Project 5: Route Planner

The deadline for this project is **Monday, 8th of July**, end of day, anywhere on earth (AoE)[1]. You can find more information about the project submission in appendix A.

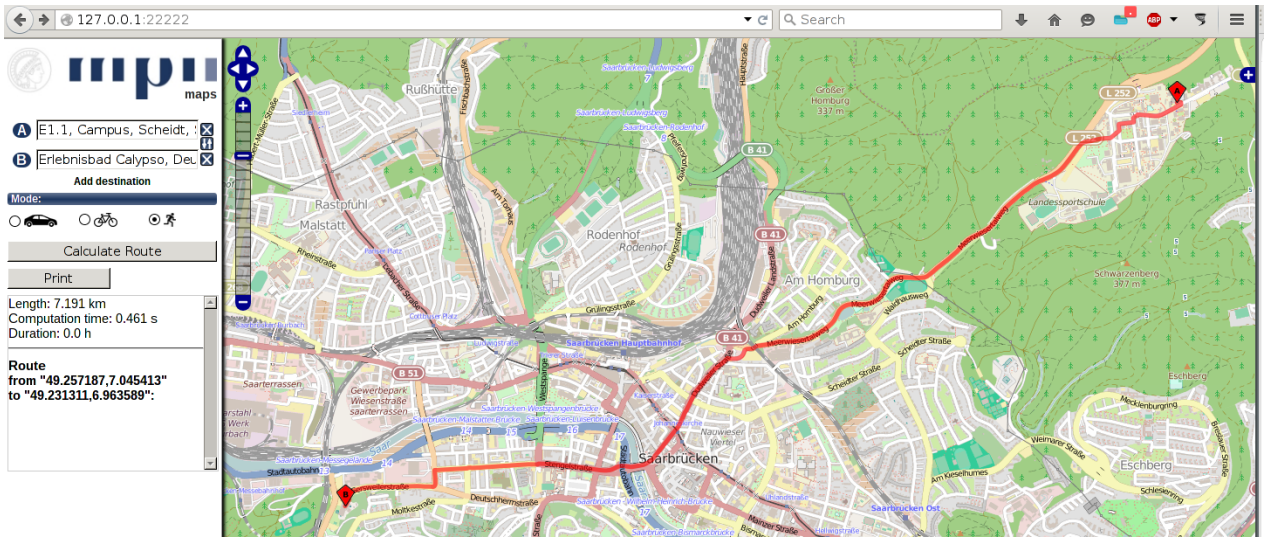


Figure 1: Browser frontend visualization of the calculated routes.

1 Overview

In this project, you will implement a route planner in Java. Given a means of transportation, the route planner shall calculate the shortest route between specified coordinates. Moreover, your route planner will also support intermediate coordinates which the route must visit.

Along with some maps, we will provide you with the front end depicted in fig. 1, as well as the components needed to be able to communicate with the front end. The front end was developed as a prototype in the software engineering[2] lecture from 2012, and altough it has been successfully used by other lectures at the university as debugging and visualization tool, it may not necessarily be free of minor bugs.

The navigation graph, which is encoded as a NAE file, uses Open Street Map (OSM)[3] identifiers (IDs) to identify the nodes. To visualize a given node on a map, you can use one of the many OSM web interfaces. As an example, the node with the ID 539518216 is visualized here:

`http://www.openstreetmap.org/node/539518216`

Maps necessary for the project are not part of your git repository, but must be downloaded from the dCMS[4] first. Once downloaded, extract the archived files into the root directory of your repository. To familiarize yourself with the project, we recommend you start by reading the JavaDoc comments in the provided `.java` files inside the `routing` package, as they are part of the project specification and complement this document.

Section 2 showcases the assignments that guide you towards a route planner implementation. Optional tasks aimed at speeding up your implementation are described in section 3. The appendix gives more technical information, documents the NAE file format, the maps we provide and the browser frontend you may use for visualization.

[1] `https://en.wikipedia.org/wiki/Anywhere_on_Earth`
[2] `https://www.st.cs.uni-saarland.de/edu/se/2012/`
[3] `http://www.openstreetmap.de/`
[4] `https://dcms.cs.uni-saarland.de/prog2_24/dl/98/Project_5_Maps_Archive.zip`

2 Assignments

The project consists of four mandatory assignments and two optional bonus assignments. In the following, we will describe each assignment in the recommended order in which you should implement them.

