Programmierung 2 (SS 2021)
Universität des Saarlandes
Fakultät MI
Compiler Design Lab

Projekt 5

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Thorsten Klößner, M. Sc.
Julia Wichlacz, M. Sc.
Maximilian Fickert, M. Sc.

Routenplaner (20 Punkte + 5 Bonuspunkte)

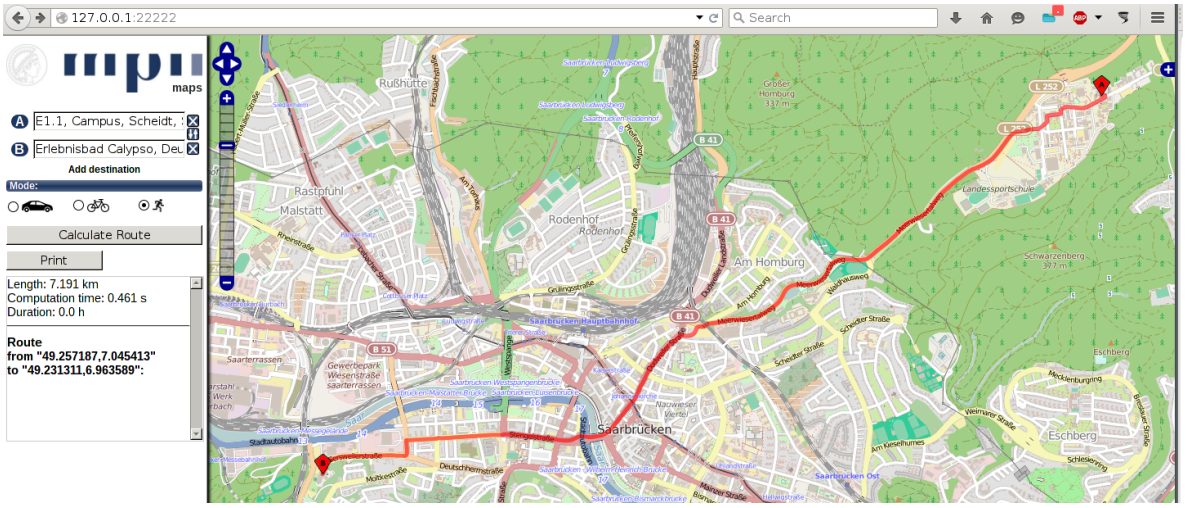


Abbildung 1: Browserfrontend zum Visualisieren der berechneten Routen.

In diesem Projekt werden Sie in Java einen Routenplaner implementieren. Dieser soll für gegebene Koordinaten die kürzeste Route berechnen, die man mit dem gewählten Fortbewegungsmittel benutzen darf. Neben den drei unterschiedlichen Fortbewegungsmitteln soll es auch möglich sein Zwischenpunkte anzugeben, über die die Route gehen muss.
Neben einigen Karten stellen wir Ihnen das Browserfrontend aus Abbildung 1, sowie die zur Kommunikation mit dem Frontend benötigten Teile zur Verfügung. Das Frontend wurde als Prototyp in der Software Engineering [3] Vorlesung 2012 entwickelt. Es diente schon den Gruppen der Algorithm Engineering [4] Vorlesung als Debug- und Visualisierungshilfe, ist aber nicht zwangsläufig frei von kleineren Fehlern und Problemen.
Der Navigationsgraph, der in den *NAE*-Dateien kodiert ist, benutzt *Open Street Map* (OSM) [2] "identifier" (ID), um die Knoten zu identifizieren. Um sich den ensprechenden Knoten auf einer Landkarte anzeigen zu lassen, können Sie die verschiedenen OSM-Webinterfaces benutzen. Als Beispiel ist der Knoten mit der ID 1597767486 unter folgender Adresse visualisiert:

`http://www.openstreetmap.org/node/1597767486`

Beachten Sie:

