Universität Dortmund, Fachbereich Informatik

GPS-Route

Endbericht

Projektgruppe 452

Heiner Ackermann
René Brüntrup
Vanessa Faber
Thomas Härtel
Miguel Liebe
Björn Scholz
Mohamed Bettahi
Maik Drozdzynski
Andreas Gaubatz
Seung-Jun Hong
Anne Scheidler
Feng Wang

Stefan Edelkamp Shahid Jabbar Tilman Mehler

Vorwort

- Stefan Edelkamp -

In der Projektgruppe 452 *GPS-Route* wurde am Fachbereich Informatik der Universität Dortmund ein Navigationssystem konzipiert und implementiert, das aus einer Menge von GPS-Aufnahmesequenzen eine mit Fahrtzeiten und Fahrtmöglichkeiten annotierte Karte generiert, diese vorverarbeitet, speichert und darin möglichst effizient nach kürzesten Wegen sucht. Die GPS-Daten werden sowohl mit einem GPS Empfänger als auch in einer zu erstellenden Simulationsumgebung erzeugt. Dieser Endbericht der Projektgruppe 452 dokumentiert den Stand des Projekts nach Abschluss der beiden Semester.

Der Vorteil einer an der Universität angesiedelten Entwicklung ist der gezielter Einsatz von aktuellen Algorithmen und Datenstrukturen und die Verwirklichung eines komplexen, längerfristigen und aufteilbaren Projektzieles. Das Thema der On-Line Navigation basierend auf GPS Traces beinhaltete auf mehreren Ebenen viele algorithmische Fragestellungen und war demnach für ein Informatik Hauptstudiumsprojekt (12 Personen, ein Jahr Entwicklungszeit) besonders gut geeignet.

Dieser Endbericht belegt die Leistungen der einzelnen Untergruppen und gliedert diese in das Gesamtkonzept ein. Es sind insgesamt vier voneinander unabhängige Systeme entstanden, die entweder über eine Internetverbindung oder über einen direkten Dateitransfer miteinander kommunizieren.

Aufgabenstellung

Der Ansatz zur Routenplanung mit dynamisch erzeugten Spurdaten eignet sich besser zur Navigation als ein statischer Graph des Wegeplanes. Aktuelle Informationen können mit Hilfe von GPS-Aufnahmegeräten bereitgestellt werden. Dabei wird der personelle und technische Aufwand zur elektronischen Kartographierung minimiert. Zudem sind kommerziell vertriebene elektronische Karten recht ungenau und die gespeicherte Information für ein Navigationssystem durch stete Realwelt-Änderungen oft veraltet.

In dem Projekt sollte ein Client-Server Routenplanungssystem entstehen, das, basierend auf eine GPS-Datenbank – ähnlich dem Bahninformationsystem HAFAS –

Auskunft über kürzeste und schnellste Verbindungswege von A nach B zum Zeitpunkt t gibt. Wegeanfragen und Einspeicherungen von aufgenommenen GPS Spuren sollten über das Internet, bzw. dem TCP/IP Protokoll, ermöglicht werden.

Um die aktive Aufnahme großer Datenmengen von GPS Spuren zu umgehen, sollte in der PG desweiteren eine Verkehrssimulationsumgebung entwickelt werden, in denen GPS Navigationsdaten erzeugt werden können. Diese Daten sollten dann zur Evaluation der Kartengenerierung und der Suchverfahren genutzt werden. Die Vektorisierung existierender topographischer Karten bietet sich dabei als Zwischenschritt an. Solche Karten, z.B. aus den der Landesvermessungsämtern, sind sehr genau vermessen und lassen sich mit GPS Informationen verknüpfen. Die Anfragen an das entstehende System und die Darstellung berechneter Routen sollte auf einem mobilen Endgerät (PDA) ermöglicht werden.

