ADS - Assignment 4

PublicTransportMap

Door:

Burak Inan (500740123)

Safak Inan (500802749)

Klas: IS203

Docent: N. Tromp

Datum: 31/12/2019

Inhoudsopgave

Inleiding

In dit verslag wordt besproken hoe we de implementaties hebben toegepast op het PublicTransportMap project. De volgende onderwerpen komen aan bod:

- Builder
- Paths
- Graphs
- BreadthFirstSearch
- DepthFirstSearch
- Dijkstra's shortest path
- A*-search

Hoofdstuk 1: Transport graph

In dit hoofdstuk worden de implementaties in de klasse TransportGraph besproken. Deze klas is bedoeld om een gehele openbaar vervoer systeem te bouwen, aan de hand van Stations, Lines en Connections. Een Line is een metro of bus lijn, waarbij er meerdere Stations kunnen staan die met elkaar Connections hebben. Deze Line wordt geïdentificeerd door een kleur en een type. Een Stations kan op meerdere Lines staat, wat betekent dat je moet overstappen.Deze Stations kunnen we beschouwen als de vertexes van de TransportGraph. Een Connection is simpel gezegd een connectie tussen stations op een Line. Deze Connections zijn in principe de edges van de TransportGraph.

Add vertex

Deze methode voegt een Station toe aan de lijst van Stations in de TransportGraph. Daarnaast wordt deze station ook toegevoegd aan e map stationIndices. Deze map neemt als key de naam van de Station in een string en als value de index van de Station.

```
public void addVertex(Station vertex) {
    stationList.add(vertex);
    stationIndices.put(vertex.getStationName(), stationList.lastIndexOf(vertex));
}
```

Snippet 1: Allereerst wordt de Station toegevoegd aan de stationList. Vervolgens wordt deze Station toegevoegd aan de stationIndices map, met als key de naam van de Station en als value de index van e Station in de stationList.

Add edge (adjacency list)

Deze methode voegt een connectie toe aan de adjacencyLists. Daarbij telt hij ook het aantal connecties er 1 punt bij op.

```
private void addEdge(int from, int to) {
    numberOfConnections++;
    if (from >= 0 && to >= 0) {
        adjacencyLists[from].add(to);
    }
}
```

Snippet 2: Allereerst wordt er 1 punt bij de numberOfConnections opgeteld. Vervolgens wordt er een check gedaan op de meegegeven parameters from en to. Er mag pas een connectie worden toegevoegd aan de adjacencyLists als from en to een positieve waarde heeft.

Add edge (connections)

Deze methode maakt gebruik van de addEdge methode hierboven. Daarnaast wordt er ook een Connection object toegevoegd aan de lijst van connections in de TransportGrpah.

```
public void addEdge(Connection connection) {
    Station from = connection.getFrom();
    Station to = connection.getTo();

    int indexFrom = getIndexOfStationByName(from.getStationName());
    int indexTo = getIndexOfStationByName(to.getStationName());

    if (indexFrom >= 0 && indexTo >= 0) {
        connections[indexFrom][indexTo] = connection;
        addEdge(indexFrom, indexTo);
    }
}
```

Snippet 3: Eerst worden de namen van de Stations in de Connection opgehaald. Vervolgens zoeken we de index op van de Stations in de TransportGraph. Daarna wordt er een check gedaan of de indexes positieve waardes heeft. Deze Connection wordt dan toegevoegd aan de lijst van connections. Ook wordt de addEdge methode aangeroepen om de adjacencyLists en de numberOfConnections up te daten.

Get connection (index)

Deze methode haalt een Connection op, op basis van de meegegeven indexes.

```
public Connection getConnection(int from, int to) {
    if (from < 0 || to < 0) {
        return null;
    }
    return connections[from][to];
}
```

Snippet 4: Eerst wordt er een check gedaan of de indexes niet negatief zijn. Vervolgens wordt de Connection gereturned.

Get index of station

Deze methode returned de naam van de Station, afhankelijk van de meegegeven index.

```
public int getIndexOfStationByName(String stationName) {
    if (!stationIndices.containsKey(stationName)) {
        System.out.println(("Station: " + stationName + " does not exist"));
        return -1;
    }
    return stationIndices.get(stationName);
}
```

Snippet 5: Eerst wordt er een check gedaan of de stationName bestaat in de stationIndices map. Als dat niet het geval is, dan wordt er -1 gereturned, wat betekent dat de Station niet bestaat. Anders wordt er een index gereturned van de Station in de lijst.