- Wie auch im letzten Projekt, sind die JavaDoc-Kommentare in den Interfaces Teil der Spezifikation und ergänzen somit dieses Dokument.
- Sie dürfen die mitgelieferten Interfaces weder verändern noch ergänzen, das gleiche gilt für die `Coordinate` und `CoordinateBox` Klassen. Falls Sie weitere Attribute oder Methoden benötigen, können Sie dies zum Beispiel mit einem `Adapter` [5] umsetzen. **Achtung!**
- Um Timeouts (und so mit Fehlschlagen) bei den Tests zu vermeiden beachten sie die Hinweise zu Effizienz und Laufzeit an den einzelnen Aufgaben.
- Für die Saarlandkarte wird ggf. mehr als der initial eingestellte Arbeitsspeicher der Virtuellen Maschine benötigt. Als Alternative zur Entwicklung in der VM können Sie auch Visual Studio Code nativ auf Ihrem System (Windows/Mac/Linux) installieren. Entscheidend ist am Ende, dass Ihr Code auf unserem Evaluationsserver die Tests besteht. Feedback dazu werden auch die Daily Tests liefern.

- Schreiben Sie Ihre Implementierung ausschließlich in das Java Paket `routing`.

1 Aufgaben

Das Projekt gliedert sich in vier Hauptaufgaben und optionale Bonusaufgaben. Als Hilfe erörtern wir diese Aufgaben im Folgenden in der Reihenfolge, die Sie auch beim implementieren befolgen sollten.
Sie können das Depot klonen in dem sie den Befehl

```
git clone https://prog2scm.cdl.uni-saarland.de/git/project5/$NAME
```

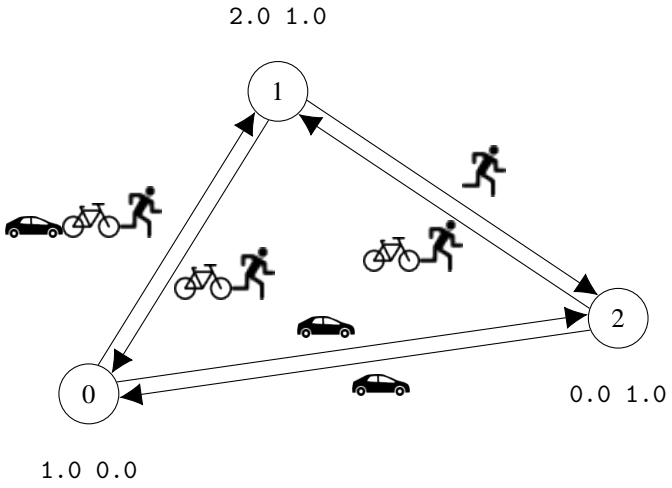
ausführen, wobei `$NAME` durch Ihren Benutzernamen zu ersetzen ist.

1.1 Karten einlesen und Navigationsgraph bauen (4 Punkte)

Wir stellen Ihnen Karten im speziell für dieses Projekt entworfenen *NAE*-Format bereit. Die Karten beschreiben einen Navigationsgraphen (Interface `Graph`) aus Knoten (Interface `Node`), die durch Kanten (Interface `Edge`) verbunden sind. Ihre Aufgabe ist es, die Methode `Factory.createGraphFromMap` zu implementieren. Sie dürfen dazu auch eine Hilfsklasse (z.B. `GraphFactory`) erstellen, falls es Ihnen sinnvoll erscheint.
Das Argument der `createGraphFromMap` Methode ist ein Pfad zu einer *NAE*-Datei, die Sie einlesen müssen. Als Rückgabewert wird ein `Graph` erwartet, der dem in der Datei kodierten Navigationsgraphen entspricht.

```
N 0 1.0 0.0
N 1 2.0 1.0
N 2 0.0 1.0
E 0 1 1 0 1 1 1 1
E 1 2 0 0 0 1 1 1
E 2 0 1 1 0 0 0 0
```

(a) Navigationsgraph im NAE-Format.



(b) Kodierter Navigationsgraph.

Abbildung 2: Beispielhafter Navigationsgraph.

Sie sehen einen beispielhaften Navigationsgraphen in Abbildung 2b. Die Kodierung des Beispielgraphen im NAE-Format sehen Sie in Abbildung 2a. Der Graph besteht aus Kanten, die Wege zwischen Knoten beschreiben. Die Position der Knoten ist in Breiten- und Längengraden angegeben. Die Klasse `Coordinate` stellt eine solche Koordinate dar. Da nicht alle Wege für alle Verkehrsmittel in beide Richtungen zugänglich sind, muss Ihre `Edge.allowsTravelType`-Implementierung berücksichtigen, in welche Richtung die Kante mit welchem Verkehrsmittel genommen werden soll. Wir unterscheiden zwischen drei Arten von Verkehrsmitteln, beschrieben durch den Enumerationstypen `TravelType`: Auto (`CAR`), Fahrrad (`BIKE`) und zu Fuß (`FOOT`). Der Enumerationstyp `TravelType` hat einen zusätzlichen Wert `ANY`, um anzugeben, dass jede Kante genommen werden darf, solange Sie mit irgendeinem Verkehrsmittel in die angegebene Richtung nutzbar ist.