Assignment 1: Reading Map Files and Building Graph (4 Points)

The graphs provided in this project are encoded in a custom file format called NAE. The maps describe a navigation graph (`Graph.java`) of nodes (`Node.java`), which are connencted via edges (`Edge.java`). Your task is to implement the method `public static Graph createGraphFromMap(String filename)` of the `Factory.java` file. The variable `filename` is a file path pointing to a NAE input file. Parse the input file and return an instance of the `Graph` interface, representing the graph described by the file. You may find it helpful to create a helper class, e.g. `GraphFactory.java`, to implement this method.

Find the documentation of the NAE format in appendix B.

Notes

- If the specified file path does not exist, your implementation should throw a `FileNotFoundException`.
- In appendix C, you can find an overview of the maps we provide to you.
- To implement `createGraphFromMap` efficiently, you cannot use a linear search to find the corresponding source and destination nodes of an edge. Instead, use an appropriate data structure to access the corresponding object via the id of a node.
- For every edge from A to B, you must create **two** `Edge` objects: the 'to' edge and 'from' edge. The 'to' edge is added to the start node with source A and destination B, the 'from' edge is added to the target node with source B and destination A. Make sure that the 'to'-edge has exactly the inverse privileges as the 'from'-edge (see appendix B).
- Make sure to familiarize yourself with the `Graph` and `Node` classes before you start your implementation.

If you need a refresher on graph theory, consult the appendix of the third project description.

Assignment 2: Graph Simplification (2 Points)

The `Graph` interface has two methods with which the represented graph can be simplified. In your implementation of the interface, implement the methods

- `public int removeIsolatedNodes()` which removes all nodes that do not have an outgoing edge, and
- `public int removeUntraversableEdges(RoutingAlgorithm ra, TravelType tt)`, which specializes the graph for a given `RoutingAlgorithm ra` and `TravelType tt`. All edges which may not be used with the given mode of transportation `tt` must be removed. If the `RoutingAlgorithm` is bidirectional (cf., section 3), you must preserve those edges which may be used in the opposite direction. The method checks whether `ra` is bidirectional.

Notes

- Your routing algorithm should not use either of the two methods. Instead, the visualization (see appendix D) will use these methods to speed up the frontend.

Assignment 3: From Coordinates to Nodes (7 Points)

Though your phone's maps application can take you basically anywhere along the shortest path, when you open it and instruct it to "get me home", the app must first find out where you are in its navigation graph. Using the GPS module of your phone, you can only obtain coordinates, which first need to be translated into nodes of the graph.

In this assignment, you will implement the `NodeFinder` interface, which returns the closest node for a given coordinate. Since the number of nodes in the graph can be very large, computing the distance from the starting coordinate to all other nodes can take a long time and is too inefficient to find the closest node for a given coordinate. Instead, you will solve this problem by implementing an acceleration data structure, which is a data structure that preprocesses the data in such a way that some questions regarding the data are easier to answer.

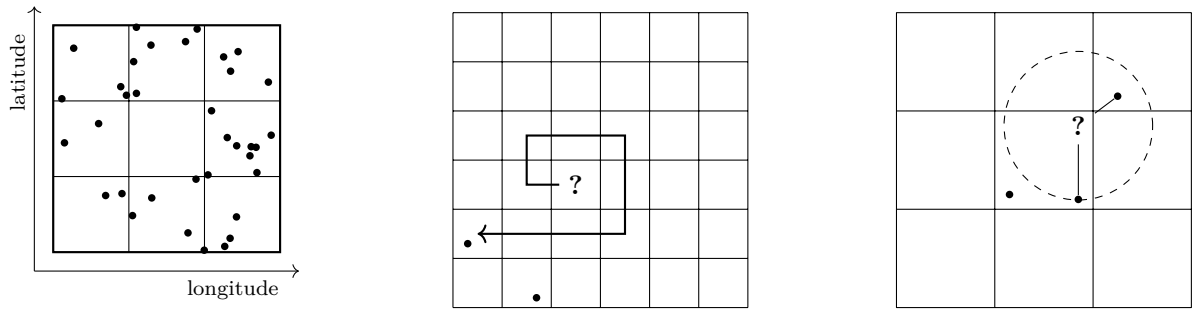
A simple acceleration strategy is to place a grid over the map and group all nodes within the same grid cell (see fig. 2a). To find the nearest node of a coordinate, one must find the corresponding grid cell(s) for that coordinate, which massively restricts the number of candidates for the closest node. We will now elaborate the steps necessary to create and use the acceleration structure.

