Algorithms and Data Structures

*Route Planner*

# 120 jaar bewegwijzering

# Introduction

In this assignment you will develop an application that finds (optimal) routes across the Dutch national road’s infrastructure from any given starting point to any given destination. You will implement / work with the following concepts:

* Directed graphs
* Searching algorithms (Depth-First Search, Breadth-First Search, Dijkstra’s Shortest Path)
* Sorting algorithms (Insertion Sort, Quick Sort)
* Builder pattern
* Singly LinkedList
* Iterators
* Java Streams API
* Comparator
* Recursion
* Efficiency of algorithms (Big O)

A starter project is provided which already implements the basic functionality to import traffic infrastructure data, report statistics about your solution and to plot maps for easy visual verification of your solutions. You find *// TODO* comments in several classes indicating where code is missing.

In the resources folder you find data sources “junctions.csv” and “roads.csv”. These files are loaded by the RoadMap class. RoadMap has a method to generate an .svg plot of its data. The .svg file will be written in the ‘target’ class-path folder. You can view .svg files with any compliant browser. Here you find the result of loading the small data set with a few roads in the Amsterdam area. No change or additional code is required in RoadMap.

Chart, radar chart

Description automatically generated

The imported data will be represented by objects of two classes:

* A **Junction** represents a town or crossing where multiple roads come together. A junction is uniquely identified by its **name**.
* A **Road** represents a pass way between Junction A and junction B. A road has a physical **length** (in km) and a **maxSpeed** (in km/h). Roads also have a **name**, for display purposes only. All roads in the source data are bi-directional with identical properties in both directions. But the RoadMap loader will replicate each road into two uni-directional instances (one from A to B and one from B to A) such that we can alter the properties of a single direction (in case of a traffic jam). In this assignment there can only be one road (two instances) between any two junctions A and B.

# Generic class DirectedGraph<V,E>

We expect you to providea clean object-oriented solution in which the algorithms are coded within a generic class DirectedGraph<V,E>. This class provides an abstraction of the roadmap of your route planner application. The V and E type parameters indicate **Vertices** and **Edges** in a directed graph representation. The V-type shall implement the Identifiable interface, which promises that vertices can be uniquely identified by a String identifier and a **getId()** method shall be implemented that retrieves the name of the vertex.

The DirectedGraph<V,E> class encapsulates the representation of the graph in two data structures:

* **Map<String,V> vertices:** maintains a map of all uniquely identified vertices in the graph.
* **Map<V,Map<V,E>> edges:** maintains a map of all directed edges in the graph, organised by source vertex and then by destination vertex. I.e.:

a) edges.get(v1) gives you a map of all outgoing edges from source vertex v1 organised by destination vertex

b) edges.get(v1).get(v2) gives you the edge from v1 to v2 (if any)

We could’ve chosen a representation of the graph with one datastructure. However, we chose separate data structures for vertices and edges to optimize different types of operations.

The **RoadMap** class extends **DirectedGraph<Junction,Road>** and realises the graph implementation of a junctions and roads traffic network.

The following are the most important methods to complete in the DirectedGraph<V,E> class:

* **V addOrGetVertex(newVertex)** adds a new vertex to the graph. If a vertex with the same id has been stored already, it returns the existing vertex and refuses to add another duplicate.
* **boolean addEdge(fromVertex, toVertex, newEdge)** adds newEdge to the map of edges in between fromVertex and toVertex in the graph and returns true if successful. If fromVertex or toVertex are not part of the graph yet, these shall first be added as well so that the newEdge also can be added still. But, if the graph already holds an edge between fromVertex and toVertex, nothing shall be added or replaced and false shall be returned.
* **boolean addEdge(fromId, toId, newEdge)** adds newEdge to the map of edges in between the vertex identified by fromId and the vertex identified by toId. If fromId or toId do not reference an existing vertex in the graph, nothing shall be added. Otherwise, the same conditions as above do apply (so this method should reuse addEdge(fromVertex, toVertex, newEdge)).