Hinweise

- Wenn der gegebene Pfad nicht existiert, soll die Methode eine `FileNotFoundException` werfen.
- Das *NAE*-Dateiformat wird in Abschnitt 3 detailiert beschrieben.
- In Abschnitt 4 finden Sie eine Übersicht über die von uns bereitgestellten Karten.

- Um eine effiziente Einlese-Methode zu implementieren können Sie beim Einlesen der Kanten keine lineare Suche verwenden um den passenden Start- und End-Knoten einer Kante zu finden. Verwenden Sie stattdessen eine geeignete Datenstruktur um effizient mittels der Id eines Knoten auf das entsprechende Objekt zugreifen zu können.
- Für jede Kante von *A* nach *B* müssen **zwei** `Edge`-Objekte anlegt werden: eines wird in den Startknoten gehangen mit Start *A* und Ziel *B*, das andere mit Start *B* und Ziel *A* in den Endknoten. Beachten Sie dabei, dass die dem Endknoten hinzugefügte Kante genau die inversen Wegerechte der dem Startknoten hinzugefügten Kante haben muss. **Achtung!**
- Beachten Sie zum Aufbau ihrer Graphstruktur besonders die beiden Klassen `Graph` und `Node`.

1.2 Graph vereinfachen (2 Punkte)

Das `Graph`-Interface sieht zwei Methoden vor, mit denen der Navigationsgraph vereinfacht werden kann.

- `removeIsolatedNodes` entfernt alle Knoten aus dem Graphen, die keine ausgehenden Kanten haben.
- `removeUntraversableEdges` spezialisiert den Graphen für einen gegebenen `RoutingAlgorithm` und `TravelType`. Alle Kanten, die mit dem gegeben Fortbewegungsmittel nicht genutzt werden dürfen, müssen aus dem Graphen entfernt werden. Wenn es sich um einen bidirektionalen `RoutingAlgorithm` handelt (vgl. Abschnitt 2.1), müssen Sie Kanten erhalten, die in umgekehrter Richtung benutzt werden dürfen. Die Methode `RoutingAlgorithm.isBidirectional` gibt Auskunft, ob der `RoutingAlgorithm` bidirektional ist.

Hinweise Die beiden Methoden sollen nicht in ihrem Routing Algorithmus verwendet werden. Sie werden aktuell zur Darstellung mit Hilfe des Frontends genutzt.

1.3 Nächsten Knoten finden (7 Punkte)

Ein richtiger Routenplaner berechnet den kürzesten Weg auf Grundlage von Koordinaten und findet nahgelegene Graphknoten von selbst. In dieser Aufgabe werden Sie das `NodeFinder`-Interface implementieren, das zu einer Koordinate den nächstgelegenen Knoten liefert. Das Problem ist allgemein bekannt als *1-Nearest Neighbor*. Die Distanzmessung zwischen zwei Koordinaten ist in der Methode `Coordinate.getDistance` vorimplementiert.
Auf kleineren Navigationsgraphen kann man alle Knoten ablaufen, um so den nächsten Knoten zu finden. Auf großen Karten werden Sie damit jedoch Laufzeitprobleme bekommen. Daher werden Sie in dieser Aufgabe eine Beschleunigungsstruktur implementieren, mit deren Hilfe müssen Sie bei wiederholten Anfragen nach dem nächsten Knoten nicht mehr über alle Knoten iterieren.

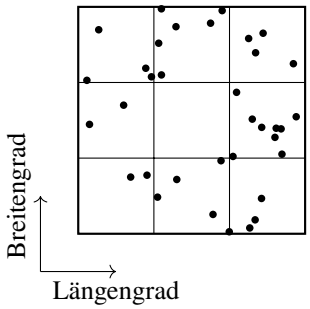
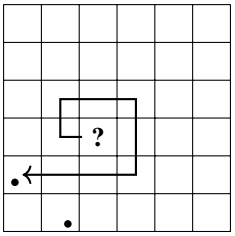


Abbildung 3: Knoten im Gitter.

