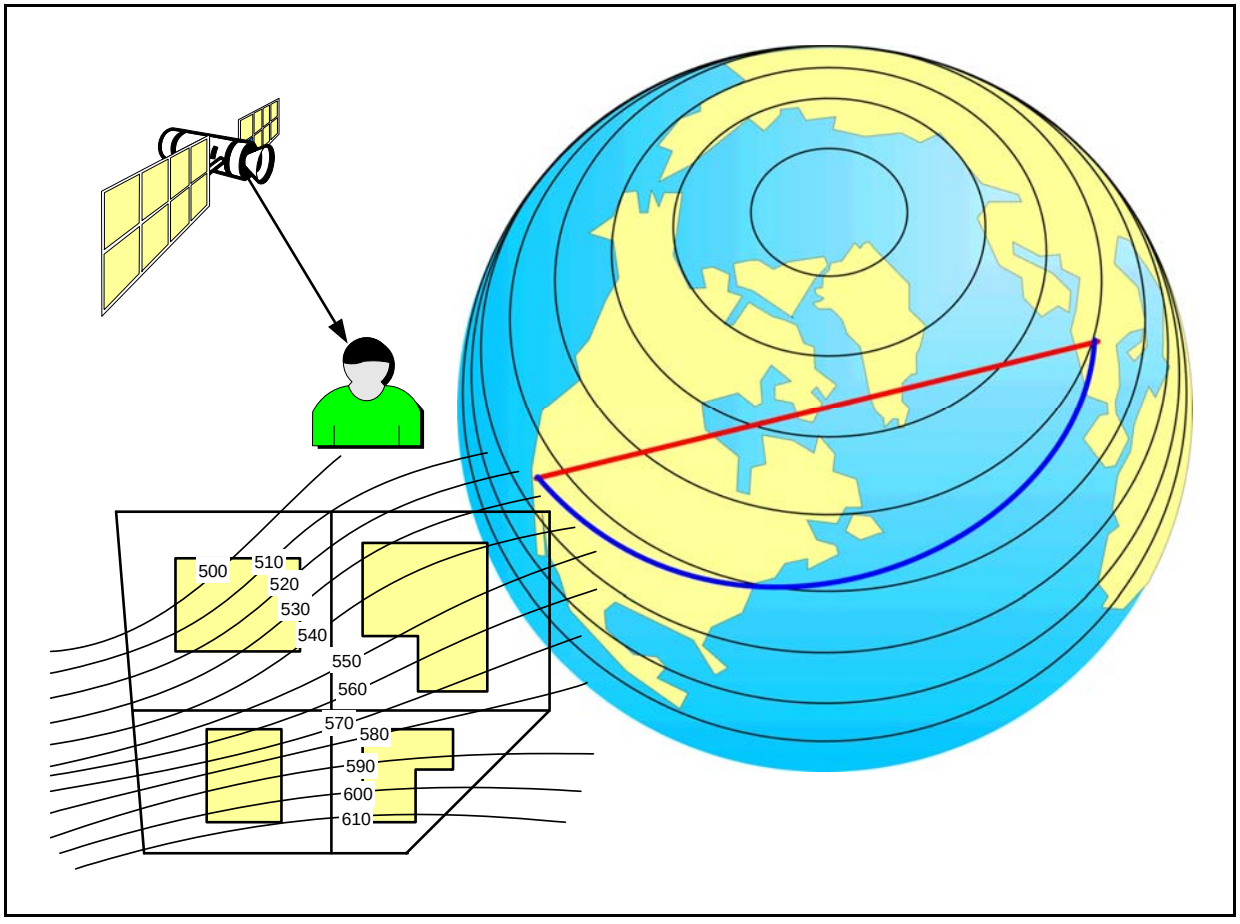
Ortsbezogene Anwendungen und Dienste

Praktikumsaufgabe im Sommer 2017

Jörg Roth



Ortsbezogene Anwendungen und Dienste
- Praktikumsaufgabe im Sommer 2017

Prof. Dr. habil. Jörg Roth
TH Nürnberg
Fakultät Informatik
Hohfederstraße 40
90489 Nürnberg

Einleitung

In diesem FWPF soll ein Programm entwickelt werden, das einen ortsbezogenen Dienst auf einem Geodaten-Bestand zur Verfügung stellt.

- Der Datenbestand besteht aus einem Straßennetz. Es wird eine Bibliothek zur Verfügung gestellt um auf die Daten des Straßennetzes zuzugreifen.
- Zusätzlich zu dem Straßennetz ist ein weiterer Bestand von Geodaten auf einem Datenbankserver der TH abgelegt, der von überall aus (per VPN) zugreifbar ist. Die Datenbank umfasst derzeit ca. 50 Mio. Geo-Objekte.
- Es steht ein Werkzeug zur Darstellung von Karten zur Verfügung.
- Die Entwicklung erfolgt in Java. Es sind keine weiteren Software-Bibliotheken zugelassen außer der zur Verfügung gestellten Bibliothek sowie die Java-Standard-Packages (Standard Edition). Darüber hinaus ist das Kopieren von Teillösungen aus anderen Quellen nicht erlaubt.
- Die Realisierung erfolgt in *Zweier-Gruppen*, bei ungerader Teilnehmerzahl zusätzlich mit einer Dreiergruppe.

Aufgabenstellung

Das Programm soll folgende Frage beantworten:

Gegeben eine Position: welche Objekte eines bestimmten Typs erreiche ich, wenn ich nicht länger als x Minuten fahren möchte?