**Algorithms to implement**

* **DGPath depthFirstSearch(graph, startVertex, targetVertex)** tries to find a path in the graph between the given start vertex and target vertex using a recursive depth-first search algorithm. If no such path can be found, it shall return null.
* **DGPath breadthFirstSearch(graph, startVertex, targetVertex)** tries to find a path in the graph between the given start vertex and target vertex using a breadth-first search algorithm. If no such path can be found, it shall return null.
* **DGPath dijkstraShortestPath(graph, startVertex, targetVertex, weightMapper)** tries to find a path in the graph between the given start vertex and target vertex using the Dijkstra Shortest Path algorithm. If no such path can be found, it shall return null. For that it uses the provided edgeWeightMapper function to obtain the weight contribution of each individual edge. In your main application you will explore two mappings: one for total distance (Junction::getDistance) and one for total travel time.
* **List insertionSort(items, comparator)** tries to sort the given list of items in place using the insertion sort algorithm. The provided comparator is used to determine the order of elements. Items are sorted in place without the use of an auxiliary list or array. It returns the sorted list.
* **List quickSort(items, comparator)** tries to sort the given list of items in place using the quicksort algorithm. The provided comparator is used to determine the order of elements. Items are sorted in place without the use of an auxiliary list or array. It returns the sorted list.

### **SinglyLinkedList in the Context of DGPath<V>**

The class **Searcher.DGPath<V>** provides all relevant information about the outcome of a search. It provides a sequence of vertices from the start vertex towards the target vertex and the total weight of all edges in between. In addition, it tracks all vertices that have been visited during the search, such that we easily visualise and compare the search characteristics of different algorithms.

In the unit tests of the project, we extend the generic class for a configuration of a map of neighbouring Countries. It uses the Integer class for the edges, because in that example we only represent border length at the edge by a single integer. That independent example could guide you how to develop a tidy implementation.

The **SinglyLinkedList<E>** class is a lightweight, custom implementation of a singly linked list that supports adding elements at both the beginning (addFirst) and the end (add), retrieving elements by index (get), checking the current size (size), and iterating over elements (iterator). This linked list is particularly useful in the context of the Searcher.DGPath class, which represents a path in a directed graph (DGPath<V>). The path contains a sequence of connected vertices stored in a SinglyLinkedList<V>

# 

Figure 1: UML diagram of the appliaction

# 

# Requirements.

You are required to complete the implementation without breaking the abstraction or encapsulation of DirectedGraph<V,E>. You are not allowed to change a signature of any public method. You may change signatures of private methods and add more public and private methods and attributes as you find appropriate. You also may change the private local classes as you find appropriate (or not use these helper classes at all but follow your own approach of implementing the public methods. The use of these helper classes guides you towards a memory efficient implementation though.)

Below is a summary of your tasks, presented in the suggested order of completion:

1. Complete the definition of the Junction class so that all code compiles. You will need to add and implement additional methods, including the Builder class in Junction.Builder().
2. Complete the implementations of the DirectedGraph data management methods such that a roadmap can be built from reading the provided input files and a basic picture of the map can be exported.
3. Complete the SinglyLinkedList class so it can be used in DGPath for search algorithms.
4. Implement Searcher.depthFirstSearch() to calculate a viable path, stop when a path is found, and populate visitedVertices in the path.
5. Implement Searcher.breadthFirstSearch() to calculate a viable path, stop when a path is found, and populate visitedVertices in the path.
6. Implement Searcher.dijkstraShortestPath() to calculate the shortest path (by total length), stop when the path is found, and populate visitedVertices. You may use the helper class Searcher.DSPNode<V>
7. Complete the RoadNetworkAnalysis class without using for-loops. Leverage Sets, Maps, and Streams, utilizing methods like .mapToDouble(), .filter(), .max(), Collectors.groupingBy(), etc.
8. Complete the implementation of Sorter.insertionSort() with an implementation of insertion sort.
9. Complete the Sorter.quickSort() method with a **recursive implementation** of the quick sort.
10. Create benchmarks to compare insertion sort and quick sort (see [Efficiency](#_Efficiency) section below).
11. Verify the outcome of the main program against the console output at the end of this document.

Deliver tidy code:  
a) with proper encapsulation, code reuse and low cyclomatic complexity.  
b) with acceptable CPU-time complexity and memory footprint (without easy avoidable waste).  
c) with appropriate naming conventions and in-line comments.  
d) with additional unit tests as appropriate.