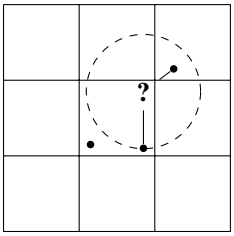
Eine einfache Beschleunigungsstruktur besteht darin, ein Gitter über die Karte zu legen und die Knoten in eine Menge pro Gitterzelle zu stecken. Ein Beispiel sehen Sie in Abbildung 3. Da die Kanten für dieses Problem keine Rolle spielen, sehen Sie dort nur die Punkte, welche die Knoten irgendeines Navigationsgraphen darstellen. Im Folgenden gehen wir auf den Aufbau der Gitterstruktur ein und besprechen die zwei Phasen des Suchalgorithmus auf der Beschleunigsstruktur (`NodeFinder.getNodeForCoordinates`).

Aufbau der Beschleunigungsstruktur

Mittels linearer Suche kann nach dem nächsten Knoten innerhalb einer Gitterzelle gesucht werden. Wählen Sie beim Aufbau des Gitters eine geeignete Anzahl an Zellen und sortieren Sie die Knoten in diese ein.



(a) Wenn die Zelle unter der angefragten Koordinate (**?**) leer ist, muss eine Zelle in der Nähe gefunden werden, die mindestens einen Knoten enthält. Der hier skizzierte Pfeil zeigt nur ein exemplarisches Vorgehen.



(b) Alle Zellen, die im Radius des nächsten bekannten Knoten liegen, könnten nähere Knoten zur Suchkoordinate beinhalten. Hier dargestellt als Kreis auf dem Koordinatenraum. [1]

Abbildung 4: Die zwei Phasen des Suchalgorithmus.

Einen nahen Knoten finden

In der ersten Phase wird ein Knoten gesucht, der nicht zwingend der nächste ist. Sie sehen ein exemplarisches Vorgehen in Abbildung 4a. Die zweite Phase arbeitet umso effizienter je näher diese ungefähre Lösung an der Suchkoordinate liegt.

Den nächsten Knoten finden

Die zweite Phase erhält die Distanz der ungefähren Lösung und untersucht alle Zellen, die nähere Knoten enthalten könnten. Angenommen ein Knoten und seine Distanz zur Suchkoordinate sind bekannt. Wie in Abbildung 4b zu sehen, ist es dann nicht ausreichend nur in der Zelle der ungefähren Lösung nach dem nächsten Knoten zu suchen. Vielmehr, kann die Suche nach dem nächsten Knoten lediglich auf die Knoten beschränkt werden, die innerhalb der Distanz der ungefähren Lösung liegen. Die Methode `Coordinate.computeBoundingBox` berechnet für eine Koordinate und einen Radius Intervalle von Breiten- und Längengraden, in denen noch nähere Knoten liegen könnten. Verwenden Sie diese Methode, um zu bestimmen, welche Zellen Sie noch auf nähere Knoten untersuchen müssen.

Hinweise

- Für diesen Projektteil sind alle Methoden des `NodeFinder` Interfaces zu implementieren.
- In den Klassen `CoordinateBox` und `Coordinate` sind bereits Funktionen implementiert die für Sie nützlich sein können, insbesondere `Coordinate.computeBoundingBox`.
- Die Beschleunigungsstruktur muss nicht mit dem leeren Graphen umgehen können.
- Ihre Implementierung muss weder den Fall berücksichtigen, dass eine Karte einen Pol enthält, noch den Sprung des Längengrades von 180°E auf 180°W.
- Eine geeignete Anzahl an Zellen zu bestimmen ist Teil der Aufgabe und kann große Auswirkungen auf die Effizienz der Beschleunigsstruktur haben.

1.4 Kürzeste Wege berechnen (7 Punkte)

Das Kernstück eines jeden Routenplaners besteht darin, kürzeste Wege (auch Pfade oder Routen) zu berechnen [7]. Auch in diesem Projektteil gibt es eine Vielzahl möglicher Lösungen. Diesmal ist es Ihnen aber weitestgehend freigestellt, welche davon Sie tatsächlich implementieren. Ihre Implementierung muss nur den kürzesten Weg in der gegeben Zeit finden. Um diese zeitlichen Begrenzungen nicht zu überschreiten, sollten Sie einen Standardalgorithmus implementieren, der geschickt den Graphen durchsucht. Ein einfacher Algorithmus ist im Folgenden erklärt und wird auch von der Referenzimplementierung als Basis genutzt, um die benötigte Zeit abzuschätzen.

