Algorithms and Data Structures

Graph Searches in a Route Planner

Assignment-5

Version: 21.1, December 16th 2021

Introduction

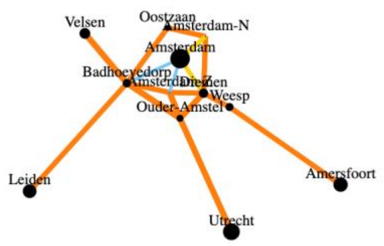
In this assignment you will develop an application that finds (optimal) routes across the Dutch national road's infrastructure from any given starting point to any given destination. You will implement three algorithms and compare results:

1. Depth-First Search
2. Breadth-First Search
3. Dijkstra's 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 to plot maps for easy visual verification of your solutions. You find **// TODO** comments in the classes **DirectedGraph**, **Junction** and **RouterPlannerMain** classes 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 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 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. 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)

The **RoadMap** class extends **DirectedGraph<Junction,Road>** and realises the graph implementation of a junctions and roads traffic network. By that you can use the generic search algorithms of DirectedGraph to calculate routes along Junctions and Roads.
In Figure 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.
- **boolean addEdge(fromVertex, toVertex, newEdge)** adds newEdge to the map of edges in between fromVertex and toVertex in the graph. If fromVertex or toVertex are not part of the graph yet, they shall be added. (If duplicates of fromVertex or toVertex are in the graph already, those shall be connected by newEdge.)
  If there is already an edge between fromVertex and toVertex nothing shall be added.
- **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. The same conditions as above apply (so this method should reuse addEdge(fromVertex, toVertex, newEdge)).
- **DGPath depthFirstSearch(startId, targetId)** and **DGPath breadthFirstSearch(startId, targetId)** search a path in the graph starting from the vertex identified by startId and ending at the vertex identified by 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.

The class **DirectedGraph.DGPath** 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.

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 definition of the Junction class 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 shortest path (in total length), stops when such path is found and also populates the visitedVertices in the path. Verify your results from the console statistics and in the .svg picture.

R6. Apply the implementation of DirectedGraph. dijkstraShortestPath to calculate both the **shortest path length** (in total distance) and the **fastest path traveltime** (in minimum total time, not exceeding any speed limits) between Amsterdam and Meppel. Verify your results from the console statistics and in the .svg picture.

R7. 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.dijkstraShortestPath to calculate the **fastest** alternate route (in minimum total time, not exceeding any speed limits). Verify your results from the console statistics and in the .svg picture.

R8. 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.

R9. Provide a report including
a) an explanation of which search algorithm has calculated the route in each of the three figures 1 – 3 below, referring to specific qualitative characteristics of each of the algorithms.
b) Explanation and justification of seven of your most relevant code snippets in your solution.
c) Full console output of the main program, and justification of the differences in your output with the reference console output that is provided below.
d) References to external sources that you may have consulted, if any.

Grading

For a sufficient grade, you shall have attempted all of above requirements, shall deliver all green unit tests and reproduce all results from below console output in Figure except those which are due to valid implementation variations or to at most two coding defects.

Appendices

Here you find four pictures of example solutions searching for a route from “Amsterdam” to “Staphorst”, each using a different algorithm from ‘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 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. You can review them from your own runs.

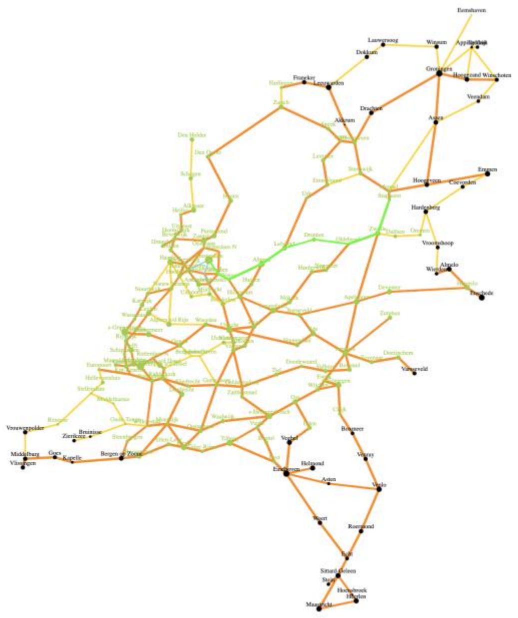


Figure 1: which search from Amsterdam to Staphorst?

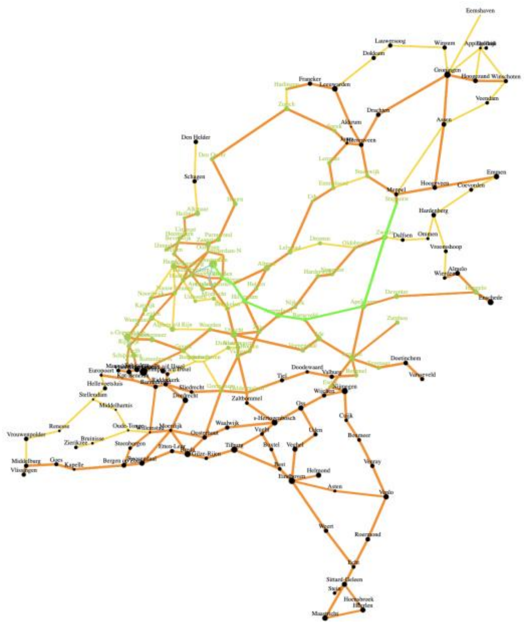


Figure 2: which search from Amsterdam to Staphorst?

