Algorithms and Data Structures

Graph Searches in a Route Planner

Assignment-5

Version: 20.2, 30 November 2020

Introduction

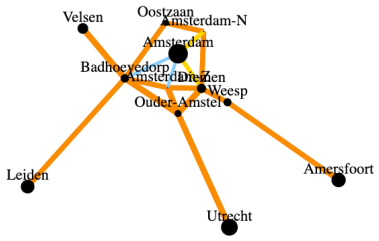
In this assignment you will develop an application that finds (optimal) routes across the Dutch national road's infrastructure from any given starting point to any given destination. You will implement four algorithms and compare results:

1. Depth-First-Search
2. Breadth-First-Search
3. Dijkstra-Shortest-Path
4. A-star-Shortest-Path

You will use both the shortest distance and the shortest travel time optimisation criteria and your algorithm will be able to find the best alternate route in case of traffic delays or obstructions.

Input data.

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



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.

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 junction has cartesian coordinates **locationX** and **locationY** in km, which indicate the position on the map. These coordinates follow the dutch RD-coordinate system, which is explained at https://nl.wikipedia.org/wiki/Rijksdriehoeksco%C3%B6rdinaten.
  Consequently, you can use Pythagoras's rule to calculate straight distances between junctions.
- A **Road** represents a passway **from** Junction A **to** junction B. Roads are uni-directional.
  A road has a physical **length** (in km) and a **maxSpeed** (in km/h). As the data source does not provide the physical length of the roads, we calculate in the road-constructor some semi-random length for the purpose of this assignment.
  Roads also have a **name**. The combination of name, from-junction and to-junction shall be unique. So, you can have two roads with different names (and length and maxSpeed) between the same junctions.

Generic class DirectedGraph<V,E>

We expect you to provide a clean object-oriented solution in which the search 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. Two interfaces DGVertex and DGEdge specify the implementation requirements of the relations between vertices and edges which will be navigated by the search algorithms:

- DGVertex provides access to its unique Id and its set of out-going edges.
- DGEdge provides access to its 'from'-vertex and 'to'-vertex that are connected by the edge.

The RoadMap class extends from **DirectedGraph<Junction,Road>** and will compile fine once you have implemented the interface requirements in your Junction and Road classes. Then also the generic search algorithms of DirectedGraph should be able to calculate routes along Junctions and Roads.
In Figure 5 you find a class diagram of how this all hangs together.

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.
- **E addOrGetEdge(newEdge)** adds newEdge to the set of edges of its 'from'-vertex. If the from- or to-vertex is not part of the graph yet, it will be added first. A run-time exception is thrown if the from- or to-vertex in newEdge is a duplicate of another vertex with the same id that had been stored into the graph before. Also, if the same edge already exists at the from-vertex, the existing edge is returned and no duplicate is added.
- **DGPath depthFirstSearch(startId, targetId)** and **DGPath breadthFirstSearch(startId, targetId)** search a path in the graph starting from the vertex with id=startId and ending at the vertex with id=targetId, leveraging depth-first and breadth first search strategies respectively.
- **DGPath dijkstraShortestPath(startId, targetId, edgeWeightMapper)** searches a similar path in the graph with minimal total weight using Dijkstra's algorithm. 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 and one for total travel time.
- **DGPath aStarShortestPath(startId, targetId, edgeWeightMapper, minimumWeightEstimator)** searches a similar path in the graph with minimal total weight using the A* algorithm. It uses the minimumWeightEstimator heuristic to optimize performance. Again, you will explore two estimators: one for total distance and one for total travel time.

The class **DirectedGraph.DGPath** provides all relevant information about the outcome of a search. It tracks the start vertex, the sequence of edges that specifies the path towards the target vertex and the total weight of all edges. 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 Countries and their Borders. That independent example could guide you how to develop a tidy implementation.

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:

R1. Complete the definitions of the Junction and Road classes up to the point that all code can be compiled.
R2. 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.
R3. Complete the implementation of DirectedGraph.depthFirstSearch such that it calculates a viable path, stops when a path is found and also populates the visitedVertices in the path. Verify your results from the console statistics and in the .svg picture.
R4. Complete the implementation of DirectedGraph.breadthFirstSearch such that it calculates a viable path, stops when a path is found and also populates the visitedVertices in the path. Verify your results from the console statistics and in the .svg picture.
R5. Complete the implementation of DirectedGraph. dijkstraShortestPath such that it calculates a viable path, stops when a path is found and also populates the visitedVertices in the path. Verify your results from the console statistics and in the .svg picture.
R6. Complete the implementation of DirectedGraph. aStarShortestPath such that it calculates a viable path, stops when a path is found and also populates the visitedVertices in the path. Provide an appropriate minimumWeightEstimator from the call in RoutePlannerMain.doPathSearches. Verify your results from the console statistics and in the .svg picture.
R7. Provide appropriate weightCalculator and minimumWeightEstimator function parameters for the RoadMap.aStarShortestPath call in RoutePlannerMain.doPathSearches that calculates the fastest route between given startId and targetId. Verify your results from the console statistics and in the .svg picture.
R8. Complete the implementation of DirectedGraph. dijkstraShortestPathByAStar such that it uses the DirectedGraph.aStarShortestPath method to effectively execute the plain Dijkstra's algorithm without involving the A* optimisation.
R9. Due to an accident under the new Vecht aqueduct at Weesp, traffic from Diemen to Weesp can only proceed at 5km/hour. Retrieve the edge from Diemen to Weesp from the RoadMap and adjust its maximum speed. Then use DirectedGraph. dijkstraShortestPathByAStar to calculate the fastest alternate route. Verify your results from the console statistics and in the .svg picture.
R10. 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).
b) with appropriate naming conventions and in-line comments.
c) with additional unit tests as appropriate.
R11. Provide a report including
a) an explanation of which of the four search algorithms has calculated the route in each of the four figures 1 – 4 below, referring to specific qualitative characteristics of each of the algorithms. (Justifications by visually matching actual calculation results are not valid)
b) Full console output of the main program.
c) Explanation and justification of seven of your most relevant code snippets in your solution.

Grading

For a sufficient grade, your solution shall comply with the guidance in this document and the source code of the start project and reproduce correct results on R1 – R5 and R9 and deliver R10 and R11. (Adding R6, R7 and R8 give you a good or excellent grade.) In Figure 6 you find samples of target results for reference.

Appendices

Here you find four pictures of example solutions searching for a route from “Amsterdam” to “Zwolle”, each using a different algorithm from ‘depthFirstSearch’, ‘breadthFirstSearch’, ‘dijkstraShortestPath’, ‘aStarShortestPath’. (Your implementation may result in different routes).

- The found routes is depicted in the lime-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.