[1] *Da die Distanz geographisch definiert ist, ergibt die Menge aller Punkte, die im Radius liegen, nicht unbedingt einen Kreis. Die von uns gestellte überaproximation mittels* `Coordinate.computeBoundingBox` *bezieht dies allerdings mit ein.*

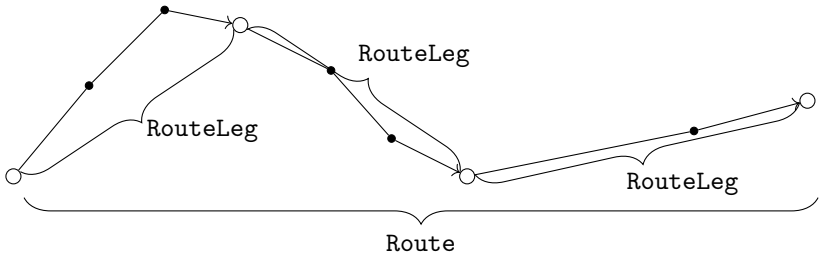
1.4.1 Der Algorithmus von Dijkstra

Der Algorithmus von Dijkstra [9](Skript Abschnitt 7.5) ist der schnellste bekannte Algorithmus um den kürzesten Pfad zwischen zwei Knoten zu berechnen[2].
Die Idee ist, den Graph vom Startknoten aus zu durchsuchen. Dabei werden immer nur unbesuchte Knoten betrachtet und ein vorläufig kürzester Weg berechnet. Sobald der Algorithmus bei dieser Suche den Zielknoten findet, kann man sich sicher sein, auch den kürzesten Weg gefunden zu haben.
In der Implementierung benutzt man hierzu sowohl eine Menge (*set*) sowie eine Prioritätswarteschlange (*priority queue*). Beide sind zu Beginn der Suche leer und werden mit den besuchten bzw. gefundenen Knoten während der Suche gefüllt. Ein Knoten wird *gefunden*, wenn er ein Nachbar des aktuellen Knotens ist, und man die Verbindung (`Edge`) mit dem gewählten Fortbewegungsmittel benutzen darf. Die Menge wird genutzt, um einen Knoten nicht doppelt zu besuchen; die Prioritätswarteschlange speichert die gefundenen Knoten sortiert nach ihrer Entfernung zum Startknoten.
Beachten Sie, dass die Entfernung hier nicht die Luftlinie zwischen den Knoten (also Entfernung der Koordinaten) ist, sondern die Länge des kürzesten Wegs durch den Navigationsgraphen. Wenn man die vom Startknoten aus transitiv erreichbaren Knoten nun in der Reihenfolge besucht, in der sie in der priority queue abgelegt sind, werden implizit die kürzesten Wege vom Startknoten zu allen besuchten Knoten berechnet. Falls der Zielknoten besucht wird, kann die Suche natürlich beendet werden. Falls es keine Route zwischen dem Start- und Zielknoten mit der gewählten Fortbewegungsart gibt, wird der gesamte vom Startknoten erreichbare Graph besucht.

Hinweis: Sie müssen eine *priority queue* nicht selbst implementieren, sondern können die Klasse `PriorityQueue` [11] aus dem Paket `java.util` verwenden. Beachten Sie, dass die Objekte die Sie sortieren möchten das Interface `Iterable<T>` [12] implementieren müssen. Machen Sie sich vorher klar, was passiert wenn der Wert, der zum Sortieren verwendet wird, sich ändert, während sich das Objekt noch in der priority queue befindet.

1.4.2 Modellierung der Route

Die Interfaces `Route` und `RouteLeg` sind dazu vorgesehen, eine berechnete Route zu beschreiben. Wir stellen Ihnen die abstrakten Klassen `RouteBase` und `RouteLegBase` bereit, die bereits für das Webfrontend nötige Funktionalität vorimplementieren. Sie können diese beiden Klassen durch Ihre Implementierung erweitern.