# 

# Efficiency

The **SorterPerformanceTest** class functions as a performance benchmarking utility rather than a conventional unit test. Its primary purpose is to measure and compare the execution times of sorting algorithms across different dataset sizes, not to validate functionality through assertions. Write your benchmarks in SorterPerformanceTest.measureEfficiency().

Determine how long it takes for **each sorting algorithm** to sort Countries (model in test) of sizes 100, 200, 400, 800, 1600, …etc., until you reach 5.000.000 countries or until sorting the list takes more than 20 seconds. A method to generate Countries is already provided (SorterPerformanceTest. GenerateCountryDataset()).

When measuring the time:

* + Compare Insertion sorter vs. quicksort
  + Verify sorting correctness by comparison with Collections.sort()
  + Use problem sizes up to 5,000,000 or 20 seconds max.
  + Ensure fair comparison: generate input lost once, copy for each algorithm
  + Call [System.gc](http://system.gc)() before each run.
  + Run JVM with **-Xint**.
  + Add results in your report. Present results in a table and graph. Explain the results (perform Big-O analysis:
    1. Provide a static analysis and big-O assessment of CPU-time complexity based on the code structure of each implementation.
    2. Provide a numerical analysis and big-O assessment of CPU-time complexity based on the computation time measurements of each implementation.

# Unit Tests

Some unit tests are provided to assist you to verify correctness of your code. These should all pass with green ticks. Note: not all edge cases are covered with tests.

**Report**

**Provide a report including**

1. Explanation and justification of seven of your most relevant code snippets in your solution. (a Dutch or English transcription of the statements in your code does not add any information.)

E.g: explain the content of a stream after every intermediate step in plain language (laymen’s terms).

1. An explanation of which search algorithm has calculated the route in each of the three figures (See appendix figures 1 - 3 below). Copy and paste the images in your report:
   1. Analyze each figure carefully.
   2. Determine which searching algorithm generated each figure (e.g., BFS, DFS, Dijkstra).
   3. **Explain your reasoning** in your report.
2. Tables with numerical data of measured execution times of all your sorting methods for problem sizes of 100, 200, 400, … etc, until the execution time exceeds 20 sec.
3. Graphical representation of execution time vs problem size of above performance results.
4. Big-O complexity analysis of each implemented sorting method based on the structure of the code.
5. Big-O complexity analysis of the measured execution times, by comparison ratios of measured execution times against ratios of problem sizes.
6. Full console output of the main program, and justification of the differences in your output with the reference console output that is provided below.
7. References to external sources that you may have consulted, if any.

# Grading

Grading criteria are according to guidance in the study manual and as per rubric at the DLO.

# Appendices

Below you find three pictures of example solutions searching for a route from “Amsterdam” to “Staphorst”, each using a different algorithm: depthFirstSearch, breadthFirstSearch and dijkstraShortestPath. Answer in your report which figure is the result of which algorithm and why (by qualitative reasoning).

* The found routes are depicted in the green colour.
* Green vertices and junction names have been visited by the search algorithm.
* Black vertices and junction names have not been visited by the search algorithm.
* Orange lines are highways with a speed limit of 100 or 120km/h.
* Yellow lines are district roads with a speed limit of 80km/h

Such pictures are generated in .svg format by the main program into a corresponding file in the ‘target’ class-path and can be viewed with a browser. You can review them from your own runs.