Solch eine Fragestellung könnte Bestandteil eines größeren Dienstes sein, z.B. in einem Reise-Portal. Konkrete Anfragen könnten sein:

- *Gebe alle Hotels innerhalb von 10 Fahrminuten von meiner aktuellen Position zurück.*
- *Ich habe nur noch Benzin für 30 Fahrminuten; gebe alle Tankstellen zurück, die ich noch erreichen kann.*
- *Mein Auto macht Probleme, welche Reparaturwerkstätten liegen max. 5 Fahrminuten entfernt.*

Das zu entwickelnde Programm soll in den folgenden Schritten die Ausgabe produzieren (Abb. 1):

- Zuerst werden anhand des gegeben Ausgangspunktes und der maximalen Fahrzeit alle Straßenpunkte berechnet, die innerhalb der Fahrzeit erreichbar sind.
- Von dieser Menge wird eine *konkave* Hülle berechnet (nicht zu verwechseln mit der bekannteren *konvexen* Hülle). Hierzu existiert eine Bibliotheksfunktion.
- Damit erhält man eine polygonale Geometrie, mit der die Geodatenbank nach den gesuchten Objekten abgefragt werden kann.
- Das Resultat soll schließlich auf einer Karte visualisiert werden. Die Visualisierung soll dabei sowohl die konkave Hülle als auch die gesuchten Objekte mit Namen darstellen.

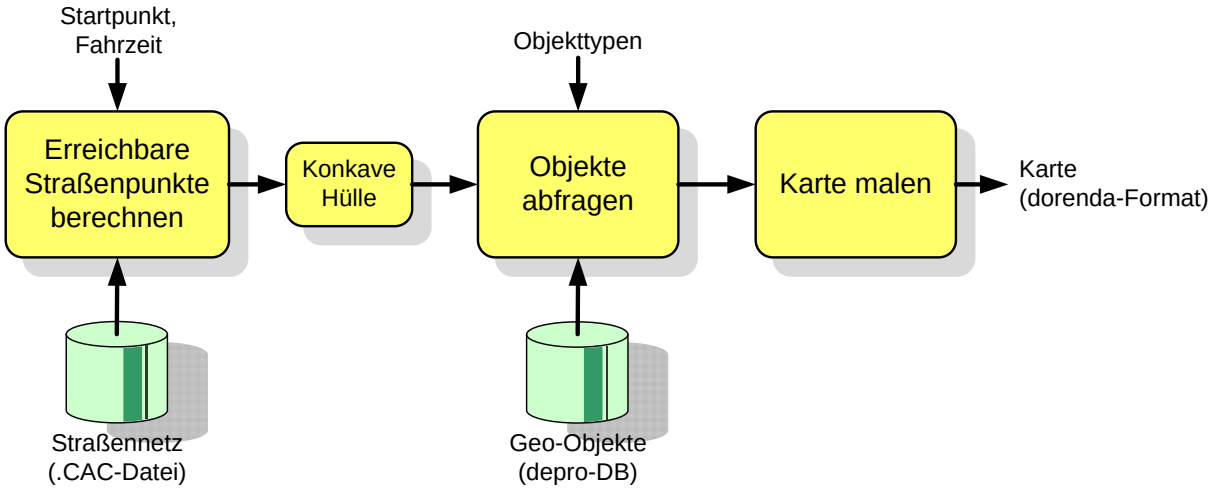


Abb. 1: Datenfluss des zu entwickelnden Programms

Das Straßennetzwerk

Das Straßennetz wird durch drei Entitäten repräsentiert (Abb. 2):

- *Domains*: das sind Straßen oder Wege die für Fahrer als "Straße" im weitesten Sinne wahrgenommen werden also z.B. die Hohfederstraße oder die A3. Häufig wird eine Straße durch eine einzige Domain repräsentiert – eine Unterteilung in den Daten in mehrere Domains erfolgt nur dann, wenn sich innerhalb deiner Straße wichtige Eigenschaften ändern, beispielsweise der Straßentyp oder Geschwindigkeitslimit.
- *Crossings*: das sind Punkte, an denen sich Straßen schneiden, d.h. man kann an diesen Punkten abbiegen.
- Links: das sind Abschnitte zwischen je zwei Crossings. Man kann Links als Teile von *Domains* auffassen, die durch die Crossings in Abschnitte geteilt werden. Jeder Link hat allerdings eine Fahrrichtung, d.h. eine Domain, die beide Fahrtrichtungen abbildet, hat pro Abschnitt jeweils zwei Links. Der Link, der dieselben Crossings in der umgekehrten Richtung verbindet wird *Reverse Link* bezeichnet.

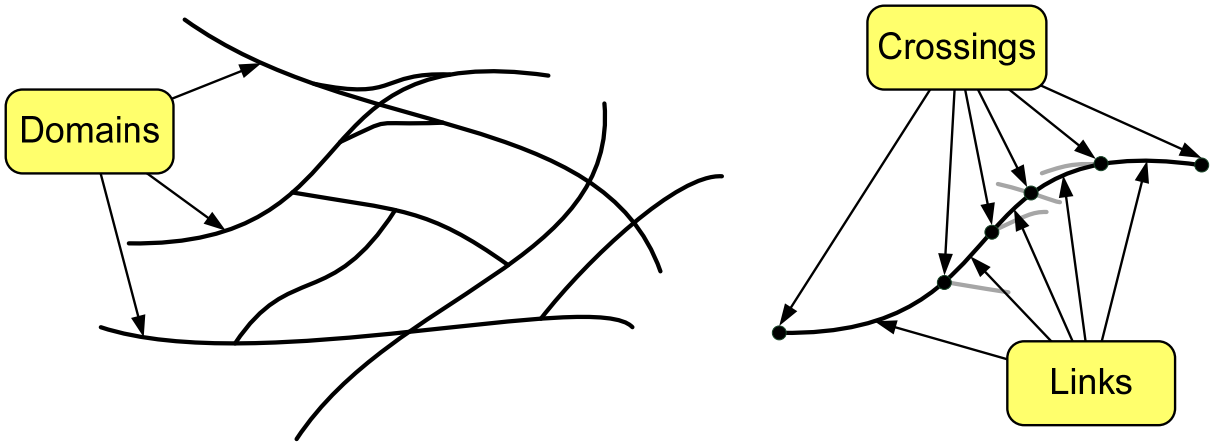


Abb. 2: Die Entitäten des Straßennetzwerks

Einige Besonderheiten zu Crossings und Links:

- Damit jeder Link genau zwei Crossings verbindet, werden auch die Endpunkte von Sackgassen auch durch Crossings modelliert.
- Es gibt auch Crossings ohne Abbiegemöglichkeit, d.h. man kann hier nur in eine Richtung weiterfahren (oder wenden). Das ist notwendig, wenn zwei zusammenhängende Links ohne Abbiegemöglichkeit zu verschiedenen Domains gehören. Als

Beispiel: eine Landstraße wird bei der Weiterfahrt zu einer Innerortstraße: dann kann man zwar nicht abbiegen, aber die Nahtstelle zwischen beiden Straßen wird durch eine Crossing modelliert.

Das gesamte Straßennetz von Deutschland bestehend aus Domains, Crossings und Links liegt in Form einer Datei vor. Eine API (bestehend aus der Klasse **`nav.NavData`**) erlaubt es, diese Datei einzulesen und auf die Einträge zuzugreifen.
Aus Effizienzgründen werden die jeweiligen Entitäten nicht durch eigene Klassen und Instanzen repräsentiert. Bei der großen Zahl von Einzelobjekten (mehrere Million) wäre der Aufwand (sowohl Speicher als auch Laufzeit) viel zu groß. Daher erlaubt die API, über IDs auf die Einträge der jeweiligen Objekte zuzugreifen. Die folgende Tabelle zeigt, welche Eigenschaften die jeweiligen Entitäten haben, und wie man darauf zugreift.

Entität, Eigenschaft	**Zugriff**
Domain	
Der Straßenname	`String getDomainName(int domainID)`
Die Domaingeometrie, d.h. die Positionen des Straßenverlaufs (linear verbunden)	`int[] getDomainLatsE6(int domainID)` `int[] getDomainLongsE6(int domainID)`
Link	
ID der Start-Crossing	`int getCrossingIDFrom(int linkID)`
ID der Ziel-Crossing	`int getCrossingIDTo(int linkID)`
Die zugehörige Domain	`int getDomainID(int linkID)`
Die Nummer der ersten Position in der Domaingeometrie	`int getDomainPosNrFrom(int linkID)`
Die Nummer der letzten Position in der Domaingeometrie	`int getDomainPosNrTo(int linkID)`
Länge des Links in Metern	`int getLengthMeters(int linkID)`
Straßentyp des Links (siehe unten)	`int getLSIclass(int linkID)`
Geschwindigkeitslimit (oder 0 wenn kein explizites)	`int getMaxSpeedKMperHours(int linkID)`
Winkel um in den Link reinzufahren	`int getNorthAngleFrom(int linkID)`
Winkel um aus dem Link herauszufahren	`int getNorthAngleTo(int linkID)`
Reverse Link eines Links	`int getReverseLink(int linkIndex)`
Geht dieser Link gegen die Einbahnstraße?	`boolean goesCounterOneway(int linkID)`
Crossing	
Die Position der Kreuzung	`int getCrossingLatE6(int crossingID)` `int getCrossingLongE6(int crossingID)`
Alle Links, in die aus dieser Kreuzung aus hereingefahren werden kann.	`int[] getLinksForCrossing(int crossingID)`
Ist diese Kreuzung überhaupt mit dem Straßennetz verbunden?	`boolean isIsolatedCrossing(int crossingID)`

Eine Reihe weiterer Methoden gibt Auskunft über den Datenbestand:

Eigenschaft	Zugriff
Anzahl der Links insgesamt; die Link IDs gehen von 0 bis Anzahl-1	`int getLinkCount()`
Anzahl der Crossings insgesamt; die Crossing IDs gehen von 1 bis Anzahl-1 (Achtung: ID=0 ist nicht belegt)	`int getCrossingCount()`
Die kleinste ID einer Domain	`int getDomainMinID()`
Die größte ID einer Domain	`int getDomainMaxID()`
Nicht alle Domain ID sind belegt	`boolean isDomain(int domainID)`
Gebe die nächste Kreuzung zu einer Position zurück	`int getNearestCrossing(int lat, int lon)`

Straßentypen (z.B. Autobahn, Innerortsstraße, Feldweg) werden durch so genannte *LSI-Klassen* definiert. Das sind erstmal Nummern. Unter dieser können allerdings weitere Eigenschaften abgefragt werden.

Eigenschaft	Zugriff
`LSIClassCentre`	
Ein LSIClass-Objekt zu einer Klassennummer	`LSIClass lc= LSIClassCentre.lsiClassByID(int lsiClass)`
Die Klassennummer zu einem Token	`int lsiClass= lsiClassByToken(String token)`
`LSIClass`	
Das Klassentoken	`String lc.classToken`
Der Klassenname	`String lc.className`
Die Klassennummer	`int lc.lsiClass`

Isochrone

Gesucht wird zuerst die Fläche aller erreichbaren Straßenpunkte, deren Distanz vom Start aus ein bestimmtes Kostenmaß nicht überschreiten. Hierbei interessiert besonders die Grenze, die diese Fläche umschließt. Diese Grenzelinie markiert die Fahrziele, die exakt durch Fahrten mit *x* Minuten erreicht werden. Solche Linien werden *Isochrone* genannt.
Abb. 3-5 zeigt die Isochronen für einen bestimmten Startpunkt für verschiedene Fahrzeiten.