Construction of the Acceleration Structure

To find the node with the closest distance to the initial coordinate, we will compute the distance between the given coordinate and all nearby nodes, returning the coordinate with minimial distance, if any. In order to keep this procedure efficient, we must ensure that the number of candidate nodes to compare is rather low.

Figure 2a shows an example grid imposed on a navigation graph. Note that the graph's edges are irrelevant for this problem, hence the depiction only shows the coordinates of the nodes. Your task is to design a data structure that, given an initial coordinate, returns all nodes inside the coordinate's cell.

Use the pre-implemented method `Coordinate.getDistance(Coordinate other)` to calculate the distance between two coordinates.



(a) Constructing the Grid (b) Finding a Candidate (c) Finding the Closest Node

Figure 2: Construction and Usage of the Acceleration Structure

Finding a Candidate

Notice that the acceleration structure may have cells that are completely empty. In order to find a candidate for the closest node, neighboring cells have to be searched. An example procedure for finding *some* cell close to the search coordinate **?** is depicted in fig. 2b Once any nearby node has been found, we can move to the next step: Ensuring that we find the node closest to the coordinate.

Finding the Closest Node

Once a candidate for the closest node has been found, we know that the true closest node is at most as far from the search coordinate as our candidate. The result is a region around the search coordinate in which the closest node can be found.

Figure 2c illustrates this behavior: Given the search coordinate **?** and the distance to the candidate, we must search all cells that intersect the resulting circle[5]. As depicted, it may not be sufficient to only scan the cell of the candidate node for the closest node. Notice how the closer our candidate node was to the search coordinate, the fewer nodes we have to consider, resulting in a faster algorithm.

For a given coordinate and a radius `distance`, the method `Coordinate.computeBoundingBox(double distance)` calculates intervals of longitudes and latitudes in which closer nodes may exist. Use this method to determine which cells must additionally be searched for closer nodes.

Remarks

- For this part of the project you must implement all methods of the `NodeFinder` interface.
- In the classes `CoordinateBox` and `Coordinate`, some methods are already pre-implemented which may be useful to you, in particular the method `Coordinate.computeBoundingBox`.
- The acceleration structure does not need to be able to handle empty graphs.
- Your implementation neither has to consider the possibility of poles, nor the jump from longitude 180°E to 180°W.
- It is part of the task to compute a suitable number of cells. This may have a big impact on the performance of the acceleration structure.

[5]Since the distance is defined geographically, the set of all points which are within a specific radius to the coordinate does not necessarily induce a circle. The pre-implemented method `Coordinate.computeBoundingBox(double distance)` considers this issue and provides a correct over-approximation.

Assignment 4: Calculating Shortest Paths (7 Points)

The essence of every route planner is the computation of shortest paths.

Many different algorithms exist for the computation of shortest paths. To implement the `RoutingAlgorithm` interface, you may freely choose any such algorithm. Your implementation is only required to compute the shortest path in the given amount of time. To avoid timeouts, we recommend you choose a standard algorithm which searches the graph intelligently. A simple algorithm is explained in the following and is also used as a basis for the reference implementation to estimate the required time.

Dijkstra's Algorithm

Dijkstra's Algorithm is an efficient algorithm to compute the shortest distance between two nodes.

For every node v, the algorithm maintains an approximation for the distance $d(v)$ from the initial node. Initially, this distance is 0 for the starting node and $+\infty$ for all other nodes. Repeatedly, the algorithm *visits* vertices v that has not been *visited* before and have minimal distance to the initial node. For each of its neighbors w_i reachable along the edge $v \to w_i$ with length $length(w_i)$, we update its distance from the initial node $d(w_i)$ to be the minimum $\min(d(w_i), d(v) + length(w_i))$, i.e. we update the distance to w_i if the path through v is shorter than the previously known path. Once we *visit* the target vertex, we have successfully found the shortest distance to it. If no route between start and target exists, the entire navigation graph will be explored.