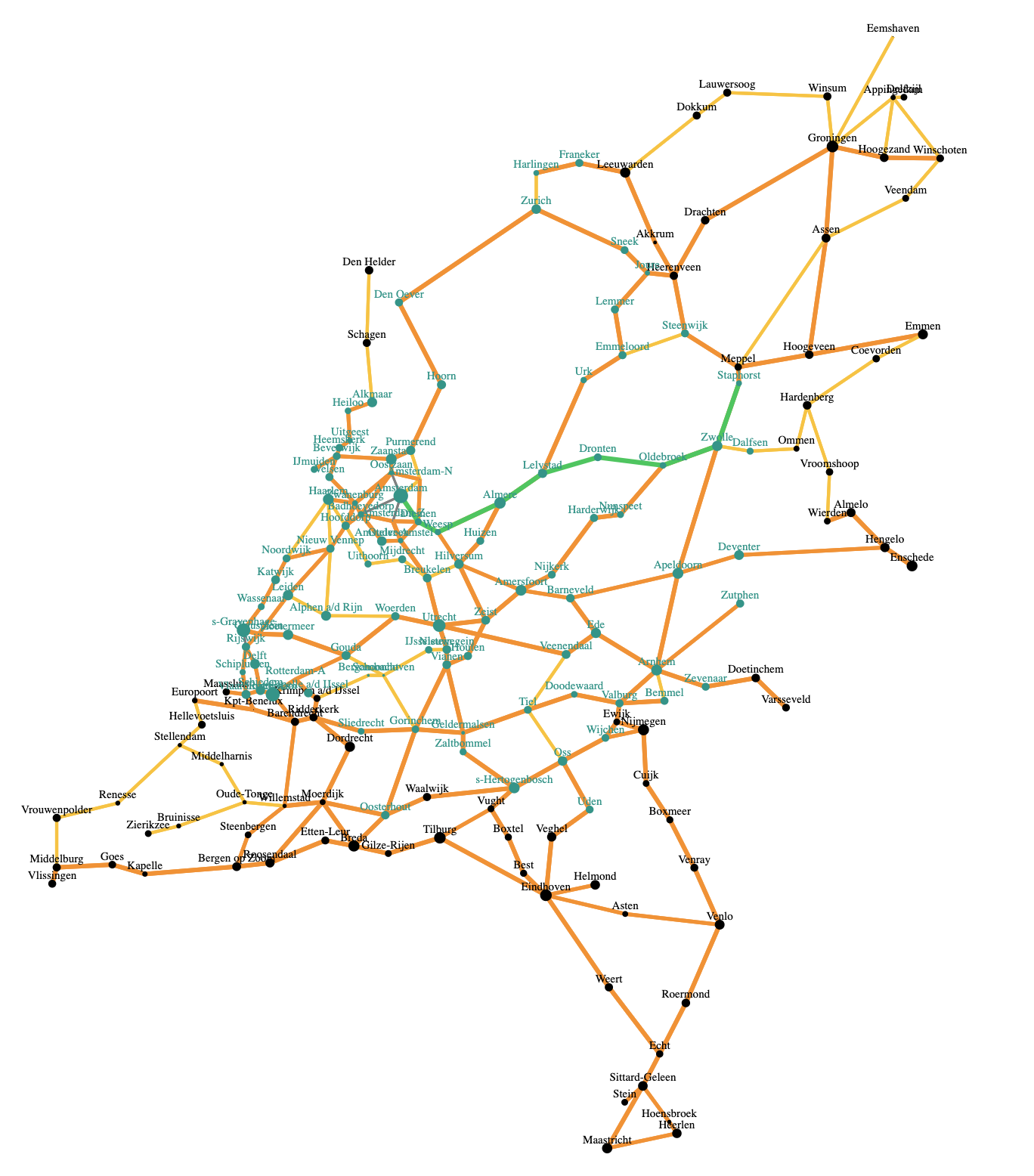


Figure 2: pathfinding result (1/3)