Es gibt zwei Methoden im `RoutingAlgorithm`-Interface, um Routen zu berechnen: `computeRouteLeg` und `computeRoute`. Diese spiegeln wieder, dass Nutzer nach einer direkten Route zwischen zwei Wegpunkten suchen können (`RouteLeg`) oder aber nach einer Route über mehrere vorgegebene Wegpunkte (`Route`). Eine `Route` setzt sich jedoch aus mehreren `RouteLeg`-Objekten zusammen, die die einzelnen Etappen beschreiben.

Hinweise

- Ein `RouteLeg` oder eine `Route` über eine Reihe von `Node`-Objekten ist nur konsistent, wenn jedes `Node`-Objekt der Reihe bis auf das letzte eine `Edge` enthält, die an dieser `Node` beginnt und auf der nächsten `Node` endet. Zusätzlich muss jede dieser `Edge`-Objekte in *Vorwärtsrichtung* verwendbar sein. **Achtung!**
- Implementieren Sie das Interface `RoutingAlgorithm` und die Methode `getRoutingAlgorithm` in der `Factory` Klasse.

[2] In Hinsicht auf die asymptotische Laufzeit ($\mathcal{O}$-Notation) und unter einigen Bedingungen an den Graphen.

- Falls Sie mit der Geschwindigkeit des Algorithmus nicht zufrieden sind, können Sie eine Erweiterung, z.B. den *A*-Algorithmus* [10] implementieren. Als Heuristik (Skript Abschnitt 7.5.1) bietet sich die Distanz zum Zielknoten an (siehe `getDistance` Methode der `Coordinate` Klasse).

  Auch die beiden Bonusaufgaben zielen auf eine verkürzte Rechenzeit ab.

- Mit *Dijkstra* sollten Sie alle *public* und *local.fast* routing-Tests bestehen. Für die *local.huge* Tests und manche der geheimen Tests benötigen Sie einen schnelleren Suchalgorithmus.

2 Bonusaufgaben

2.1 Bidirektionaler Dijkstra-Algorithmus (2 Punkte)

Das `RoutingAlgorithm`-Interface und die in beide Richtungen annotierten Zugangsrechte an den Kanten erlauben es, einen bidirektionalen Algorithmus zu benutzen. Bidirektional heißt, dass man Kanten nicht nur vorwärts folgt, um vom Start zum Ziel zu kommen, sondern auch Kanten in der Gegenrichtung durchläuft und somit vom Ziel her zum Start sucht. Da dies "gleichzeitig" passiert, muss man im Schnitt weniger Knoten besuchen um den kürzesten Weg zu finden.
In dieser Bonusaufgabe sollen sie Dijkstra (oder A*, etc.) nun bidirektional implementieren. Beachten Sie, dass im Gegensatz zum unidirektionalen Algorithmus, der erste gefundene Weg vom Start zum Ziel nicht zwangsläufig auch der kürzeste ist. Bevor Sie diese Aufgabe beginnen, machen Sie sich klar warum dies der Fall ist (ggf. an unserem minimalen Beispielgraphen). Überlegen sie sich ebenfalls wie sie mit asymmetrischen Kanten (z.B. in nur eine Richtung mit dem Auto befahrbar) umgehen.

2.2 Schnelles Routing durch Kreuzungsgraphen (1+2 Punkte)

In den bereitgestellten Navigationsgraphen sind alle Knoten, die auch auf OSM zu finden sind, eingetragen. Während bei der Vereinfachung des Graphen schon isolierte Knoten entfernt wurden, gibt es immer noch Knoten, die beim Finden des kürzesten Weges eher stören als helfen. Solche Knoten haben genau zwei Verbindungen zum Rest des Graphen. Bei der Routensuche werden solche Knoten über eine dieser Kanten besucht und über die andere wieder verlassen. Danach können sie nie wieder besucht werden, sie stellen also nur eine Verbindung zwischen ihren beiden Nachbarn her. Besonders in langen Straßen und Kurven gibt es viele solcher Knoten, da sie es erlauben, den echten Straßenverlauf zu visualisieren (als Alternative gäbe es eine gerade Linie zwischen ihren Kreuzungsknoten).
In dieser Bonusaufgabe sollen Sie einen Kreuzungsgraphen (eng. *overlay graph*) für den Navigationsgraphen berechnen, in dem alle oben beschriebenen Knoten entfernt wurden (1 Punkt). Das heißt, er enthält nur noch Kreuzungen und Sackgassen. Beim entfernen der Knoten müssen Sie beachten die Kanten anzupassen, so dass Verbindungen zwischen Kreuzungen nicht verloren gehen (oder verfälscht werden). Zusätzlich sollte es möglich sein auf diesem Kreuzungsgraphen, genau wie auf dem Original, die kürzeste Route zwischen *beliebigen* Punkten zu berechnen (2 Punkte). Hierzu müssen Sie zunächst den originalen Graphen zur Berechnung der Routen bis zu den nächsten Kreuzung nutzen. Anschließend kann der Rest der Route auf dem Kreuzungsgraphen ermittelt werden.