Vorgabe

Im Schloss Dagstuhl wurde vom 7. bis 8. April 2004 ein Einstiegsseminar veranstaltet, u.a. zu den Themen *geometrische Algorithmen* und *geometrisches Runden*, aktuelle *GPS-Technologie* und *Kartographie*, sowie *Integration inertialer Information* aufgrund zusätzlicher Messdaten (*Kalman* Filterung), *Datenkompression* und *statistische Analyse*, effiziente *kürzeste-Wege Suchverfahren*, die *LEDA* Algorithmenbibliothek, das bestehende System *GPS-Route* und Recherchen über existierende Systeme der GPS Navigation. Die Ausarbeitungen zu den Themen formten zusammen mit ersten Implementierungserfolgen den Zwischenbericht.

Als technisches Inventar wurde ein portabler Handheld-Computer (T-Mobile MDA I) mit Internetandbindung und GPS-Empfänger, eine Telefonkarte und ein 512 MByte SD Speicherchip zur Verfügung gestellt. Desweiteren konnte ein weiterer GPS Empfänger der Firma GARMIN in Verbindung mit einem Laptop zur Aufzeichnung genutzt werden. Bei der Kartenvektorisierung und Simulation sind wir von dem DTK-Kartensatz von Dortmund des Landesvermessungsamts NRW (Bonn) ausgegangen. Es wurden ca. 400 *Kacheln* der Größe 1 km^2 im TIFF Format zum Stückpreis von ca. 1 Euro pro Kachel eingekauft. Für die Implementierung haben wir uns auf die Sprache `C/C++` geeinigt.

Ein Prototyp eines spurbasierten Routenplanungssystem wurde im Rahmen der Diplomarbeit von dem Mitbetreuer Shahid Jabbar entwickelt und der Projektgruppe zur Verfügung gestellt. Er ermöglicht das Einlesen von GPS-Spuren, die Berechnung des Schnittpunktgraphens, grundlegende Graphkompressionsverfahren, Punktlokalisation und kürzeste-Wege-Suche. Ansätze zur Effizienzsteigerung wurden entwickelt und die dynamischen Veränderung der Karten aufgrund von einlaufender Information untersucht. Als Infrastruktur wurde das proprietäre System auf den PG-Rechnern des Fachbereichs Informatik an der Universität Dortmund installiert. Desweiteren wurde die Algorithmenbiblithek LEDA zur Verfügung gestellt.

Aufteilung

Bei der Aufteilung in Projektmodule haben wir vier Kleingruppen gebildet, die bis zum Ende des Projektes bestehen blieben.

Gruppe „Simulation“ Die Aufgabe in dieser Teilgruppe war eine geeignete *Rasterkartenvektorisierung* und die Verwirklichung einer *Verkehrsimulationumgebung* zur automatischen Erzeugung von GPS-Spuren. Zur Vektorisierung der Karten, d.h. zur Straßenextraktion und Graphgenerierung, wurden verschiedene graphische Skelettierungs- und Trackingalgorithmen benötigt. Auf diesen Graphen wird dann eine Verkehrssimulation durchführen, die für jedes an der Simulation beteiligte Fahrzeug eine GPS-Spur liefert, sozusagen als Protokoll des Verkehrsflusses und des Fahrzeugverhaltens. Dazu wurde eine existierende Simulationsumgebung (SUMO) sowohl auf der Eingabe- als auch auf der Ausgabeseite deutlich erweitert.

Gruppe „Map Generation“ Die Aufgabe dieser Teilgruppe war die *dynamische Kartengenerierung* auf Basis von vorhandenen GPS-Spurdateien. Auf einer initial leere Weltkarte werden nach und nach die erzeugten Spuren zu einer Karte verschmolzen. Hierbei wurde ein parametrisierter maschineller Lernansatz genutzt, der die verteilten Kartenelemente in der Datenbank verteilt aktualisiert. Mit Fehlern behafteten GPS-Spuren werden gefiltert.