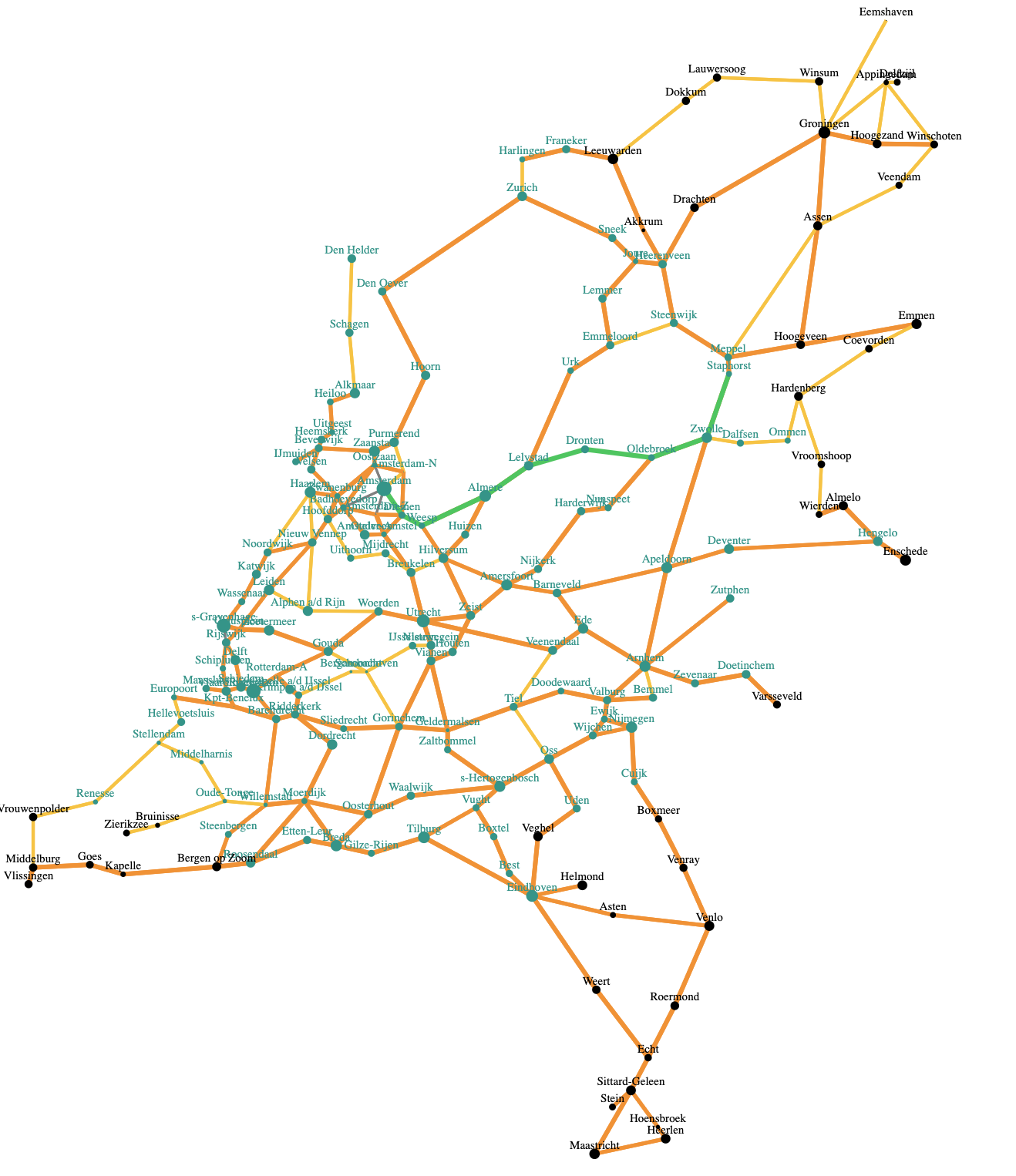


Figure 3: pathfinding result (2/3)