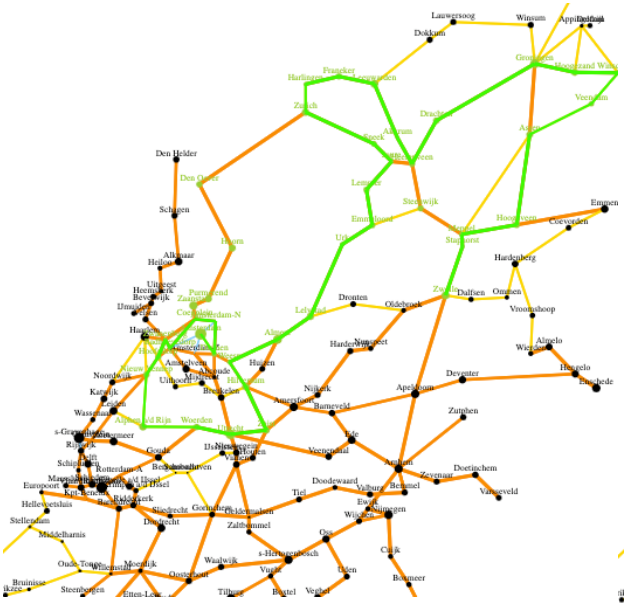


Figure 3: path from Amsterdam to Zwolle

Figure 1: path from Amsterdam to Zwolle

Figure 2: path from Amsterdam to Zwolle

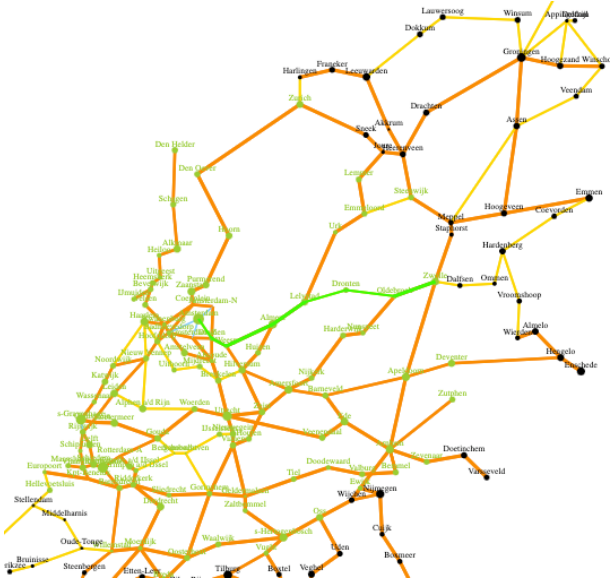


Figure 4: path from Amsterdam to Zwolle

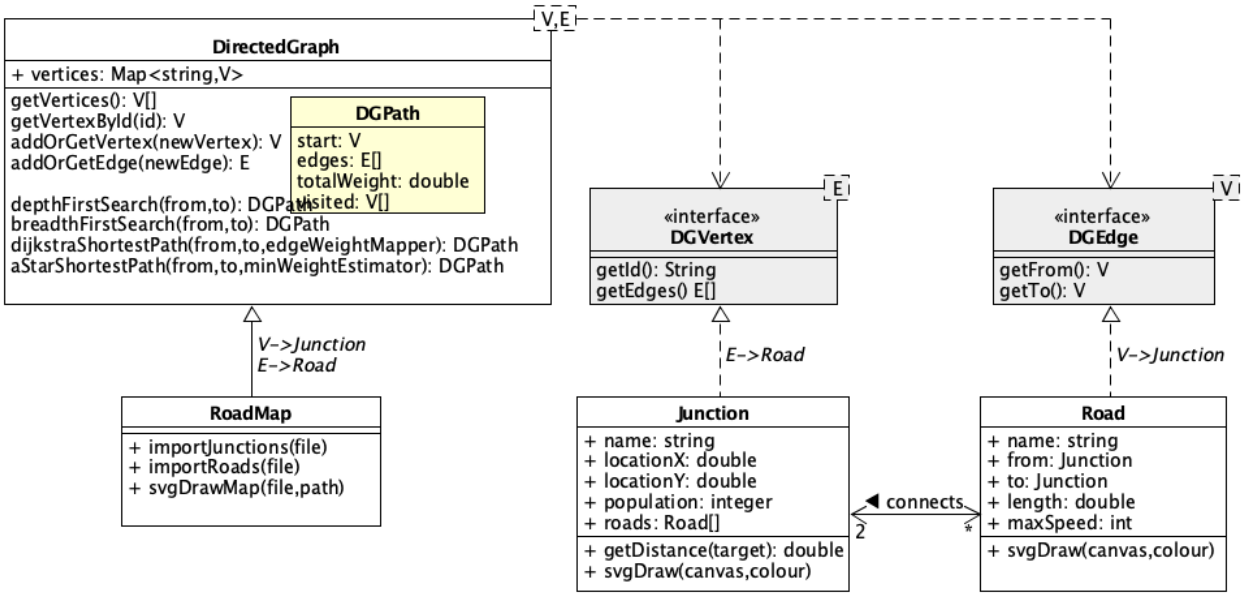


Figure 5: UML class diagram of starter project

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.

```
Welcome to the HvA RoutePlanner

Importing junctions and roads from Junctions0.csv and Roads0.csv...
12 junctions and 20 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(S108) Utrecht(A2) Badhoevedorp(A9) Amsterdam-Z(A2) Diemen(A9)],
  Badhoevedorp[Amsterdam(S112) Velsen(A9) Ouder-Amstel(A9) Amsterdam-Z(A10) Oostzaan(A10) Leiden(A4)],
  Utrecht[Ouder-Amstel(A2)],
  Amsterdam[Diemen(S102) Amsterdam-Z(S108) Amsterdam-N(S102) Badhoevedorp(S112)],
  Diemen[Weesp(A1) Amsterdam-Z(A10) Amsterdam(S102) Ouder-Amstel(A9) Amsterdam-N(A10)],
  Amersfoort[Weesp(A1)],
  Amsterdam-Z[Diemen(A10) Ouder-Amstel(S108) Ouder-Amstel(A2) Badhoevedorp(A10) Amsterdam(S108)],
  Oostzaan[Amsterdam-N(A10) Badhoevedorp(A10)],
  Leiden[Badhoevedorp(A4)],
  Velsen[Badhoevedorp(A9)],
  Weesp[Amersfoort(A1) Diemen(A1)],
  Amsterdam-N[Oostzaan(A10) Amsterdam(S102) Diemen(A10)]
}

Results from path searches from Oostzaan to Ouder-Amstel:
Depth-first-search: Weight=0,000000 Length=6 Visited=8 (Oostzaan, Amsterdam-N, Amsterdam, Diemen, Amsterdam-Z, Ouder-Amstel)
Depth-first-search return: Weight=0,000000 Length=6 Visited=8 (Ouder-Amstel, Amsterdam-Z, Diemen, Amsterdam, Amsterdam-N, Oostzaan)
Breadth-first-search: Weight=0,000000 Length=3 Visited=7 (Oostzaan, Badhoevedorp, Ouder-Amstel)
Breadth-first-search return: Weight=0,000000 Length=3 Visited=8 (Ouder-Amstel, Badhoevedorp, Oostzaan)
Dijkstra-Shortest-Path: Weight=29,523711 Length=4 Visited=11 (Oostzaan, Amsterdam-N, Diemen, Ouder-Amstel)
Dijkstra-Shortest-Path return: Weight=29,523711 Length=4 Visited=12 (Ouder-Amstel, Diemen, Amsterdam-N, Oostzaan)
AStar-Shortest-Path: Weight=29,523711 Length=4 Visited=10 (Oostzaan, Amsterdam-N, Diemen, Ouder-Amstel)
AStar-Shortest-Path return: Weight=29,523711 Length=4 Visited=11 (Ouder-Amstel, Diemen, Amsterdam-N, Oostzaan)
AStar-Fastest-Route: Weight=0,280603 Length=3 Visited=10 (Oostzaan, Badhoevedorp, Ouder-Amstel)
```