Gruppe „Routing“ Die Aufgabe war es, effiziente Datenstrukturen und Algorithmen zur *Punktlokalisation* und zur *Kürzeste-Wege Berechnung* auf dem Server, basierend auf den erstellten Karten, zu realisieren. In diesem Kernmodul wird ein möglichst effizientes Routing auf einem komprimierten, mit Zeitfenstern annotierten Graphen realisiert. Die Karten entsprechen Graphen, die eingelesen, verarbeitet und dargestellt werden. Anfragen an den Routing-Server werden durch ein einfaches Protokoll realisiert.

Gruppe „PDA“ Die Aufgabe war die Erstellung eines *PDA*-Programms zur effizienten *Karten- bzw. Routendartellung*, zur *GPS Spuraufnahme* und zur Verarbeitung von *Start/Ziel Anfragen*. Die Karten können in verschiedenen Skalierungsstufen betrachtet werden und ein leichtes Navigieren für Fußgänger, Radfahrer und Autofahrer ermöglichen. Insbesondere wird der aktuelle Standpunkt zur Kalibrierung der Darstellung genutzt und eine Sprachausgabe der Routinginformation realisiert. Eine Schnittstelle kleiner Bandbreite zum Routenplanungsserver musste ebenso gestaltet werden, wie die Kommunikation der GPS-Spur zum Server der Kartengenerierung.

Das Zusammenspiel der Gruppen und damit die Architektur des Gesamtsystems ist wie folgt. Die PDA-Gruppe fragt mittels einer telefonisch aufgebauten Internetverbindung nach einer kürzesten oder schnellsten Route, basierend auf einer Eingabe des

Benutzers auf dem PDA. Sie bekommt einen annotierten Pfad von der Routenplaungsgruppe zurück, der auf dem Endgerät auf dem existierenden Kartensatz dargestellt werden kann. Zur besseren Navigation wird die Karte bezogen auf dem aktuellen Standort kalibriert und der Benutzer auf der erfragten Route sprachlich geleitet. Desweiteren lassen sich mit dem Endgerät aufgenommene GPS-Spuren an die Kartengenerierungsgruppe zur Verarbeitung mittels der Telefonverbindung weiterleiten. Die Routenplanungsgruppe wiederum erhält den der Navigation zugrundeliegenden Graphen als Extrakt aus der Kartengenerierung. Die Anfrage nach einem neuen Routinggraphen werden über durch einen gewöhnlichen Transfer von (mitunter komprimierten) ASCII-Dateien realisiert. Aus praktischen Erwägungen heraus, wird das Kartengenerierungsmodul nicht nur von Eingaben aus der PDA-Gruppe gespeist, sondern auch von der Simulationsgruppe. Die dort erzeugten GPS-Spurdatenpunkte werden entweder über eine Internetanbindung oder über einen gewöhnlichen ASCII Dateitransfer entgegengenommen.

Leistungsnachweis und Ausblick

Die Projektgruppe hat ihr Mindestziel mehr als erfüllt. Die Grundbausteine des Systems wurden um eine reichhaltige Funktionalität erweitert. Die Implementierungen sind stabil und auch in größeren Spurdatenmengen für den Praxiseinsatz geeignet. Die einzelnen Komponenten wurden mehrfach getestet und arbeiten problemlos zusammen.

Prinzipiell sind Simulation, Routing und Kartengenerierung *universell*, d.h. für beliebige Rasterkarten bzw. unkartographierte Regionen der Welt anwendbar. Um jedoch wettbewerbstauglich zu sein, benötigt man viel mehr Kartenmaterial und/oder GPS Daten. Wir sind mit einem vergleichbar kleinen Datensatz von Dortmund und seiner Umgebung bzw. mit einigen wenigen aufgezeichneten GPS Spuren gestartet. Für eine direkte über einen Stadtführer herausreichende Anwendung fehlen uns die personellen und finanziellen Mittel. Ein Ziel mit Abschuss dieser Arbeit ist es demnach, die *praktische Akzeptanz* zu erhöhen. Wir bemühen uns um eine permanente Plattform, auf der unsere Routing- und Kartendienste allgemein zur Verfügung gestellt werden. Desweiteren sollen mit den erzielten Fortschritten Kontakte zu Firmen aufgebaut werden.