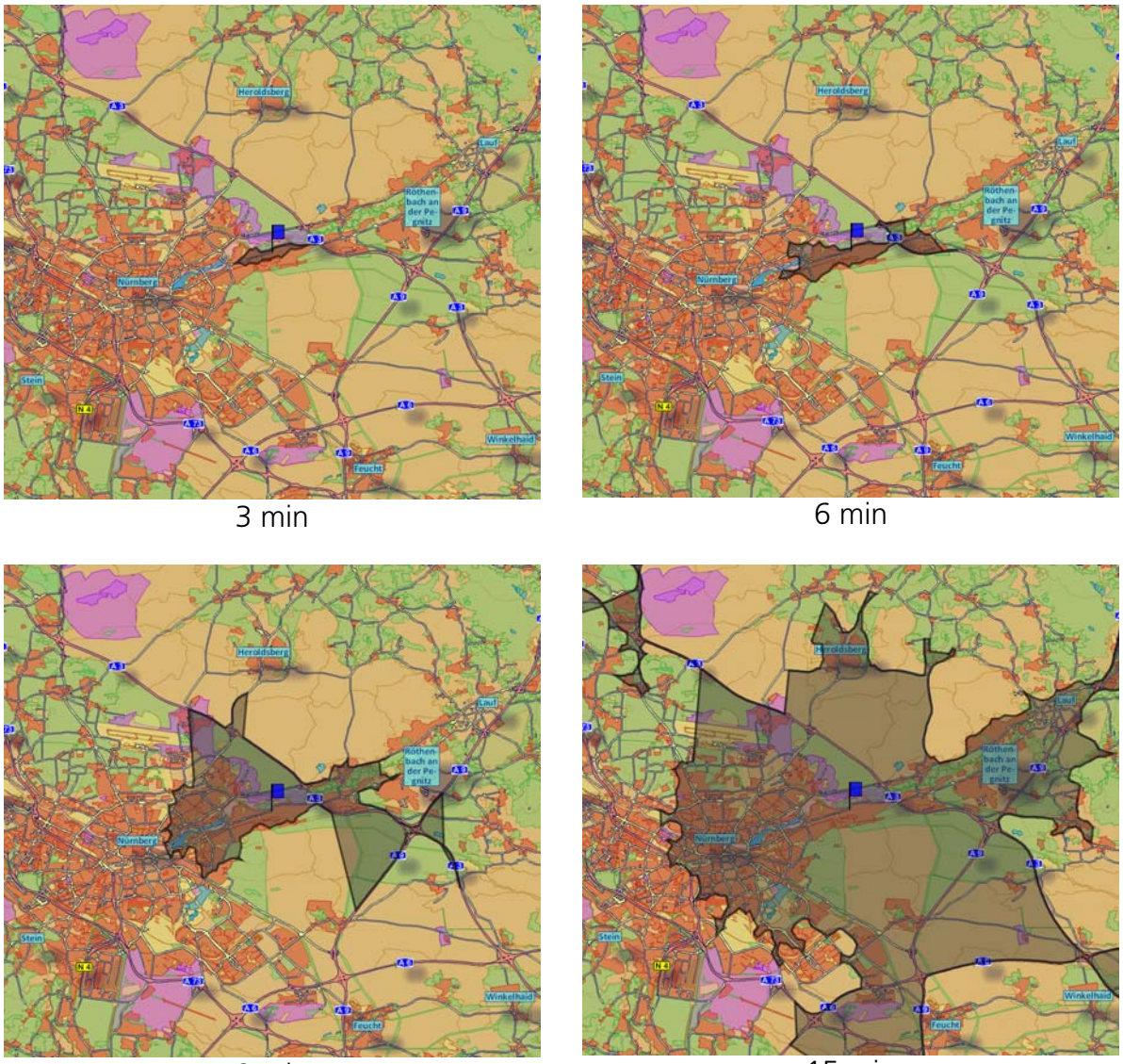
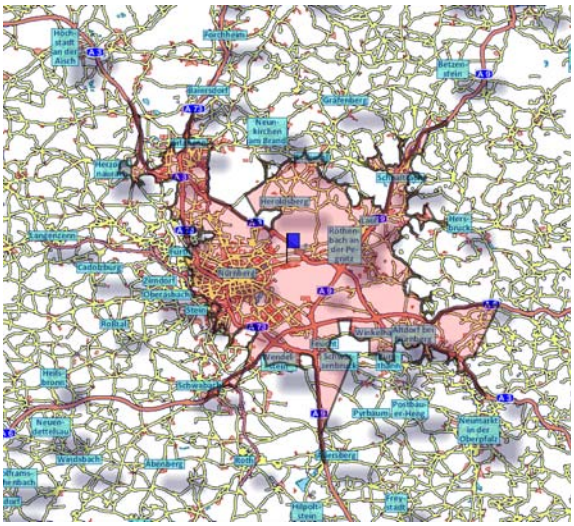
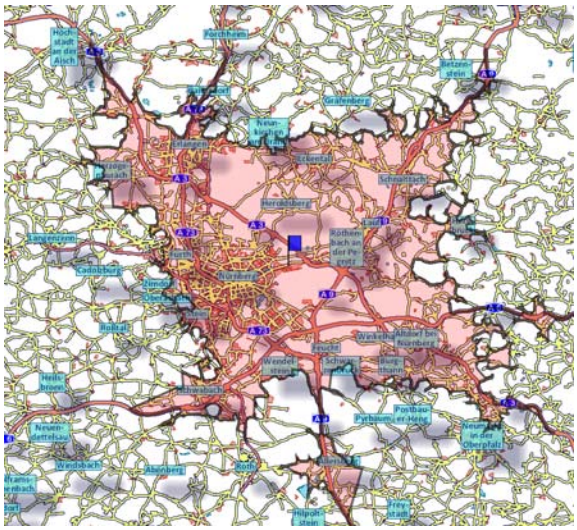