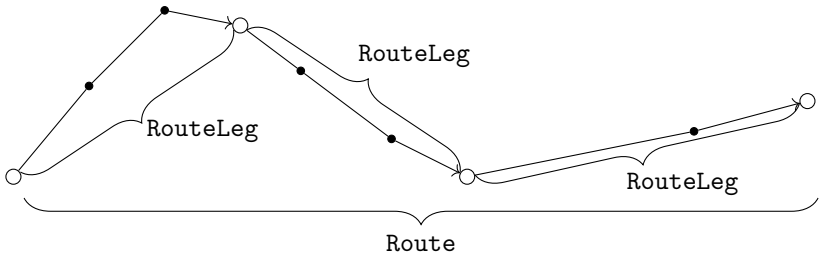
To always update the values of vertices with minimal combined cost, you should use the *priority queue* data structure. A priority queue allows you to efficiently maintain a sorted collection of objects, supporting removal of the smallest (or shortest) object. Moreover, you should maintain a *set* of vertices that you have already visited, to avoid visiting any vertex twice. Both data structures initially start empty and will be filled with neighboring nodes of the currently visited node.

Note that the distance value $d(v)$ stored for each vertex v is the combined length of the edges when following a path through the navigation graph. So far, we only discussed obtaining shortest distances from start to goal. To obtain the *shortest paths* from the initial vertex, we need to remember which vertex w caused the most recent update to the distance $d(v)$. This update implies that the shortest path to v uses the edge $w \to v$. To reconstruct the shortest path, we can follow those edges backwards up to the initial node.

Remarks: Do not implement a priority queue yourself, instead use the class `PriorityQueue`[6] from the package `java.util`. Please note that the objects you want to keep sorted must implement the interface `Iterable<T>`[7]. Before using this class, ask yourself what happens when the logical priority of an object changes while it is in the queue.

Modelling the Route

The interfaces `Route` and `RouteLeg` are used to represent the calculated route. We provide the abstract classes `RouteBase` and `RouteLegBase` to you, which already implement the functionality necessary for the web front end. You may extend these classes with your own implementation.



The `RoutingAlgorithm` interface has two methods which can be used to compute the routes: `computeRouteLeg` and `computeRoute`. These search for a direct route between two waypoints (`RouteLeg`) or for a route visiting multiple waypoints (`Route`). A `Route` consists of multiple `RouteLeg` objects, which describe the individual segments.

[6] `https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html`
[7] `https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html`

Notes

- A `RouteLeg` or `Route` over a sequence of `Node` objects is only consistent, if every `Node` object of the sequence except the last one contains an `Edge` which starts at this `Node` and ends with the next `Node`. Additionally, every single one of these `Edge` objects must be usable in the *forward direction*.
- Implement the interface `RoutingAlgorithm` and the method `createRoutingAlgorithm` in the class `Factory`.
- If you are not satisfied with the performance of the algorithm, you can implement an extension, for example the A^* algorithm. Though conceptually very similar to Dijkstra's algorithm, A^* uses a heuristic to navigate to the goal faster. As a heuristic, the distance to the target node (see the `Coordinate.getDistance` method) is feasible. The bonus exercises also deal with shortening the computation time.
- With Dijkstra's algorithm, you should pass all `public` and `local.fast` routing tests. It is possible to solve all `local.huge` and the biggest `eval` tests with Dijkstra's algorithm, though some optimizations might be required.

3 Bonus exercises

Assignment 5: Meet in the Middle (2 Bonus Points)

The `RoutingAlgorithm` interface and the accessibility restrictions along the edges in both directions allow for *Meet-in-the-Middle* or *bidirectional* solutions. In bidirectional routing, you don't only explore edges forwards to reach the goal from the start, but also follow edges in the opposite direction to reach the start from the goal. Combining the forwards and backwards exploration, one visits fewer nodes on average when computing shortest paths, especially in highly branching graphs.