Unser *universitäres Interesse* gilt auch weiterhin der Entwicklung, Ausarbeitung und Verwirklichung von von effizienten, insbesonderer geometrischen, Algorithmen. Hieraus entspringen einige, aber längst nicht alle, möglichen Erweiterungsmöglichkeiten und Zielsetzungen, die auch als Anregung für eine Diplomarbeit dienen können. So soll die Einbindung dynamischer Routeninformation z.B. über nicht (mehr) passierbares Gelände und stockenden Verkehrsfluss verbessert werden. Desweitern sind die Integration weiterer Messdaten (Tachometer), die Implementation von Externspeicherplatzverfahren, sowie die Zusammenfassung und Bereinigung von GPS-Daten von zentralem Interesse.

Inhaltsverzeichnis

Kapitel 1

Technische Grundlagen

1.1 Globale Positionsbestimmung

Dieses Kapitel beschäftigt sich mit dem GPS-System. Es wird erklärt, wie es aufgebaut ist und wie die Positions- bzw. Zeitbestimmung abläuft. Es ist angelehnt an [1].

1.1.1 Mathematischer Hintergrund

Es soll erklärt werden, wie man seine Position mit Hilfe von Referenzzeiten die mittels elektromagnetischen Wellen (Radiowellen) ausgesendet werden, bestimmen kann. Die Erklärung erfolgt *dimensionsweise*, d. h. es wird mit einem eindimensionalen Fall begonnen und schrittweise bis zur dritten Dimension fortgeführt. Unterschieden wird anfänglich nach synchronen und asynchronen Uhren.

1D - mit synchronen Uhren Betrachtet man den eindimensionalen Fall so gibt es eine bekannte Referenzstation und Empfänger auf einer Geraden. Hier sollen die Uhren von Sender und Empfänger synchron laufen. Macht man weiter die Annahme, dass sich die Empfänger nur in eine Richtung vom Sender befinden, kann man die Position eindeutig bestimmen nach folgender Gleichung:

$$x = v \cdot (t - t_1) \tag{1.1}$$

Wobei gilt:

- x errechneter Ort
- $v = c$ Lichtgeschwindigkeit
- t Empfangszeit
- t_1 Sendezeit

1D - mit asynchronen Uhren Hier gelte nun, dass die Uhren des Senders und der Empfänger nicht mehr synchron laufen. Damit lässt sich Gleichung (1.1) nicht ohne weiteres anwenden. Gelöst wird dieses Problem indem man einen zweiten Sender mit bekannter Position aufstellt. Die Uhren der Sender *müssen* synchron laufen. Nun gelten folgende Gleichungen:

$$\begin{aligned} x_1 &= v \cdot (t - t_1) \\ x_2 &= v \cdot (t - t_2) \\ E &= x_1 + x_2 \end{aligned}$$

Somit liegt ein eindeutig lösbares Gleichungssystem vor. Anschaulich machen kann man sich den Sachverhalt an Abbildung 1.1.

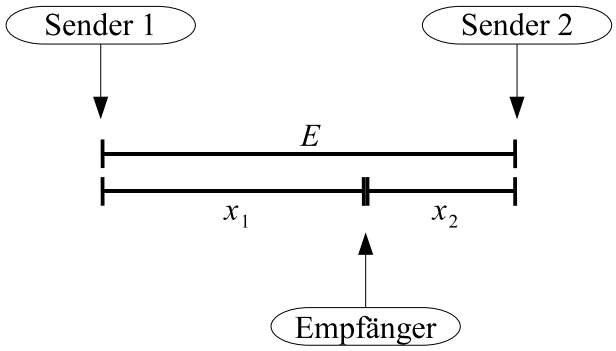


Abbildung 1.1: Beispiel für eindimensionalen Fall mit asynchronen Uhren

2D - mit asynchronen Uhren Im folgenden wird der zweidimensionale Fall betrachtet, wobei die Uhr des Empfängers nicht mit den Uhren der Sender synchron läuft. Auch hier *müssen* die Uhren der Sender zueinander synchronisiert sein. Geht man vom oben beschriebenen System aus und übernimmt dieses ohne Änderungen in den zweidimensionalen Fall, so ergeben sich Mehrdeutigkeiten in der Positionsbestimmung.