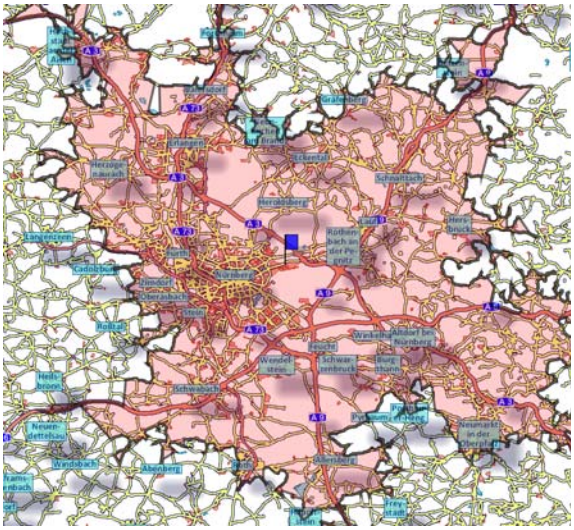
Abb. 3: Isochrone für eine Autofahrt von 3-15 Minuten



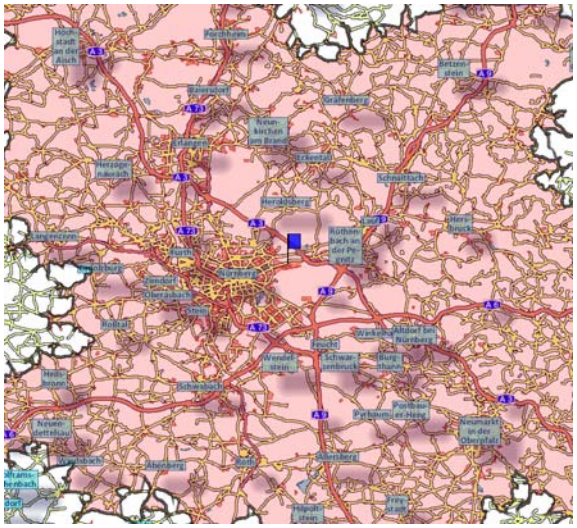
20 min



25 min



30 min



40 min

Abb. 4: Isochrone für eine Autofahrt von 20-40 Minuten

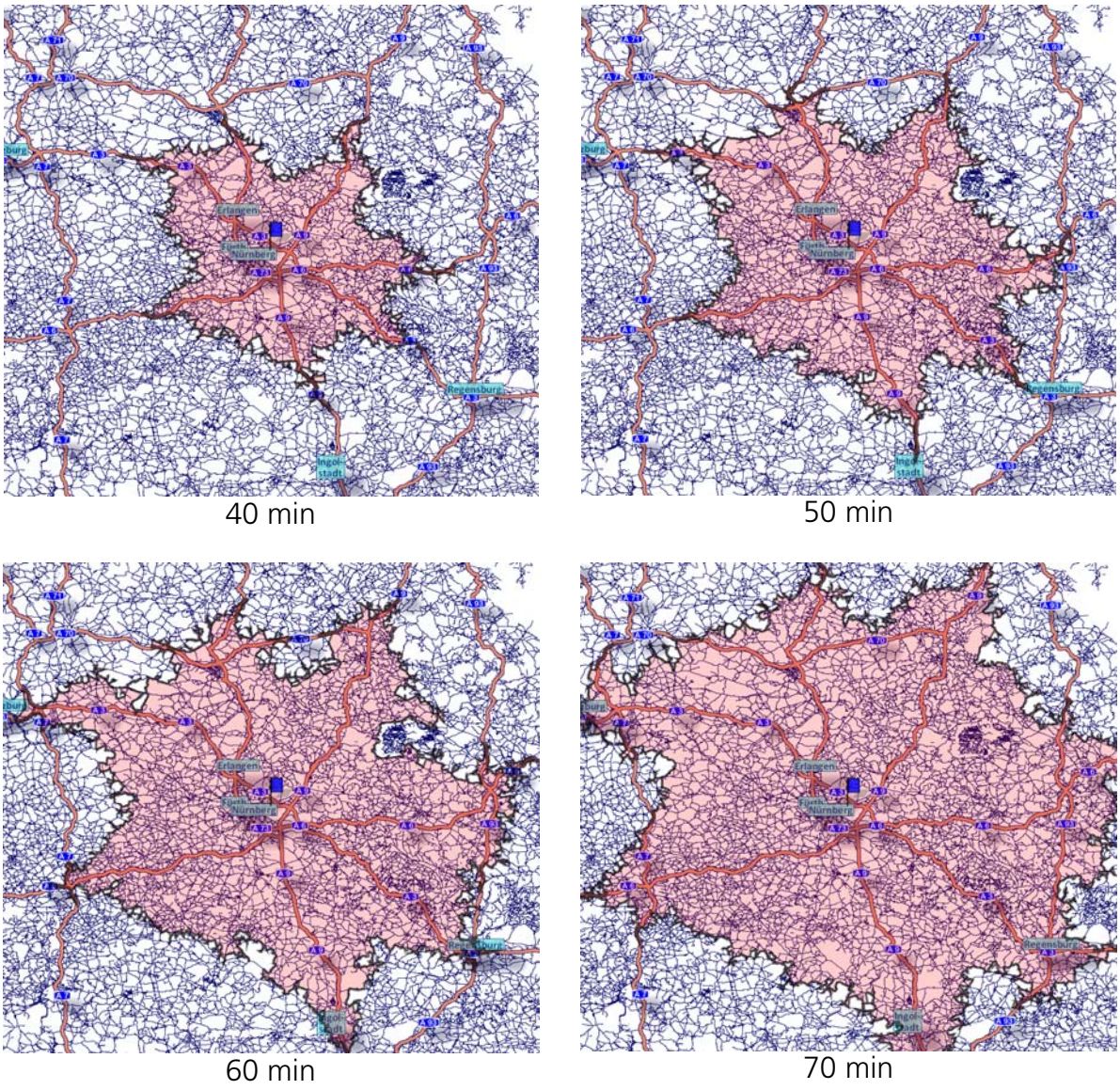


Abb. 5: Isochrone für eine Autofahrt von 40-70 Minuten

Einige Details:

- Das liegt Straßennetz in Form von Links und Crossings vor (siehe oben). Die Links haben einen Straßentyp (z.B. Autobahn) und eine Länge in Metern. Sie müssen selbst eine sinnvolle Abbildung von einem konkreten Link auf die Fahrzeit entwickeln. Diese Abbildung muss neben dem Straßentyp auch eventuelle Tempo-Limits und Einbahnstraßen-Richtungen berücksichtigen.
- Der Start-Punkt liegt immer auf einer Crossing, also nicht auf einer Teilstrecke *zwischen* zwei Kreuzungspunkten oder gar außerhalb des Wegenetzes. Da die Koordinaten-Angabe durch den Aufrufer allerdings im Rahmen der Darstellungsgenauigkeit nicht exakt einen Kreuzungspunkt treffen kann, muss das Programm den am nächsten liegende Kreuzungspunkt (bis zu einer maximalen Distanz) auswählen.
  Bemerkung: sollte die Gruppe ausnahmeweise aus drei Gruppenmitgliedern bestehen, wird diese Vereinfachung außer Kraft gesetzt. In diesem Fall kann der Startpunkt auf beliebiger Position auf einem Link liegen.