In this bonus task, you shall implement a bidirectional routing algorithm (Dijkstra, A*, etc.). Note that in contrast to the unidirectional algorithm, the first found path from the starting node to the goal is not necessarily the shortest one. Before you start with this exercise, think about why this is the case (consider the minimal sample graph). Also think about how you deal with edges that are asymmetric, for example edges which can only be used by car in one direction.

Assignment 6: Overlay Graph Simplification (3 Points)

In the given navigation graphs, all nodes that are contained are also visible in OSM. While isolated nodes are already pruned out during graph simplification, there are still many nodes left which are more of a distraction. Such nodes have exactly two connections to the rest of the graph. During route search, such nodes are visited via one of these edges and left via the other edge. Afterwards, they can never be visited again, i.e., they only induce a connection between both of their neighbors. Especially in long and winding streets, there are many such nodes which make it possible to visualize the course of the road, however make navigating more expensive.

In this bonus task, you shall implement an overlay graph for the navigation graph, in which all of the nodes described above are removed (1 point). This means the overlay graph only contains crossings and dead ends. When removing these nodes pay close attention to update its adjacent edges, such that the connections between crossings are not lost (or falsified). Additionally, it should be possible to compute the shortest path between *arbitrary* nodes on the overlay graph, just like on the original graph (2 points). To this end, you need to use the original graph when computing the routes until an intersection is reached. Afterwards, the rest of the route can be computed from the overlay graph.

A Infrastructure, How to pass, Assorted Hints

Using dGit

Use git to clone your personal project repository from dGit, for example by entering

```
1 git clone ssh://git@dgit.cs.uni-saarland.de:2222/prog2/2024/students/project-5-{your matriculation number}
```

Submit your project using `git push` until Monday, 8th of July, end of day, anywhere on earth (AoE). If you need a refresher on how to use git, consult the materials from the Programming 2 Pre-Course or additional material online[8].

Grading

Similar to the way you obtain your project, you will submit the project through git. The last commit pushed to our server before the deadline will be considered for grading. Please ensure that the last commit actually represents the version of your project that you wish to hand in.

Once you `git push`ed your project to our server, we will automatically grade your implementation on our server and give you feedback about its correctness. You can view your latest test results by following the link to the Prog2 Leaderboard in dCMS Personal Status page. We use four types of tests: **public** tests, **local** tests, **regular** tests, and **eval** tests. The public and local tests are available to you from the beginning of the project in your project repository. You should use them to test your implementation locally on your system. Once you pass all public tests related to one assignment, we will run the regular tests for your implementation disclosing to you the names of the regular tests and whether or not you successfully pass them.

The local tests are not relevant for passing or points. Make use of them as you wish.

Eval tests are similar to regular tests, but run only once after the project deadline. They directly help determine the number of points your implementation scored.

Notes

(a) Attempts at tampering with our grading system, trying to deny service to, purposefully crash, or extract grading related data from it will (if detected) result in your immediate expulsion from the lecture and a notification to the examination board. In the past, similar attempts have led to us filing criminal complaints with the police.

(b) Download the map files from the dCMS, and extract them into the project directory.

(c) Please refer to the description of project 4 if you need help setting up Visual Studio Code for the project.

(d) As in the last project, the JavaDoc comments inside the `.java` files are part of the project specification and complement this project description.

(e) You may not alter the provided `Interfaces`, `Coordinate` and `CoordinateBox` classes, as they will be replaced by the original versions on the server.

(f) Keep your own implementations in the Java package `routing`.

(g) To prevent timeouts when running the tests please note the remarks about runtime efficiency for the project tasks.

(h) Avoid using the `Scanner` class to read the NAE files, as it may be too slow to read large files. Use `FileReader` and `BufferedReader`, or the new `java.nio.file.Files` API instead.

[8] Such as `https://git-scm.com/book/en/v2`

B The NAE File Format

A NAE file (**N**ode **A**nd **E**dge) consists of individual lines, where all characters use the ASCII enconding. Every line consists of individual elements, which are seperated by single spaces. The first element of every line encodes how the line is to be interpreted. There are two possibilities: If the line starts with `N`, it describes a node, if it starts with `E`, it describes an edge instead. How both types of lines are structured is described in fig. 3. The first columns describe the type, the second columns give a description.