Figure 6: Console output small roadmap of Amsterdam area

```
Importing junctions and roads from Junctions.csv and Roads.csv...
383 junctions and 244 bi-directional roads have been imported.
176 junctions and 488 one-way roads have been stored into the graph.

Results from path searches from Amsterdam to Staphorst:
Depth-first-search: Weight=0,000000 Length=12 Visited=12 (Amsterdam, Diemen, Weesp, Hilversum, Huizen, Almere, Lelystad, Urk,
 Emmeloord, Steenwijk, Meppel, Staphorst)
Depth-first-search return: Weight=0,000000 Length=80 Visited=98 (Staphorst, Zwolle, Dalfsen, Ommen, Hardenberg, Vroomshoop,
 Wierden, Almelo, Hengelo, Deventer, Apeldoorn, Arnhem, Bemmel, Valburg, Doodewaard, Tiel, Geldermalsen, Zaltbommel,
 s-Hertogenbosch, Waalwijk, Oosterhout, Moerdijk, Willemstad, Steenbergen, Bergen op Zoom, Kapelle, Goes, Middelburg,
 Vrouwenpolder, Renesse, Stellendam, Hellevoetsluis, Europoort, Kpt-Benelux, Barendrecht, Rotterdam, Schiedam, Vlaardingen,
 Schipluiden, Rijswijk, Clausplein, s-Gravenhage, Wassenaar, Katwijk, Noordwijk, Nieuw Vennep, Haarlem, Zwanenburg, Hoofddorp,
 Uithoorn, Mijdrecht, Breukelen, Hilversum, Huizen, Almere, Lelystad, Urk, Emmeloord, Steenwijk, Meppel, Hoogeveen, Assen,
 Veendam, Winschoten, Hoogezand, Groningen, Drachten, Heerenveen, Akkrum, Leeuwarden, Franeker, Harlingen, Zurich, Den Oever,
 Hoorn, Purmerend, Zaanstad, Oostzaan, Amsterdam-N, Amsterdam)
Breadth-first-search: Weight=0,000000 Length=9 Visited=77 (Amsterdam, Diemen, Weesp, Hilversum, Amersfoort, Barneveld, Apeldoorn,
 Zwolle, Staphorst)
Breadth-first-search return: Weight=0,000000 Length=9 Visited=94 (Staphorst, Zwolle, Apeldoorn, Barneveld, Amersfoort, Hilversum,
 Weesp, Diemen, Amsterdam)
Dijkstra-Shortest-Path: Weight=130,355230 Length=9 Visited=133 (Amsterdam, Diemen, Weesp, Almere, Lelystad, Dronten, Oldebroek,
 Zwolle, Staphorst)
Dijkstra-Shortest-Path return: Weight=130,355230 Length=9 Visited=86 (Staphorst, Zwolle, Oldebroek, Dronten, Lelystad, Almere,
 Weesp, Diemen, Amsterdam)
AStar-Shortest-Path: Weight=130,355230 Length=9 Visited=43 (Amsterdam, Diemen, Weesp, Almere, Lelystad, Dronten, Oldebroek,
 Zwolle, Staphorst)
AStar-Shortest-Path return: Weight=130,355230 Length=9 Visited=29 (Staphorst, Zwolle, Oldebroek, Dronten, Lelystad, Almere, Weesp,
  Diemen, Amsterdam)
AStar-Fastest-Route: Weight=1,269243 Length=10 Visited=53 (Amsterdam, Diemen, Weesp, Almere, Lelystad, Urk, Emmeloord, Steenwijk,
 Meppel, Staphorst)

DijkstraByAStar-accident-Weesp: Weight=1,446783 Length=12 Visited=143 (Amsterdam, Diemen, Ouder-Amstel, Breukelen, Hilversum,
 Amersfoort, Nijkerk, Harderwijk, Nunspeet, Oldebroek, Zwolle, Staphorst)
```

Figure 7: Console output Amsterdam-Staphorst on national roadmap