Get all connections

Deze methode returned een lijst van Connections.

```
private List<Connection> getAllConnections() {
    List<Connection> connections = new ArrayList<>();
    for (Connection[] connection : this.connections) {
        for (int j = 0; j < connection.length; j++) {
            if (connection[j] != null) {
                connections.add(connection[j]);
            }
        }
    }

    return connections;
}
```

Snippet 6: Eerst instantiëren we een lijst genaamd connections. Vervolgens loopen we door de 2-dimensionale lijst van connections heen in TransportGraph. Hierin kijken we of er op plek connection[j] een Connection object bestaat. Als dat wel het geval is dan wordt deze toegevoegd aan de connections list in de methode. Uiteindelijk wordt deze list gereturned.

Get adjacent connections

Deze methode returned alle Connections in de TransportGraph, behalve de Connections met de meegegeven Station.

```
public List<Connection> getAdjacentConnections(Station from) {
    return getAllConnections().stream()
            .filter(x -> x.getFrom().equals(from))
            .collect(Collectors.toList());
}
```

Snippet 7: We streamen door de lijst van Connections. Hierbij filteren we de stream, waarbij de from Station van de Connection (x in dit geval) niet gelijk mag zijn aan de meegegeven Station.

Builder

Add line

Deze methode voegt een Line toe aan de lineList in de Builder. Deze Line is meestal een metro of bus lijn, waarbij een naam, type en natuurlijk de Stations worden meegegeven.

```
public Builder addLine(String[] lineDefinition) {
    String name = lineDefinition[0];
    String type = lineDefinition[1];
    Line line = new Line(name, type);
    for (int i = 2; i < lineDefinition.length; i++) {
        Station station = new Station(lineDefinition[i]);
        stationSet.add(station);
        line.addStation(station);
    }
    lineList.add(line);
    return this;
}
```

Snippet 8: Eerst wordt de naam en de type van de Line opgehaald van de meegegeven array van Strings. Vervolgens wordt de Line geïnstantieerd. Daarna loopen we door de rest van de array heen, deze bestaan uit Station namen. In de loop zeggen we dat er een Station moet worden geïnstantieerden deze worden toegevoegd aan de stationSet in de Builder in aan de Line die we eerder hadden geïnstantieerd. Als laatst zeggen we dat de Line toegevoegd moet worden aan de lineList.

Add weight

Deze methode voegt aan alle Connections in alle Lines een weight toe. Deze weight is de afstand tussen de Stations in de Connection.

```
public Builder addWeight(String[] lineDefinition, double[] weights) throws Exception {
    String lineName = lineDefinition[0];
    String lineType = lineDefinition[1];
    int index = 0;

    if (weights.length > (lineDefinition.length - 2)) {
        throw new Exception("Weights array is not compatible with lines array.");
    }
```

```
    Line line = getLine(lineName, lineType);

    if (line == null) {
        throw new Exception("Line does not exist!");
    }

    Station tempPrev = null;
    for (Station station : line.getStationsOnLine()) {
        if (tempPrev == null) {
            tempPrev = station;
            continue;
        }

        Station prevStation = tempPrev;
        Optional<Connection> connection = connectionSet.stream()
                .filter(x -> x.getFrom().equals(prevStation) &&
x.getTo().equals(station))
                .findFirst();
        Optional<Connection> reverseConnection = connectionSet.stream()
                .filter(x -> x.getFrom().equals(station) &&
x.getTo().equals(prevStation))
                .findFirst();

        if (connection.isPresent() && reverseConnection.isPresent()) {
            Connection conn = connection.get();
            Connection revConn = reverseConnection.get();
            conn.setWeight(weights[index]);
            revConn.setWeight(weights[index++]);
        }

        tempPrev = station;
    }

    return this;
}
```

Snippet 9: Aller eerst halen we de Line op in de TransportGraph. Daarbij worden er een aantal checks gedaan. Eerst wordt gekeken of de aantal Stations in de lineDefinition gelijk is aan de aantal weights in de weights array. Ook wordt gekeken of de Line, die we ophalen, bestaat in de TransportGraph. Vervolgens loopen we door alle Stations in de Line. We houden een prev Station bij. Deze Station is simpel gezegd de Station dat voor de huidige Station staat. Als deze prev Station null is, dan zeggen we dat de prev station gelijk is aan de huidige Station in de loop en zeggen we dat we door moeten naar de volgende Station. Vervolgens halen we de Connections op van de connectionSet en voegen we aan deze connections de weights toe.