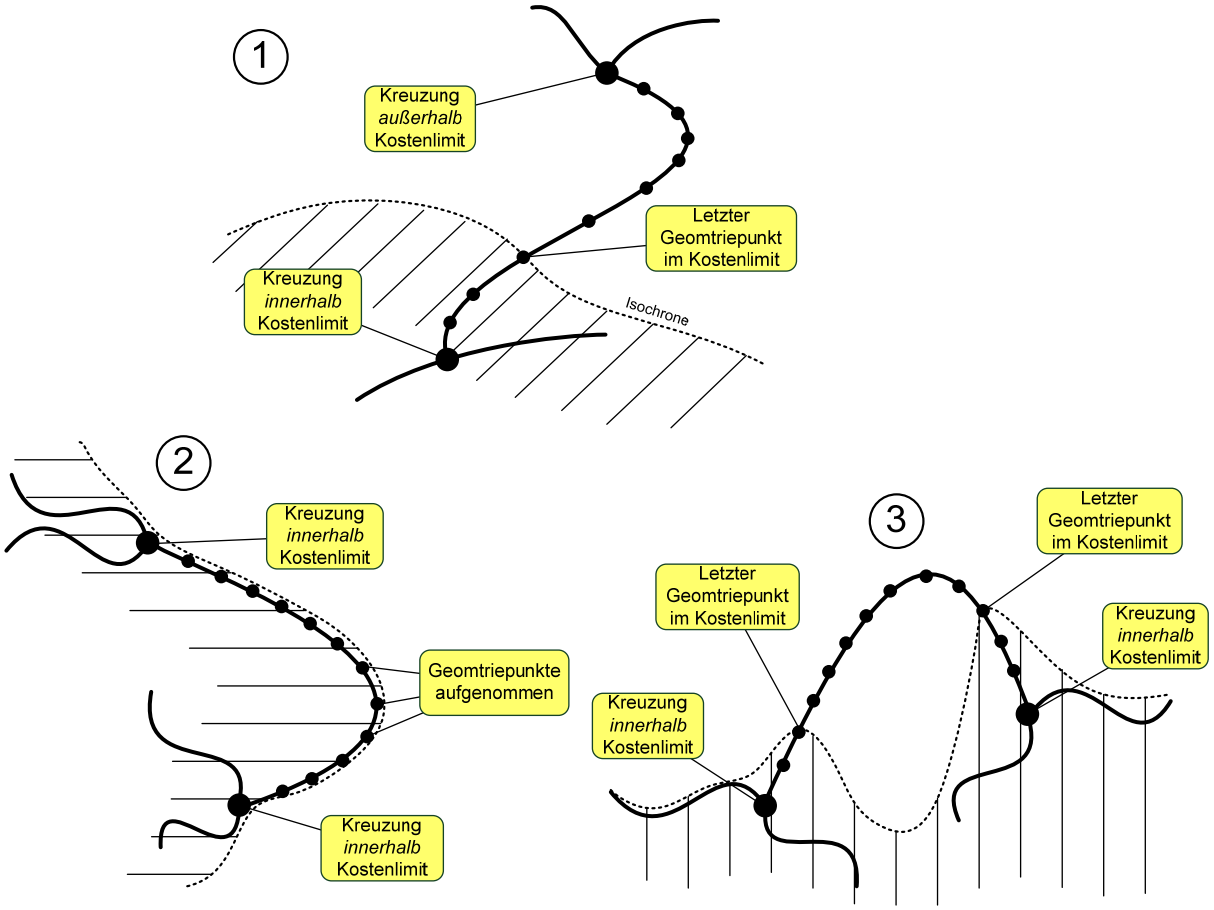


Abb. 6: Sonderfälle bei der Isochrone-Berechnung

- Bei der Berechnung der erreichbaren Punkte sind u.U. auch Geometriepunkte innerhalb eines Links zu beachten (Abb. 6). Fall 1: ein Link beginnt innerhalb der Isochrone-Fläche und endet außerhalb. Dann muss der letzte Geometriepunkt identifiziert und berücksichtigt werden, der noch innerhalb der Fläche liegt. Fall 2: ab einer bestimmten (selbst zu definierenden) Link-Länge werden alle Geometriepunkte bei der Berechnung der Isochrone berücksichtigt. Fall 3: ein Link beginnt von beiden Seiten innerhalb der Isochrone, die Geometriepunkte liegen aber teilweise außerhalb. Hier müssen die zwei Geometriepunkte identifiziert und berücksichtigt werden, die noch innerhalb der Fläche liegen.
- Der Ansatz, um alle erreichbaren Kreuzungspunkte innerhalb der vorgegeben Fahrzeit zu ermitteln, wird hier nicht vorgegeben. Er wird während des FWPFs besprochen und entwickelt.