Da Radiowellen sich radial um den Sender ausbreiten, ergeben sich bei zwei gemessenen Zeiten auch zwei mögliche Positionen: Kreise schneiden sich nunmal in zwei Punkten (zumindest in dieser Konfiguration; den unwahrscheinlichen Fall, dass sich der Empfänger exakt mittig zwischen den Sendern befindet, ausgenommen).

Die Lösung ist ein dritter Sender mit den *üblichen* Eigenschaften (bekannte Position & Uhr synchronisiert mit den anderen Sendern). Anschaulich ergeben sich nun drei Kreise, die sich ein genau einem Punkt schneiden: der Position des Empfängers.

Die mathematischen Gleichungen hierzu erhält man aus der Kreisglei-

chung $x^2 + y^2 = r^2$:

$$r_1 = v \cdot (t - t_1) = \sqrt{(x - x_1)^2 + (y - y_1)^2}$$
$$r_2 = v \cdot (t - t_2) = \sqrt{(x - x_2)^2 + (y - y_2)^2}$$
$$r_3 = v \cdot (t - t_3) = \sqrt{(x - x_3)^2 + (y - y_3)^2}$$

Dieses nichtlineare Gleichungssystem lässt sich iterativ hinreichend genau lösen.

3D - mit asynchronen Uhren Im dreidimensionalen Fall funktioniert die Positionsbestimmung analog zum oben beschriebenen zweidimensional Fall. Hierbei benötigen man vier Sender. Die Postition wird über die Kugelgleichung $x^2 + y^2 + z^2 = r^2$ berechnet.

1.1.2 GPS - Global Positioning System

Die Informationen dieser Sektion sind [19] und [24] entnommen worden.

Aufbau Das GPS-System wird in drei Segmente eingeteilt: Weltraum-, Kontroll- und Benutzersegment.

Das *Weltraumsegment* besteht aus mindestens 24 Satelliten, die verteilt auf sechs gleichmäßig verteilten Bahnen die Erde in einer Höhe von ca. 20200 km umkreisen. Jede Bahn ist um 55° gegenüber der Äquatorebene inkliniert, d.h. geneigt. Damit wird garantiert, das zu jedem Zeitpunkt an jedem Ort der Erde mindestens vier Satelliten sichtbar sind.

Jeder Satellit hat mindestens drei Atomuhren an Bord, die jeweils mindestens eine Ganggenauigkeit von 10^{-13} s haben. Diese sehr genauen Uhren werden für die Positionsbestimmung gebraucht, da bereits Abweichungen im Nanosekundenbereich das Ergebnis entscheidend verfälschen.

Das *Kontrollsegment* besteht aus fünf Überwachungsstationen (Monitor Stations), wobei die Hauptkontrollstation (Master Control Monitor Station; Sitz ist Schriever AFB, Colorado) die Daten verarbeitet und wenn nötig Korrekturdaten an die Satelliten schickt. Hierbei kann es sich zum Beispiel um Bahnkorrekturdaten handeln. Dieses Kontrollsegment liegt komplett in der Hand des US-Militärs.

Das*Benutzersegment* besteht aus den passiven GPS-Empfängern. Hier lässt sich eine Aufteilung in militärische, zivile und Spezialempfänger machen. Sie unterscheiden sich in den Frequenzen, die sie zur Positionsbestimmung nutzen (können) und in ihrer Genauigkeit.

Signale Die Satelliten senden die für die Positionsbestimmung nötigen Informationen auf zwei Frequenzen: 1575,42 MHz (L1) und 1227,60 MHz (L2). Diese Frequenzen liegen im L-Band (1 - 2 GHz) und sind für Zweck des GPS hinreichend gut.

Hier spielen Faktoren wie Ausbreitungsgeschwindigkeit, Brechung in einzelnen Atmosphärschichten, Beeinflussung durch das Wetter und weitere eine Rolle.