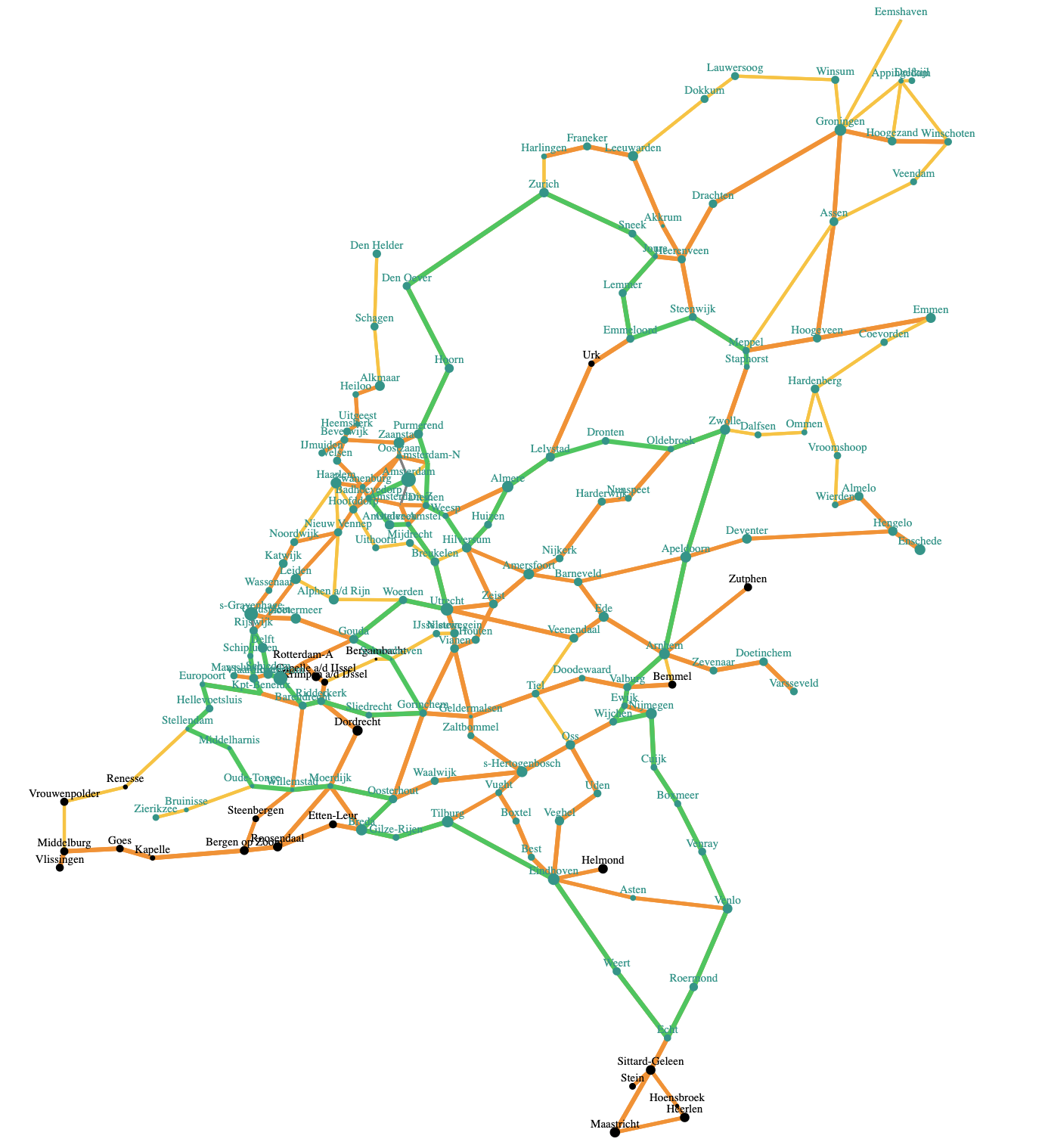


Figure 4: Pathfinding result (3/3)

Below you find console output of the sample routes explored by the main program in the starter project.  
Your implementation may differ in some of the numbers where the result is sensitive to the order in which neighbour vertices are explored. Explain/justify such differences in your report.

🚗 Welcome to the HvA RoutePlanner Adventure!

📍 Loading small demo map...

Importing junctions and roads from Junctions0.csv and Roads0.csv...

12 junctions and 18 bi-directional roads have been imported.

12 junctions and 36 one-way roads have been stored into the graph.

Roadmap lay-out:

{ Ouder-Amstel: [Amsterdam-Z(A2/120),Utrecht(A2/120),Badhoevedorp(A9/120),Diemen(A9/120)],

Badhoevedorp: [Ouder-Amstel(A9/120),Amsterdam-Z(A10/100),Oostzaan(A10/100),Velsen(A9/120),Leiden(A4/100),Amsterdam(S112/50)],

Utrecht: [Ouder-Amstel(A2/120)],

Amsterdam: [Amsterdam-Z(S108/50),Badhoevedorp(S112/50),Amsterdam-N(S102/80),Oostzaan(S108/50),Diemen(S102/80)],

Diemen: [Weesp(A1/100),Ouder-Amstel(A9/120),Amsterdam-Z(A10/100),Amsterdam-N(A10/100),Amsterdam(S102/80)],

Amersfoort: [Weesp(A1/100)],

Amsterdam-Z: [Ouder-Amstel(A2/120),Badhoevedorp(A10/100),Diemen(A10/100),Amsterdam(S108/50)],

Oostzaan: [Badhoevedorp(A10/100),Amsterdam-N(A10/100),Amsterdam(S108/50)],

Leiden: [Badhoevedorp(A4/100)],

Velsen: [Badhoevedorp(A9/120)],

Weesp: [Amersfoort(A1/100),Diemen(A1/100)],

Amsterdam-N: [Oostzaan(A10/100),Diemen(A10/100),Amsterdam(S102/80)]