Add location

Deze methode voegt aan alle Stations in alle Lines een Location toe. Deze Location wordt gebruikt om de afstand van Station naar Station te bepalen met behulp van de x en y coördinaten.

```
public Builder addLocation(String[] lineDefinition, int[][] coordinates) throws
Exception {
    String lineName = lineDefinition[0];
    String lineType = lineDefinition[1];

    if (coordinates.length > (lineDefinition.length - 2)) {
        throw new Exception("Weights array is not compatible with lines array.");
    }

    Line line = getLine(lineName, lineType);

    if (line == null) {
        throw new Exception("Line does not exist!");
    }

    for (int i = 0; i < line.getStationsOnLine().size(); i++) {
        int x = coordinates[i][0];
        int y = coordinates[i][1];

        Location location = new Location(x, y);
        Station station = line.getStationsOnLine().get(i);
        station.setLocation(location);
    }

    return this;
}
```

Snippet 10: Eerst halen we de Line op in de TransportGraph. Daarbij worden er een aantal checks gedaan. Eerst wordt gekeken of de aantal Stations in de lineDefinition gelijk is aan de aantal coordinates in de coordinates array. Ook wordt gekeken of de Line, die we ophalen, bestaat in de TransportGraph. Vervolgens loopen we door alle Stations heen in de huidige Line. In de loop instantieren we een Location op basis van de coordinates die we meegeven in de parameters. Deze Location geven we dan mee aan de huidige Station.

Build station set

Deze methode bouwt de stationSet op in de Builder class. Dit doet die aan de hand van de lineList.

```
public Builder buildStationSet() {
    lineList.forEach(line -> stationSet.addAll(line.getStationsOnLine()));
return this;}
```

Snippet 9: We loopen door alle Lines heen en geven aan dat de Stations in de Line toegevoegd moeten worden aan de stationSet.

Add lines to stations

Deze methode voegt een Line toe aan een Station.

```
public Builder addLinesToStations() {
    for (Station station : stationSet) {
        for (Line line : lineList) {
            if (line.getStationsOnLine().contains(station) && !station.hasLine(line)) {
                station.addLine(line);
            }
        }
    }
    return this;
}
```

Snippet 10: We loopen door alle Stations heen in de stationSet en hierin loopen we door alle Lines heen in de lineList. Vervolgens kijken we of de huidige Line wel de huidige Station bevat en dat de huidige Station niet de huidige Line heeft. Als ze aan beide condities voldoen dan wordt deze Line toegevoegd aan de Station.

Build connections

Deze methode bouwt alle mogelijke Connections op in een Line.

```
public Builder buildConnections() {
    for (Line line : lineList) {
        Station prev = null;
        for (Station station : line.getStationsOnLine()) {
            if (prev == null) {
                prev = station;
                continue;
            }
            Connection forthConnection = new Connection(prev, station, 0.0, line);
            connectionSet.add(forthConnection);
            Connection backConnection = new Connection(station, prev, 0.0, line);
            connectionSet.add(backConnection);
            prev = station;
        }
    }

    return this;}
```

Snippet 11: We loopen door alle Lines heen van de lineList en hierin loopen we door alle Stations heen dat in de huidige Line staat. Voordat we alle Stations heen loopen houden we een prev Station bij. Deze Station is simpel gezegd de Station dat voor de huidige Station staat. Als deze prev Station null is, dan zeggen we dat de prev station gelijk is aan de huidige Station in de loop en zeggen we dat we door moeten naar de volgende Station, dus we maken nog geen Connections aan. Nu gaan we hetzelfde proces af, maar dit keer is de prev Station de Station in de vorige "loop ronde". Daarna vertellen we dat de Connections zowel heen als terug moeten worden aangemaakt en worden toegevoegd aan de connectionSet.