Please note that `Bit` can represent the character '`1`' or the character '`0`'. A `1` means that the property holds, a `0` means that the property does not hold. `long` refers to a whole number (all node IDs are unsigned). `double` means a decimal number which fits into the range of the Java type with the same name. The decimal point is encoded with the character '`.`'.

`N`	First element of a node line
long	Node ID
double	Latitude
double	Longitude

`E`	First element of an edge line
long	ID of the start node
long	ID of the destination node
Bit	Forward-Passable by car
Bit	Backward-Passable by car
Bit	Forward-Passable by bike
Bit	Backward-Passable by bike
Bit	Forward walkable
Bit	Backwards walkable

Figure 3: Encoding of nodes and edges in the NAE file format.

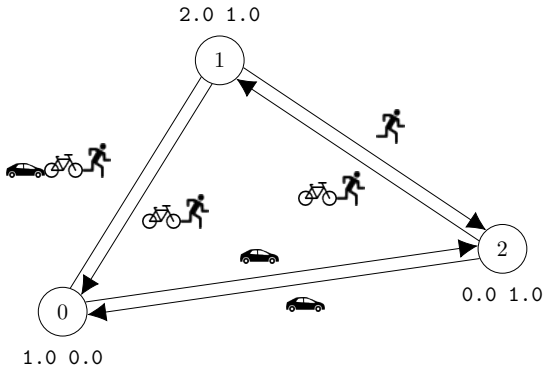
Example

Figure 4 depicts an example of a navigation graph. Its NAE encoding is depicted in fig. 4a. A node is described by its unique identifier and its `Coordinate` composed by latitude and longitude, which are shown inside and besides the nodes, respectively. The edges are described by the IDs of the nodes at their endpoints, as well as six bits describing the access rights for the modes of transport car, bike and walking. For example, line 2 of fig. 4a declares the node with ID `1` at coordinate $(2.0, 1.0)$, and line 6 declares the edge $1 \to 2$ to only be used when walking, but in the backwards direction $2 \to 1$ to be used by bike and walking.

The implementation of your method `Edge.allowsTravelType(TravelType tt, Direction dir)` must consider both the edge's direction `dir` and travel type `tt` of the intended means of transport. To distinguish between the different modes of transportation, we provide the `TravelType` enumeration type, representing car (`CAR`), bike (`BIKE`) and walking (`FOOT`). The extra value `TravelType.ANY` indicates that any edge may be taken, as long as the edge is accessible to *some* mode of transportation.

```
1 N 0 1.0 0.0
2 N 1 2.0 1.0
3 N 2 0.0 1.0
4
5 E 0 1 1 0 1 1 1 1
6 E 1 2 0 0 0 1 1 1
7 E 2 0 1 1 0 0 0 0
```

(a) Navigation graph in NAE format.



(b) Encoded navigation graph.

Figure 4: Navigation graph example.

Notes The following holds for all NAE files on which we will test your implementation:

- Among all nodes, the nodes' IDs are unique.
- All NAE files are sorted in such a way that all nodes are described first and edges are described afterwards.

C Maps

You can find the map data required for the public and locall tests in the materials section of the dCMS. With the exception of the map `minimal.nae` all maps have been created from publicly available OpenStreetMap data. During this process, it could have happened that some paths were not assigned the same access rights as in the original data due to the extreme amount of annotations.

We provide

- `minimal.nae`: a small test graph
- `campus.osm.nae`: University Campus
- `saarbruecken.osm.nae`: Saarbücken City
- `saarland.osm.nae`: Saarland

Please unpack the archive into the root directory of your project repository.

D Browser Frontend

We provide you with a simple web server which you can start locally on your computer to visualize your routing algorithm (see fig. 1). To this end, run the `main` method of the `utils/Server.java` file in VSCode and open the `http://127.0.0.1:22222`[9]with a web browser.

The server uses your `NodeFinder` implementation to first determine the nodes from given coordinates. Alternatively —or while you have not yet implemented the `NodeFinder`— you may use OSM identifiers as waypoints. After using your shortest path implementation, the resulting route will be displayed in your browser.

Note that you need a working internet connection to load the background graphics as well as the coordinates from OpenStreetMap.

[9]The port (here 22222) can be configured in `Server.java`. This enables you to run multiple serves at once and to compare the calculated routes.