3 Das *NAE*-Dateiformat

Eine *NAE* Datei (**N**ode **A**nd **E**dge) ist aus einzelnen Zeilen aufgebaut, wobei alle Zeichen in ASCII kodiert sind. Jede Zeile besteht aus einzelnen Elementen, die jeweils durch ein Leerzeichen voneinander getrennt sind. Das erste Element jeder Zeile kodiert wie die Zeile zu interpretieren ist. Es gibt zwei Möglichkeiten: Beginnt die Zeile mit einem `N`, so handelt es sich um die Beschreibung eines Knotens (Node), beginnt sie mit einem `E`, handelt es sich um die Beschreibung einer Kante (Edge).
Wie die beiden Zeilentypen genau aufgebaut sind, wird in den Tabellen 1a und 1b beschrieben. Dort gibt die erste Spalte den Typ an, neben einer Beschreibung in der zweiten Spalte.
Beachten Sie, dass hierbei `Bit` für das Zeichen `1` oder `0` stehen kann. Eine `1` bedeutet, dass die Eigenschaft gilt, eine `0`, dass sie nicht gilt. `long` steht für eine Ganzzahl (alle Knoten IDs sind vorzeichenlos angegeben). `double` beschreibt eine Kommazahl, die in den Wertebereich des gleichnamigen Javatyps passt. Hierbei ist die Kommastelle mit einem Punkt '`.`' kodiert.

Um Ihnen das Einlesen zu vereinfachen, sind *NAE* Dateien zusätzlich noch so sortiert, dass zuerst alle Knoten beschrieben werden und dann erst die Kanten.

`N`	Erstes Element einer Knotenzeile
long	Knoten ID
double	Breitengrad
double	Längengrad

Beispiel:

```
N 0 1.0 2.0
```

Knoten (`N`) mit ID (`0`) an Koordinate (`1.0 2.0`)

(a) Aufbau einer Knotenzeile.

`E`	Erstes Element einer Kantenzeile
long	ID des Startknotens
long	ID des Endknotens
Bit	Vorwärts mit dem Auto befahrbar
Bit	Rückwärts mit dem Auto befahrbar
Bit	Vorwärts mit dem Fahrrad befahrbar
Bit	Rückwärts mit dem Fahrrad befahrbar
Bit	Vorwärts begehbar
Bit	Rückwärts begehbar

Beispiel:

```
E 4 5 0 0 1 0 1 1
```

Kante (`E`) ausgehend vom Knoten mit der ID (`4`) zum Knoten mit der ID (`5`) Die Kante ist nicht mit dem Auto (`0 0`), und nur vorwärts mit dem Fahrrad (`1 0`) befahrbar. Außerdem ist sie in beide Richtungen begehbar (`1 1`).

(b) Aufbau einer Kantenzeile.

Tabelle 1: Kodierung von Knoten und Kanten im NAE Dateiformat.

Hinweise Folgendes gilt für alle NAE-Dateien, auf denen wir Ihre Implementierung testen werden:

- Eine Knoten-ID taucht niemals in mehreren Knotenzeilen auf.
- Die Kantenzeilen beziehen sich nur auf Knoten, die zuvor per Knotenzeile eingeführt wurden.

4 Karten

Die Kartendaten finden Sie auf der Webseite unter Materialien im Unterpunkt Projekte [1]. Mit Ausnahme der `minimal.nae` sind die Karten aus den öffentlich zugänglichen OpenStreetMap [2] Daten erstellt worden. Bei diesem Prozess kann es passiert sein, dass Wege aufgrund der unüberschaubaren Anzahl an Annotationen nicht die gleichen Zugangsrechte wie in den originalen OSM Daten haben.