Build

Deze methode bouwt de TransportGraph door alle Stations en Connections toe te voegen aan de TransportGraph. Deze Stations worden toegevoegd als vertexes en de Connection worden toegevoegd als edges.

```
public TransportGraph build() {
    TransportGraph graph = new TransportGraph(stationSet.size());
    for (Station s : this.stationSet) {
        graph.addVertex(s);
    }

    for (Connection c : this.connectionSet) {
        graph.addEdge(c);
    }

    return graph;
}
```

Snippet 12: Eerst wordt de TransportGraph geïnstantieerd met als size het aantal Stations in stationSet. Vervolgens loopen we door de stationSet en voegen alle Stations toe als vertex aan de graph. Hetzelfde doen we voor de Connections. Als laatst returnen we de graph zelf.

Hoofdstuk 2: Location

In dit hoofdstuk worden de implementaties in de klasse Location besproken. Deze klasse geeft simpelweg de locatie van een Station aan. Dit doet die aan de hand van een x en y coördinaten. Hierbij wordt ook de tijd berekend om van Location A naar B te rijzen. Een transitie van bijvoorbeeld [3,5] naar [3,6], dus 1 coordinaat verschuiven, heeft een vaste reistijd van 1.5 minuten.

Get travelled squares

Deze methode berekend het totaal aantal ‘vierkanten’ tussen de huidige Location en de Location van onze eindbestemming.

```
public int getTravelledSquares(Location to) {
    int xDiff = Math.abs(x - to.getX());
    int yDiff = Math.abs(y - to.getY());
    return xDiff + yDiff;
}
```

Snippet 13: De variabelen xDiff en yDiff berekenen de afstand van de x coördinaat en de afstand van de y coördinaat . Dit wordt gedaan aan de hand van Math.abs. Dit berekend de absolute waarde van het verschil. Dus stel we doen 3-5 dan zouden we 2 krijgen als resultaat. Het blijft positief.

Travel time

Deze methode berekend de reistijd tussen 2 Locations. Dit wordt gedaan aan de hand van de getTravelledSquares() methode.

```
public double travelTime(Location to) {
    return getTravelledSquares(to) * TRAVEL_TIME;
}
```

Snippet 14: De totale vierkanten dat wordt afgelegd tussen het huidige Location met de eind bestemming ‘to’. Vervolgens wordt dit vermenigvuldigd met de constante TRAVEL_TIME. Dat is dan 1.5.

Hoofdstuk 3: Graph algorithms

Abstract path

In dit hoofdstuk worden de implementaties in de klasse AbstractPathSearch besproken. Deze klasse is bedoeld om de verschillende zoekopdrachten uit te voeren afhankelijk van de type search. In totaal zijn er 4 verschillende klassen: BreadthFirstSearch, DepthFirstSearch, DijkstraShortestPath en A_Star. De methodes die in AbstractPathSearch zijn geïmplementeerd zijn hasPathTo() ,pathTo() en countTransfer().

Has path to

Deze methode checked of er een mogelijke route is naar een bepaalde Station.

```
public boolean hasPathTo(int vertex) {
    return marked[vertex];
}
```

Snippet 15: De methode checked of er op de index van de meegegeven Station index al bezocht is. Dit checked die aan de hand van de marked array.

Path to

Deze methode bouwt een route naar de endIndex. Deze endIndex wordt al van tevoren bepaald tijdens het initialiseren van een graphalgorithm class.

```
public void pathTo(int vertex) {
    if (!hasPathTo(vertex)) {
        return;
    }

    Line commonLine = null;
    Stack<Station> path = new Stack<>();
    for (int x = vertex; x != startIndex; x = edgeTo[x]) {
        Station current = graph.getStation(x);
        Station next = graph.getStation(edgeTo[x]);

        path.push(current);

        Connection connection = graph.getConnection(x, edgeTo[x]);
        if (connection != null) {
            totalWeight += connection.getWeight();
        }
```

```
        if (commonLine == null) {
            commonLine = current.getCommonLine(next);
            continue;
        }

        if (!commonLine.getStationsOnLine().contains(next)) {
            countTransfers();
            commonLine = current.getCommonLine(next);
        }
    }

    path.push(graph.getStation(startIndex));

    while (!path.isEmpty()) {
        Station current = path.pop();
        nodesInPath.add(current);
        verticesInPath.add(graph.getIndexOfStationByName(current.getStationName()));
    }
}
```

Snippet 16: MISSING

Count transfers

Deze methode telt 1 punt bij de aantal transfers.

```
public void countTransfers() {
    transfers++;
}
```

Snippet 17: De methode telt 1 punt bij transfers.