}

🗺️ Exploring Oostzaan → Ouder-Amstel

Results from path searches from Oostzaan to Ouder-Amstel:

Depth-first-search: Weight=0,000000 Length=3 visited=3 (Oostzaan, Badhoevedorp, Ouder-Amstel)

Breadth-first-search: Weight=0,000000 Length=3 visited=5 (Oostzaan, Badhoevedorp, Ouder-Amstel)

Dijkstra-Shortest-Path (distance): Weight=21,232941 Length=4 visited=10 (Oostzaan, Amsterdam, Amsterdam-Z, Ouder-Amstel)

Dijkstra-Fastest-Route (time): Weight=0,264123 Length=3 visited=10 (Oostzaan, Badhoevedorp, Ouder-Amstel)

🌍 Loading full map of the Netherlands...

Importing junctions and roads from Junctions.csv and Roads.csv...

383 junctions and 245 bi-directional roads have been imported.

176 junctions and 490 one-way roads have been stored into the graph.

🚦 Planning road trip: Amsterdam → Meppel

Results from path searches from Amsterdam to Meppel:

Depth-first-search: Weight=0,000000 Length=42 visited=163 (Amsterdam, Amsterdam-Z, Ouder-Amstel, Breukelen, Utrecht, Veenendaal, Ede, Barneveld, Amersfoort, Zeist, Hilversum, Weesp, Almere, Lelystad, Dronten, Oldebroek, Zwolle, Apeldoorn, Deventer, Hengelo, Almelo, Wierden, Vroomshoop, Hardenberg, Coevorden, Emmen, Hoogeveen, Assen, Groningen, Drachten, Heerenveen, Akkrum, Leeuwarden, Franeker, Harlingen, Zurich, Sneek, Joure, Lemmer, Emmeloord, Steenwijk, Meppel)

Breadth-first-search: Weight=0,000000 Length=9 visited=100 (Amsterdam, Diemen, Weesp, Almere, Lelystad, Urk, Emmeloord, Steenwijk, Meppel)

Dijkstra-Shortest-Path (distance): Weight=127,105064 Length=10 visited=138 (Amsterdam, Diemen, Weesp, Almere, Lelystad, Dronten, Oldebroek, Zwolle, Staphorst, Meppel)

Dijkstra-Fastest-Route (time): Weight=1,201533 Length=9 visited=133 (Amsterdam, Diemen, Weesp, Almere, Lelystad, Urk, Emmeloord, Steenwijk, Meppel)

⚠️ Accident detected between Diemen and Weesp...

➡️ Fastest alternative route avoiding accident: Weight=1,445145 Length=13 visited=146 (Amsterdam, Diemen, Ouder-Amstel, Breukelen, Hilversum, Amersfoort, Nijkerk, Harderwijk, Nunspeet, Oldebroek, Zwolle, Staphorst, Meppel)

📊 Analyzing road network...

🌆 Most populated city: Amsterdam

🏙️ Top 5 biggest cities: [Amsterdam, Rotterdam, s-Gravenhage, Utrecht, Groningen]

🛣️ Total road length: 2697.225370124258

⚡ Amount of roads faster than average: 90

📌 Provinces with >10 cities: [NB, OV, NH, FR, GLD, ZH, UT]

📍 Road length per province: {ZLD=112.49477199950353, NB=385.8141163668238, FL=103.95017609956199, LB=154.66094404511017, OV=211.5890907695222, NH=340.14617809656625, GR=159.38942147109748, FR=250.50565852173048, DR=116.34292617557472, GLD=361.09361065929494, ZH=363.9009682875785, UT=137.33750763189403}

--- Sorting Demo ---

Quick Sort (by population) (3 runs):

Sample first 5: Clausplein Kpt-Benelux Eemshaven Rotterdam-A Amsterdam-N

Avg: 0,501 ms

Quick Sort (by name) (3 runs):

Sample first 5: Akkrum Alkmaar Almelo Almere Alphen a/d Rijn

Avg: 0,698 ms

🎉 Trip complete! Thanks for using HvA RoutePlanner 🚀

Process finished with exit code 0