- Hat man die Menge aller erreichbaren Kreuzungspunkte berechnet, muss schließlich die Grenzlinie ermittelt werden. Der hierfür notwendige Ansatz wird *konkave Hülle* genannt. Es muss kein Algorithmus hierzu entwickelt werden, es liegt schon eine Bibliothek dazu vor (**`fu.util.ConcaveHullGenerator`**).

Kartenausgaben mit dorenda

Während der Entwicklungsphase wird es notwendig sein, Daten auf einer Karte darstellen zu können. Auch die Ausgabe des fertigen Programms soll auf der Karte erfolgen. Hierzu steht das Werkzeug *dorenda* zur Verfügung (Abb. 7).

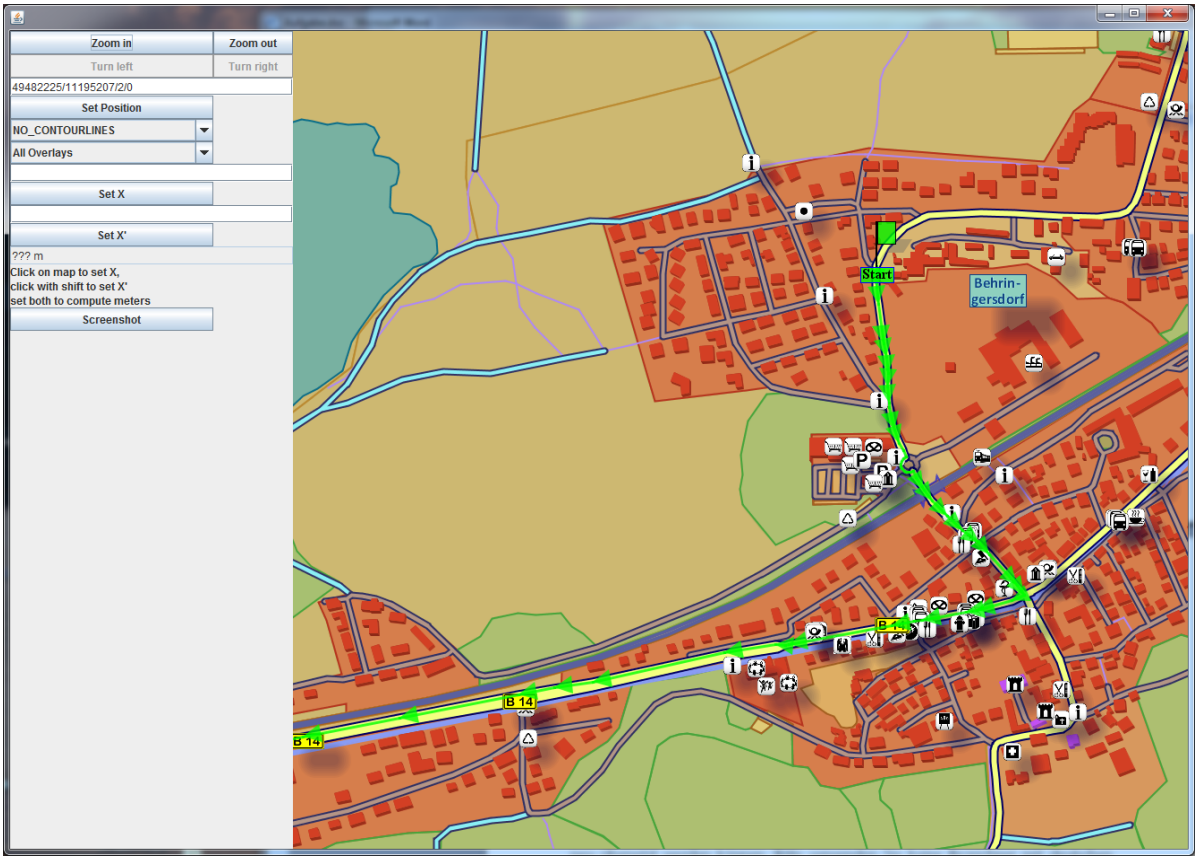


Abb. 7: Das dorenda Kartenwerkzeug

Mit dorenda kann man

- eine Karten betrachten,
- Koordinaten von Punkten ermitteln oder Koordinaten auf der Karte markieren,
- eigene Daten aus Linienzügen, Punkten oder Flächen auf der Karte anzeigen.