Breadth first path

De BreathFirstPath is een child class van AbstractPathSearch. BreadthFirstPath maakt gebruik van een queue voor het zoeken van een route.

Search

De search methode zoekt een route van de startIndex tot aan de endIndex.

```
@Override
public void search() {
    Queue<Integer> queue = new LinkedList<>();
    nodesVisited.add(graph.getStation(startIndex));
    marked[startIndex] = true;
    queue.add(startIndex);
```

```
    loop:
    while (!queue.isEmpty()) {
        int currentIndex = queue.remove();

        for (int w : graph.getAdjacentVertices(currentIndex)) {
            if (!marked[w]) {

                nodesVisited.add(graph.getStation(w));
                edgeTo[w] = currentIndex;
                marked[w] = true;
                if (w == endIndex) {
                    break loop;
                }
                queue.add(w);
            }
        }
    }
    pathTo(endIndex);
}
```

Snippet 18: Eerst instantiëren we de queue waarin we de bezochte Stations in opslaan. Vervolgens zeggen we we dat de startIndex al bezocht is, aangezien we daar oom beginnen en deze voegen we dan ook toe aan de queue. Vervolgens loopen we door de queue heen en halen we de eerst volgende Station eruit. Nu kijken we naar de buren van de huidige Station, dus de adjacant vertices. Vervolgens checken we of de adjacent vertex w al bezocht is. Zo niet dan voegen we deze toe aan de nodesVisited lijst en zeggen we dat deze bezocht is. Verder checken we of de huidige vertex gelijk is aan de endIndex. Als dat het geval is, dan moeten we stoppen met de gehele while loop en bouwen we de path met de methode pathTo().

Depth first path

De DepthFirstPath is een child class van AbstractPathSearch. DepthFirstPath maakt gebruik van een stack structuur. Hierbij maken we gebruik van recursion.

Search

Deze methode roept de recursiveSearch() methode aan en vervolgens ook de pathTo() methode.

```
@Override
public void search() {
nodesVisited.add(graph.getStation(startIndex));
    recursiveSearch(startIndex);
    pathTo(endIndex);
}
```

Snippet 19: Allereerst wordt de eerste Station van de TransportGraph toegevoegd aan nodesVisited. Vervolgens wordt de recursiveSearch() methode aangeroepen en daarna de pathTo() methode.

Recursive search

Deze recursive methode wordt aangeroepen in de search() methode hierboven. Hier wordt de gehele zoek algoritme uitgevoerd. Het lijkt veel op de search() methode in BreadthFirstPath, alleen hier wordt er een Stack structuur gebruikt.

```
private void recursiveSearch(int vertex) {

    marked[vertex] = true;
    for (int w : graph.getAdjacentVertices(vertex)){
        if (!marked[w]){
            nodesVisited.add(graph.getStation(w));
            edgeTo[w] = vertex;
            marked[w] = true;
            if (w == endIndex){
                break;
            }
            recursiveSearch(w);
        }
    }
}
```

Snippet 20: De recursiveSearch() methode heeft dezelfde implementatie als de search() in BreadthFirstPath. We loopen door alle buur Stations en checken of de huidige Station w bezocht is. Zo niet dan voegen we deze toe aan de nodesVisited lijst en zeggen we dat deze bezocht is. Daarnaast checken we ook of de huidige Station gelijk is aan de eind bestemming. Dit stopt de loop, aangezien we dan al klaar zijn. Als laatst roepen we recursiveSearch opnieuw aan en gaan we verder met de huidige Station w en kijken naar zijn buur Stations.

Dijkstra's shortest path

Relax

```
public void relax(TransportGraph graph, int currentVertex) {
    Station station = graph.getStation(currentVertex);
    nodesVisited.add(station);

    for (Connection connection : graph.getAdjacentConnections(station)) {
        int nextVertex =
graph.getIndexOfStationByName(connection.getTo().getStationName());
        edgeToType[nextVertex] = connection.getLine();
        double totalTravelCost = distTo[currentVertex] + connection.getWeight() +
getTransferPenalty(currentVertex, nextVertex);
```

```
        if (distTo[nextVertex] > totalTravelCost) {
            distTo[nextVertex] = totalTravelCost;
            edgeTo[nextVertex] = currentVertex;

            if (pq.contains(nextVertex)) {
                pq.decreaseKey(nextVertex, distTo[nextVertex]);
            } else {
                pq.insert(nextVertex, distTo[nextVertex]);
            }
        }
    }
}
```