- `minimal.nae` - ein minimaler Testgraph
- `campus.osm.nae` - Uni Campus
- `saarbruecken.osm.nae` - Stadtgebiet Saarbrücken
- `saarland.osm.nae` - Saarland

5 Tests

In diesem Projekt gibt es folgende Testkategorien

- **Public tests** - Sie müssen alle Tests dieser Kategorie bestehen, um Punkte zu erlangen.
- **Local tests** - Diese Tests sind nicht Voraussetzung, um Punkte zu erhalten. Sie stehen Ihnen zur freien Verfügung. Sie befinden sich im Ordner `routing/tests/local`
- **Daily tests**
- **Evaluation tests** - Wir führen wir diese Tests nach der Abgabe aus, um Ihre Implementierung zu bewerten. Die anderen Kategorien geben Ihnen eine Orientierung, wie diese Tests aussehen werden.

6 Browser Frontend

Wir stellen Ihnen einen einfachen Server bereit, den Sie lokal auf Ihrem Computer starten können. Führen Sie dafür die `main` Methode in der `utils/Server.java` in Visual Studio Code aus und öffnen Sie mit einem Webbrowser die URL `http://127.0.0.1:22222` [3].
Der Server verwendet Ihre Implementierung, um zuerst aus Koordinaten Knoten zu ermitteln [4] (vgl. Abschnitt 1.3) und danach die kürzeste Route zwischen diesen Knoten zu berechnen (vgl. Abschnitt 1.4). Anschliessend wird die Route in Ihrem Browser angezeigt.
Beachten Sie, dass Sie eine Internetverbindung benötigen, um die Hintergrundgrafiken sowie die Koordinaten von OpenStreetMap [2] zu laden.

7 Abgabe

Die Abgabe erfolgt durch das Einspielen in das zur Verfügung gestellte Versionsverwaltungssystem. Die Deadline für die finale Abgabe ist der *9. Juli, 23:59 Uhr*. In die Bewertung fließen außer den öffentlichen Tests auch die daily und secret Tests ein. Zur finalen Bewertung ist ausschließlich der letzte im Versionsverwaltungssystem abgelegte Stand Ihres Codes entscheidend. Denken Sie daran Ihr Projekt nicht nur am Ende, sondern regelmäßig zu pushen damit ihr Projekt auf unseren Servern gesichert ist und Sie Feedback von den daily Tests erhalten.

8 Hinweise

- Legen Sie keine Dateien im Ordner `prog2` an. Dieser wird von unseren automatischen Tests überschrieben.
- Klonen Sie das Projekt in einen beliebigen Ordner:
  `git clone https://prog2scm.cdl.uni-saarland.de/git/project5/$NAME project5`
  wobei Sie $NAME durch ihren Benutzernamen ersetzen müssen.
- Laden Sie sich die Kartendaten von der Webseite herunter (unter Materialien), und entpacken Sie diese in das Verzeichnis wohin Sie das Projekt geklont haben.
- Entnehmen Sie der Projektbeschreibung von Projekt 4 (2048), wie Sie das Projekt in Visual Studio Code einrichten können.

Literatur

[1] `https://cms.sic.saarland/prog2_21/materials/`

[2] `http://www.openstreetmap.de/`

[3] `https://www.st.cs.uni-saarland.de/edu/se/2012/`

[4] `http://resources.mpi-inf.mpg.de/departments/d1/teaching/ss12/alg_eng/`

[5] `https://de.wikipedia.org/wiki/Adapter_(Entwurfsmuster)`

[6] `https://en.wikipedia.org/wiki/Nearest_neighbor_search`

[7] `https://en.wikipedia.org/wiki/Shortest_path_problem`

[8] `https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Math.html`

[9] `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`

[10] `https://en.wikipedia.org/wiki/A*_search_algorithm`

[3] Der Port (hier 22222) kann in der Server.java angepasst werden. Dies erlaubt es Ihnen mehrere Server gleichzeitig zu starten und die berechneten Routen visuell zu vergleichen.

[4] Sie können alternativ auch direkt OSM identifier als Wegepunkte angeben und somit den NodeFinder umgehen.

[11] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/PriorityQueue.html

[12] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Iterable.html