Die Anzeige von eigenen Daten erfolgt über eine Textdatei, die dorenda beim Start mitgegeben wird. Diese Textdatei enthält dann Zeichenkommandos – deren Ausführung führen zu Zeichnungen (z.B. Linienzüge) auf der eigentlichen Karte. Die Syntax solch der Textdatei müssen Sie nicht selbst beherrschen. Der komfortable Weg geht über die Klasse **`pp.dorenda.client2.additional.UniversalPainterWriter`**. Als Beispiel: der Aufruf

```
line(testLats,testLongs,
     0,255,0,200,4,3,
     "Start","...Route...","End");
```

mit entsprechenden Arrays **`testLats`**, **`testLongs`** führt zu der Ausgabe in Abb. 7.

Kommandozeilenparameter der Anwendung

Die Anwendung erhält den Auftrag ausschließlich über Kommandozeilenparameter. Es gibt weder eine graphische Benutzungsschnittstelle noch Konfigurationsdateien. Die Anwendung muss die folgenden Kommandozeilenparameter einlesen (bei exakter Reihenfolge und Anzahl):

<dbaccesstr> <cacfile> <latitude> <longitude> <minutes> <fromLSI> <toLSI>

mit:

- <dbaccesstr>: Zugriff-String für die Datenbank in der Form <dbhost>/<dbport>/<dbuser>/<dbpasswort>/<dbname>

  also z.B.

  `geosrv.informatik.fh-nuernberg.de/5432/dbuser/dbuser/deproDB20`

- <cacfile>: Dateiname des Straßennetzes, also z.B. **`CAR_CACHE_de_noCC.CAC`**

- <latitude> <longitude> <minutes>: Startpunkt in Grad, maximale Fahrzeit in Minuten, z.B.: **`49.46591000 11.15800500 15`**

- <fromLSI> <toLSI>: Intervall der Objekttypen von Interesse, z.B. für Tankstellen: **`20505600 20505699`**

Es darf keine weiteren Parameter geben auch keine Optionsparameter (z.B. **`-?`**, **`-output`**).
Die Ausgabe-Datei (also die generierte Kartenausgabe) muss den Namen **`result.txt`** haben. Dieser Name ist fest definiert und darf auch nicht über einen Parameter definiert werden.

Was ist abzuliefern

Damit das Überprüfen der Abgaben vereinfacht wird, sollen an dieser Stelle eine Reihe von Festlegungen gemacht werden. Diese müssen (inkl. Groß/Kleinschreibung) exakt eingehalten werden.

- Jeder Gruppe wird ein eindeutiger Package-Name mitgeteilt. Dieser wird in der Regel aus den beiden Anfangsbuchstaben der Nachnamen der Gruppen-Beteiligten (kleingeschrieben) gebildet. Für eine Gruppe bestehend aus den Mitgliedern *Piccard* und *Riker* müssen sich also beispielsweise alle Klassen in dem Package **`piri`** befinden.
- Die Hauptklasse der Anwendung liegt auf oberster Ebene im Package und hat den Klassennamen **`Isochrone`**. Weitere Klassen im Package oder in Subpackages sind natürlich erlaubt. Hier ist der Klassenname dann beliebig.
- Alle entwickelten Klassen müssen als eine Java-Bibliothek in Form einer **`.jar`**-Datei abgegeben werden. Der Dateiname entspricht dem Package-Namen, also z.B. **`piri.jar`**. Das Package enthält alle übersetzten Klassen *und* die java-Quellen (also sowohl **`.class`**- als auch **`.java`**-Dateien). Beide Dateitypen liegen in denselben Verzeichnissen, d.h. jede **`.java`**-Datei liegt im selben Verzeichnis wie die zugehörige **`.class`**-Datei. Insb. liegen die **`.java`**-Dateien *nicht* (!) in einem separaten Verzeichniszweig z.B. **`src/`**.
- Die Anwendung muss mit der Kommandozeile

  ```
  java -cp geo.jar;piri.jar piri.Isochrone
  geosrv.informatik.fh-nuernberg.de/5432/dbuser/dbuser/deproDB20
  CAR_CACHE_de_noCC.CAC 49.46591000 11.15800500 15
  20505600 20505699
  ```

  gestartet werden können, wenn das Package wie im Beispiel oben benannt wurde. Das bitte *vor* der Abgabe testen.

- Die Anzeigedatei des Ergebnisses **`result.txt`** muss direkt über den dorenda-Aufruf

```
java -cp .;geo.jar pp.dorenda.client2.testapp.TestActivity
-m webservice;geosrv.informatik.fh-nuernberg.de
-c pp.dorenda.client2.additional.UniversalPainter
-a result.txt;s
```

angezeigt werden können.

Bitte verwenden Sie keine Klassennamen, Bezeichner oder Kommentare mit nationalen Sonderzeichen, da diese bei der Übersetzung auf dem Testrechner zu Problemen führen können. Die Quellen sollen mit dem JDK Standard Edition 8 übersetzbar und ausführbar sein.

Bitte schicken Sie zur Abgabe die **`.jar`**-Datei per Email an mich.

Es ist keine Zusammenarbeit zwischen den Gruppen erlaubt. Jede Gruppe muss komplett eine eigene Lösung entwickeln. Insbesondere ist die Weitergabe von Quellen zwischen den Gruppen nicht erlaubt, auch nicht "zur Ansicht" oder "nur zur Information".

Die Abgabe ist fällig am **30.06.2017** (harter Termin).
Bitte per Email einsenden an **`Joerg.Roth@th-nuernberg.de`**.