Snippet 21: MISSING

Search

Deze methode voert de search uit met behulp van de PriorityQueue en de relax() methode.

```
@Override
public void search() {
    while (!pq.isEmpty()) {
        int index = pq.delMin();
        if (index == endIndex) {
            break;
        }
        relax(graph, index);
    }
    pathTo(endIndex);
}
```

Snippet 22: Er wordt door de PriorityQueue heen geloopt en pakt telkens de kleinste key in de queue. Vervolgens wordt gechecked of deze key gelijk is aan de endIndex. Daarna wordt de relax methode aangeroepen. Op het moment dat de queue leeg is of als de eind bestemming bereikt is stoppen we met loopen en bouwen we de route met de pathTo() methode.

Has path to

Deze methode checked of er een route mogelijk is naar de meegegeven Station.

```
@Override
public boolean hasPathTo(int vertex) {
    return distTo[vertex] < Double.POSITIVE_INFINITY;
}
```

Snippet 23: De methode checked of de afstand naar de meegegeven Station index kleiner is dan POSITIVE_INFINITY. Simpel gezegd is dit een positieve groot getal.

Get transfer penalty

```
protected int getTransferPenalty(int from, int to) {
    final int metroPenalty = 6;
    final int busPenalty = 3;

    Line currentLine = edgeToType[from];
    Line newLine = graph.getConnection(from, to).getLine();

    if (currentLine == null || newLine == null) {
        return 0;
    }

    String currentLineType = currentLine.getType();
    String newLineType = newLine.getType();

    switch (currentLineType) {
        case "metro":
            if (newLineType.equals("metro")) {
                return metroPenalty;
            }

            if (newLineType.equals("bus")) {
                return busPenalty;
            }

            break;

        case "bus":
            return busPenalty;
    }

    return 0;
}
```

Snippet 24: MISSING

A*

Relax

```
@Override
public void relax(TransportGraph graph, int currentVertex) {
    Station station = graph.getStation(currentVertex);
    nodesVisited.add(station);

    for (Connection connection : graph.getAdjacentConnections(station)) {
```

```
        int nextVertex =
graph.getIndexOfStationByName(connection.getTo().getStationName());
        edgeToType[nextVertex] = connection.getLine();
        double totalTravelCost = distTo[currentVertex] + connection.getWeight() +
getTransferPenalty(currentVertex, nextVertex) + euclideanFrom[nextVertex];

        if (distTo[nextVertex] > totalTravelCost) {
            distTo[nextVertex] = totalTravelCost;
            edgeTo[nextVertex] = currentVertex;

            if (pq.contains(nextVertex)) {
                pq.decreaseKey(nextVertex, distTo[nextVertex]);
            } else {
                pq.insert(nextVertex, distTo[nextVertex]);
            }
        }
    }
}
```

Snippet 25: MISSING

Output

De resultaat van de simulatie:

```
10
30
Graph with 10 vertices and 30 edges:
A: B-E-G
B: A-C-E-F
C: B-D-G-I
D: C-G-H
E: A-B-H
F: B-G
G: A-C-D-F-J
H: D-E-I
I: C-H
J: G
```

Het verwachte resultaat:

```
Graph with 10 vertices and 15 edges:
A: B-E-G
B: A-C-E-F
C: B-D-G-I
D: C-G-H
E: A-B-H
F: B-G
G: A-C-D-F-J
H: D-E-I
I: C-H
J: G
```

De visited nodes en het aantal vertices komen overeen met het verwachte resultaat, maar het aantal edges helaas niet. Dit heeft te maken met het feit dat we voor zowel heen als terug een Connection hebben aangemaakt.