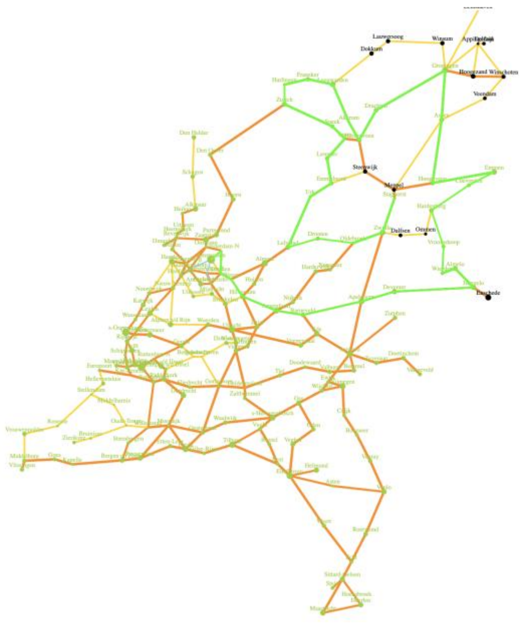


Figure 3: which search from Amsterdam to Staphorst?

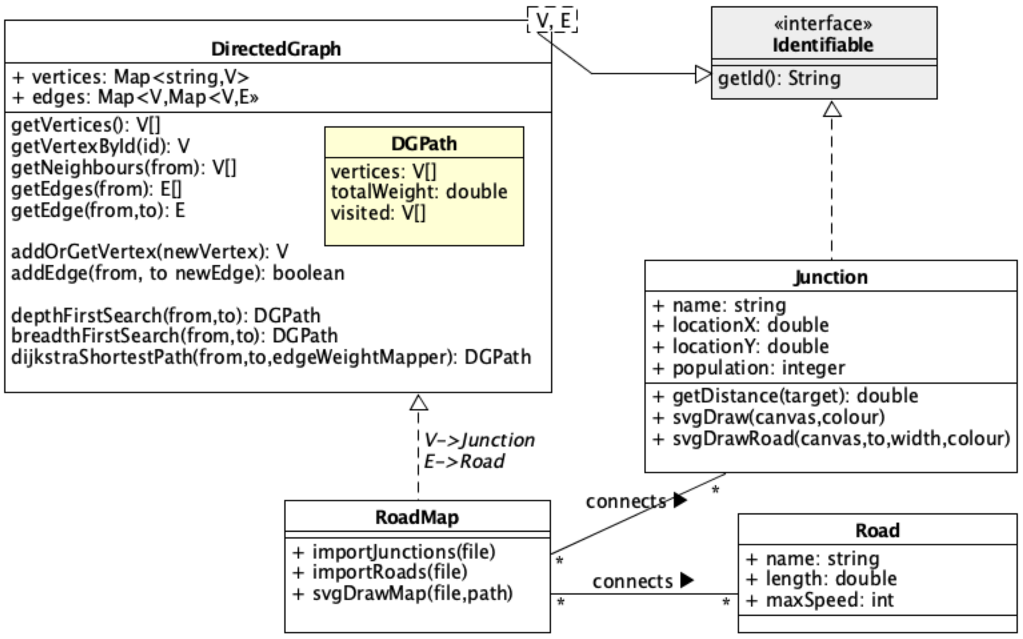


Figure 4: UML class diagram of starter project

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. Explain/justify such differences in your report.

```
Welcome to the HvA RoutePlanner

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: [Utrecht(A2/120),Diemen(A9/120),Amsterdam-Z(A2/120),Badhoevedorp(A9/120)],
  Badhoevedorp: [Leiden(A4/100),Ouder-Amstel(A9/120),Oostzaan(A10/100),Velsen(A9/120),Amsterdam-
    Z(A10/100),Amsterdam(S112/50)],
  Utrecht: [Ouder-Amstel(A2/120)],
  Amsterdam: [Amsterdam-N(S102/80),Oostzaan(S108/50),Diemen(S102/80),Amsterdam-
    Z(S108/50),Badhoevedorp(S112/50)],
  Diemen: [Ouder-Amstel(A9/120),Amsterdam-N(A10/100),Amsterdam-
    Z(A10/100),Weesp(A1/100),Amsterdam(S102/80)],
  Amersfoort: [Weesp(A1/100)],
  Amsterdam-Z: [Ouder-Amstel(A2/120),Diemen(A10/100),Amsterdam(S108/50),Badhoevedorp(A10/100)],
  Oostzaan: [Amsterdam-N(A10/100),Amsterdam(S108/50),Badhoevedorp(A10/100)],
  Leiden: [Badhoevedorp(A4/100)],
  Velsen: [Badhoevedorp(A9/120)],
  Weesp: [Diemen(A1/100),Amersfoort(A1/100)],
  Amsterdam-N: [Oostzaan(A10/100),Diemen(A10/100),Amsterdam(S102/80)]
}

Results from path searches from Oostzaan to Ouder-Amstel:
Depth-first-search: Weight=0,000000 Length=4 visited=4 (Oostzaan, Amsterdam-N, Diemen, Ouder-
    Amstel)
Depth-first-search return: Weight=0,000000 Length=4 visited=5 (Ouder-Amstel, Diemen, Amsterdam-N,
    Oostzaan)
```

```
Breadth-first-search: Weight=0,000000 Length=3 visited=8 (Oostzaan, Badhoevedorp, Ouder-Amstel)
Breadth-first-search return: Weight=0,000000 Length=3 visited=10 (Ouder-Amstel, Badhoevedorp,
    Oostzaan)
Dijkstra-Shortest-Path: Weight=21,232941 Length=4 visited=10 (Oostzaan, Amsterdam, Amsterdam-Z,
    Ouder-Amstel)
Dijkstra-Shortest-Path return: Weight=21,232941 Length=4 visited=12 (Ouder-Amstel, Amsterdam-Z,
    Amsterdam, Oostzaan)
Dijkstra-Fastest-Route: Weight=0,264123 Length=3 visited=10 (Oostzaan, Badhoevedorp, Ouder-
```

Figure 5: Console output small roadmap of Amsterdam area

```
    Amstel)


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.

Results from path searches from Amsterdam to Meppel:
Depth-first-search: Weight=0,000000 Length=46 visited=90 (Amsterdam, Amsterdam-N, Purmerend,
    Zaanstad, Beverwijk, Velsen, Zwanenburg, Haarlem, Hoofddorp, Uithoorn, Mijdrecht, Breukelen,
    Utrecht, Woerden, Alphen a/d Rijn, Leiden, Clausplein, Zoetermeer, Gouda, Schoonhoven,
    Gorinchem, Oosterhout, Breda, Gilze-Rijen, Tilburg, Eindhoven, Veghel, Uden, Oss, Tiel,
    Doodewaard, Valburg, Bemmel, Arnhem, Apeldoorn, Deventer, Hengelo, Almelo, Wierden,
    Vroomshoop, Hardenberg, Ommen, Dalfsen, Zwolle, Staphorst, Meppel)
Depth-first-search return: Weight=0,000000 Length=73 visited=89 (Meppel, Assen, Groningen,
    Drachten, Heerenveen, Joure, Lemmer, Emmeloord, Urk, Lelystad, Dronten, Oldebroek, Zwolle,
    Apeldoorn, Arnhem, Bemmel, Valburg, Doodewaard, Tiel, Oss, Uden, Veghel, Eindhoven, Best,
    Boxtel, Vught, Tilburg, Gilze-Rijen, Breda, Etten-Leur, Roosendaal, Bergen op Zoom, Kapelle,
    Goes, Middelburg, Vrouwenpolder, Renesse, Stellendam, Hellevoetsluis, Europoort, Kpt-Benelux,
    Vlaardingen, Schiedam, Rotterdam, Barendrecht, Willemstad, Moerdijk, Oosterhout, Waalwijk, s-
    Hertogenbosch, Zaltbommel, Geldermalsen, Gorinchem, Sliedrecht, Ridderkerk, Krimpen a/d
    IJssel, Capelle a/d IJssel, Rotterdam-A, Gouda, Woerden, Utrecht, Nieuwegein, Vianen, Houten,
    Zeist, Hilversum, Huizen, Almere, Weesp, Diemen, Ouder-Amstel, Amsterdam-Z, Amsterdam)
Breadth-first-search: Weight=0,000000 Length=9 visited=99 (Amsterdam, Diemen, Weesp, Almere,
    Lelystad, Urk, Emmeloord, Steenwijk, Meppel)
Breadth-first-search return: Weight=0,000000 Length=9 visited=85 (Meppel, Steenwijk, Emmeloord,
    Urk, Lelystad, Almere, Weesp, Diemen, Amsterdam)
Dijkstra-Shortest-Path: Weight=127,105064 Length=10 visited=138 (Amsterdam, Diemen, Weesp,
    Almere, Lelystad, Dronten, Oldebroek, Zwolle, Staphorst, Meppel)
Dijkstra-Shortest-Path return: Weight=127,105064 Length=10 visited=84 (Meppel, Staphorst, Zwolle,
    Oldebroek, Dronten, Lelystad, Almere, Weesp, Diemen, Amsterdam)
Dijkstra-Fastest-Route: Weight=1,201533 Length=9 visited=133 (Amsterdam, Diemen, Weesp, Almere,
    Lelystad, Urk, Emmeloord, Steenwijk, Meppel)

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

Process finished with exit code 0
```

Figure 6: Console output Amsterdam-Meppel on national roadmap