek alle knopen waar nog geen kortste pad voor werd
        // gevonden en die bereikbaar zijn uit knopen waar reeds korste pad voor
        // werd gevonden en zoek hierin de kleinste afstand
        for (int j = 0; j < this.gewichtenMatrix.length; j++) {
          if (res[i + 1][j] != 0 && res[0][j] == Integer.MAX_VALUE)
            if (res[0][i] + res[i + 1][j] < kleinste) {
              indexKleinsteJ = j;
              indexKleinsteI = i + 1;
              kleinste = res[0][i] + res[i + 1][j];
            }
        }
      }
    }
    if (kleinste == Integer.MAX_VALUE) {
      ok = true;
    } else {
      res[0][indexKleinsteJ] = kleinste;
      for (int i = 1; i <= this.gewichtenMatrix.length; i++)
        if (i != indexKleinsteI)
```

```
          res[i][indexKleinsteJ] = 0;
    }
  }

  return res;
}

private int[][] initMatrixDijkstra(int vanKnoop) {
  int[][] res = new int[this.gewichtenMatrix.length + 1][this.gewichtenMatrix.length];

  //initialiseer alle getallen op eerste rij nu op een grote waarde
  //omdat de kortste afstanden nog niet gevonden zijn
  for (int i = 0; i < this.gewichtenMatrix.length; i++)
    res[0][i] = Integer.MAX_VALUE;

  for (int i = 1; i <= this.gewichtenMatrix.length; i++) {
    for (int j = 0; j < this.gewichtenMatrix.length; j++) {
      if (this.gewichtenMatrix[i - 1][j] == Integer.MAX_VALUE)
        res[i][j] = 0;
      else
        res[i][j] = this.gewichtenMatrix[i - 1][j];
    }
  }

  for (int i = 0; i < this.gewichtenMatrix.length; i++) {
    res[i][vanKnoop - 1] = 0;
  }
  return res;
}
```

Oplossing 3.5

Listing 7 vindPad

```
private ArrayList<Integer> vindPad(int vanKnoop, int naarKnoop, int[][] res) {
  ArrayList<Integer> pad = new ArrayList<>();
  pad.add(naarKnoop);

  while (naarKnoop != vanKnoop) {
    int k = 1;
    while (k < res.length && res[k][naarKnoop - 1] == 0)
      k++;
    pad.add(0, k);
    naarKnoop = k;
  }
  return pad;
}
```