De resultaat van de searches:

```
## DFS ##
Path from E to J: [E, A, B, C, D, G, J] with 3 transfers
Nodes in visited order: E A B C D G F J H I
```

```
## BFS ##
Path from E to J: [E, A, G, J] with 1 transfers
Nodes in visited order: E A B H G C F D I J
```

Het verwacht resultaat van de searches:

```
Result of DepthFirstSearch:
Path from E to J: [E, A, B, C, D, G, J] with 3 transfers
Nodes in visited order: E A B C D G F J H I

Result of BreadthFirstSearch:
Path from E to J: [E, A, G, J] with 1 transfers
Nodes in visited order: E A B H G C F D I J
```

Overview DFS
Path from A to B: [A, B] with 0 transfers
Nodes in visited order: A B

Path from A to C: [A, B, C] with 0 transfers
Nodes in visited order: A B C E H D G F J I

Path from A to D: [A, B, C, D] with 0 transfers
Nodes in visited order: A B C D E H I F G J

Path from B to A: [B, A] with 0 transfers
Nodes in visited order: B A

Path from B to C: [B, A, E, H, D, C] with 2 transfers
Nodes in visited order: B A E H D C I G F J

Path from B to D: [B, A, E, H, D] with 1 transfers
Nodes in visited order: B A E H D G C I F J

Path from C to A: [C, B, A] with 0 transfers
Nodes in visited order: C B A D G F J H E I

Path from C to B: [C, B] with 0 transfers
Nodes in visited order: C B

Path from C to D: [C, B, A, E, H, D] with 1 transfers
Nodes in visited order: C B A E H D G F J I

Path from D to A: [D, C, B, A] with 0 transfers
Nodes in visited order: D C B A G F J I H E

Overview BFS
Path from A to B: [A, B] with 0 transfers
Nodes in visited order: A B

Path from A to C: [A, B, C] with 0 transfers
Nodes in visited order: A B C E H D G F J I

Path from A to D: [A, B, C, D] with 0 transfers
Nodes in visited order: A B C D E H I F G J

Path from B to A: [B, A] with 0 transfers
Nodes in visited order: B A

Path from B to C: [B, A, E, H, D, C] with 2 transfers
Nodes in visited order: B A E H D C I G F J

Path from B to D: [B, A, E, H, D] with 1 transfers
Nodes in visited order: B A E H D G C I F J

Path from C to A: [C, B, A] with 0 transfers
Nodes in visited order: C B A D G F J H E I

Path from C to B: [C, B] with 0 transfers
Nodes in visited order: C B

Path from C to D: [C, B, A, E, H, D] with 1 transfers
Nodes in visited order: C B A E H D G F J I

Path from D to A: [D, C, B, A] with 0 transfers
Nodes in visited order: D C B A G F J I H E

Overview Dijkstra
Path from Haven (14, 1) to Marken (12, 3): [Haven (14, 1), Marken (12, 3)] with 0 transfers
Nodes in visited order: Haven

Path from Haven (14, 1) to Steigerplein (10, 5): [Haven (14, 1), Marken (12, 3), Steigerplein (10, 5)] with 0 transfers
Nodes in visited order: Haven Marken

Path from Haven (14, 1) to Centrum (8, 8): [Haven (14, 1), Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8)] with 0 transfers
Nodes in visited order: Haven Marken Steigerplein Ymeerdijk Trojelaan
---
Path from Haven (14, 1) to Meridiaan (6, 9): [Haven (14, 1), Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8), Meridiaan (6, 9)] with 0 transfers
Nodes in visited order: Haven Marken Steigerplein Ymeerdijk Trojelaan Centrum Swingstraat
---
Path from Haven (14, 1) to Dukdalf (3, 10): [Haven (14, 1), Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8), Meridiaan (6, 9), Dukdalf (3, 10)] with 0 transfers
Nodes in visited order: Haven Marken Steigerplein Ymeerdijk Trojelaan Centrum Swingstraat Meridiaan Coltrane Cirkel Nobelplein Bachgracht Robijnpark Grote sluis Grootzeil

---
Path from Haven (14, 1) to Oostvaarders (0, 11): [Haven (14, 1), Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8), Meridiaan (6, 9), Dukdalf (3, 10), Oostvaarders (0, 11)] with 0 transfers
Nodes in visited order: Haven Marken Steigerplein Ymeerdijk Trojelaan Centrum Swingstraat Meridiaan Coltrane Cirkel Nobelplein Bachgracht Robijnpark Grote sluis Grootzeil Dukdalf Violetplantsoen
---
Path from Marken (12, 3) to Haven (14, 1): [Marken (12, 3), Haven (14, 1)] with 0 transfers
Nodes in visited order: Marken

Path from Marken (12, 3) to Steigerplein (10, 5): [Marken (12, 3), Steigerplein (10, 5)] with 0 transfers
Nodes in visited order: Marken Haven

Path from Marken (12, 3) to Centrum (8, 8): [Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8)] with 0 transfers
Nodes in visited order: Marken Haven Steigerplein Trojelaan

Path from Marken (12, 3) to Meridiaan (6, 9): [Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8), Meridiaan (6, 9)] with 0 transfers
Nodes in visited order: Marken Haven Steigerplein Trojelaan Centrum Swingstraat Ymeerdijk
---

Overview A_star
Path from Haven (14, 1) to Marken (12, 3): [Haven (14, 1), Marken (12, 3)] with 0 transfers
Nodes in visited order: Haven

Path from Haven (14, 1) to Steigerplein (10, 5): [Haven (14, 1), Marken (12, 3), Steigerplein (10, 5)] with 0 transfers
Nodes in visited order: Haven Marken

Path from Haven (14, 1) to Centrum (8, 8): [Haven (14, 1), Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8)] with 0 transfers
Nodes in visited order: Haven Marken Steigerplein Ymeerdijk Trojelaan
---
Path from Haven (14, 1) to Meridiaan (6, 9): [Haven (14, 1), Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8), Meridiaan (6, 9)] with 0 transfers
Nodes in visited order: Haven Marken Steigerplein Ymeerdijk Trojelaan Centrum Swingstraat
---
Path from Haven (14, 1) to Dukdalf (3, 10): [Haven (14, 1), Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8), Meridiaan (6, 9), Dukdalf (3, 10)] with 0 transfers
Nodes in visited order: Haven Marken Steigerplein Ymeerdijk Trojelaan Centrum Swingstraat Meridiaan Coltrane Cirkel Nobelplein Bachgracht Robijnpark Grote sluis Grootzeil

---
Path from Haven (14, 1) to Oostvaarders (0, 11): [Haven (14, 1), Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8), Meridiaan (6, 9), Dukdalf (3, 10), Oostvaarders (0, 11)] with 0 transfers
Nodes in visited order: Haven Marken Steigerplein Ymeerdijk Trojelaan Centrum Swingstraat Meridiaan Coltrane Cirkel Nobelplein Bachgracht Robijnpark Grote sluis Grootzeil Dukdalf Violetplantsoen
---
Path from Marken (12, 3) to Haven (14, 1): [Marken (12, 3), Haven (14, 1)] with 0 transfers
Nodes in visited order: Marken

Path from Marken (12, 3) to Steigerplein (10, 5): [Marken (12, 3), Steigerplein (10, 5)] with 0 transfers
Nodes in visited order: Marken Haven

Path from Marken (12, 3) to Centrum (8, 8): [Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8)] with 0 transfers
Nodes in visited order: Marken Haven Steigerplein Trojelaan

Path from Marken (12, 3) to Meridiaan (6, 9): [Marken (12, 3), Steigerplein (10, 5), Centrum (8, 8), Meridiaan (6, 9)] with 0 transfers
Nodes in visited order: Marken Haven Steigerplein Trojelaan Centrum Swingstraat Ymeerdijk
---

Conclusie

Deze opdracht heeft ons geleerd dat je op verschillende manieren routes kan bepalen. We hebben gebruik gemaakt van de volgende zoek algoritmes: DepthFirstSearch, BreadthFirstSeach, DijkstrasShortestPath en A-star.

In het begin was het wel puzzelen hoe de paden gebouwd moesten worden in de methode pathTo(). Daarnaast kregen wij ook een aantal bugs in de Builder, omdat we een aantal methodes zijn vergeten aan te roepen dat de Stations en dergelijke toevoegt aan de lijst van Stations. Uiteindelijk zijn we er helemaal doorheen kunnen gaan.

BreadthFirstSearch en DepthFirstSearch waren vergeleken met Dijkstra en A-star makkelijker te implementeren, omdat het structuur erachter niet te complex en uitgebreid is. Hoewel, de route dat wordt afgelegd is niet efficiënt. Dijkstra daarentegen kiest een betere route, maar het proces neemt langer de tijd. Uiteindelijk kunnen we concluderen dat A-star het meest optimale.A-star heeft dezelfde implementatie als Dijkstra, maar er komt een extra stuk informatie bij, namelijk de Location. Dankzij de Location kunnen we de afstand bepalen tussen de huidige Station tot aan de eind Station en zo hoeven we minder nodes oftewel Stations bezoeken.