Die Frequenz L1 wird mit dem binären C/A-Code[1] und dem ebenfalls binären P-Code[2] moduliert. Das GPS-System verwendet hierfür die Phasenmodulation. L2 wird nur mit dem P-Code moduliert. Da der P-Code verschlüsselt wird und somit nur mit entsprechenden (militärischen) Empfängern nutzbar ist, wird im weiteren nicht mehr detailiert darauf eingegangen.

Der C/A-Code ist ein PRN[3]-Code mit einer Länge von 1023 chips und wird mit 1,023 MHz aufmoduliert. Für jeden Satelliten existiert ein eindeutiger Code.

Die Navigationsdaten werden in einem ebenfalls binären 50 Hz-Signal übermittelt. Dieses Signal wird auf den C/A-Code *aufmoduliert*, und zwar durch eine einfache XOR-Verknüpfung (welche äquivalent zu einer modulo-2 Addition ist). Veranschaulicht wird dieses in Abbildung 1.2.

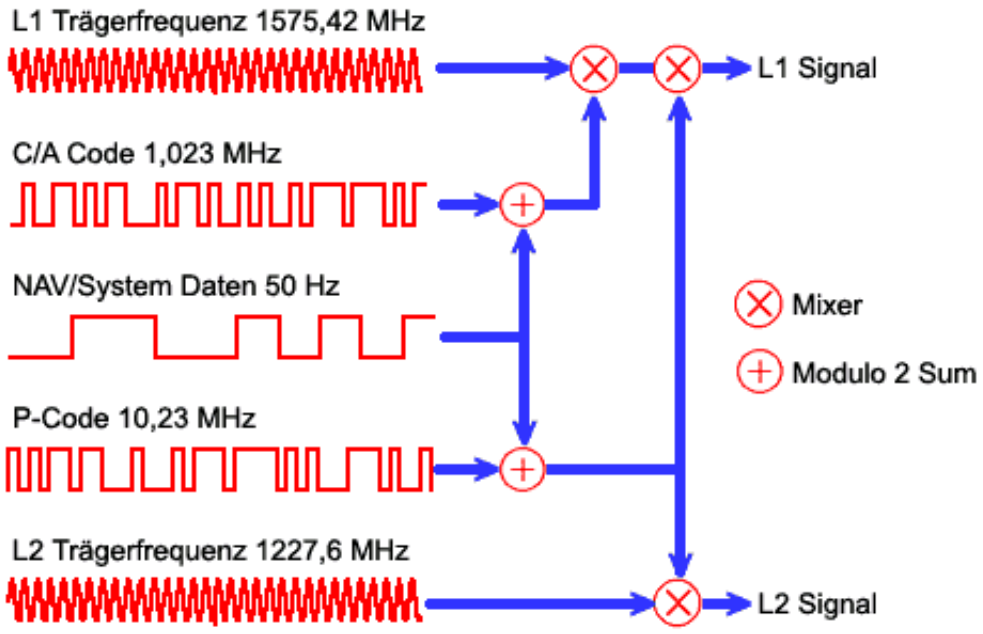


Abbildung 1.2: Zusammensetzung der Signale (nach Peter H. Dana [24])

Die Navigationsdaten sind in Rahmen (Frames) aufgeteilt. Jeder Rahmen ist 1500 bits lang, somit dauert die Übertragung 30 s. In einem Rahmen sind Zeitstempel, die Bahndaten des aktuell empfangenen Satelliten (Ephemeridendaten) sowie Teile der ungefähren Bahndaten aller anderen Satelliten (Almanachdaten) enthalten.

[1] Coarse Aquisition = grobe Bestimmung
[2] Precise = genau
[3] Pseudo Random Noise = pseudozufälliges Rauschen

Positionsbestimmung Für die Positionsbestimmung wird eine Technik, die Kreuzkorrelation, genutzt, die nun im weiteren erklärt wird: Ein GPS-Empfänger generiert die gleichen PRN-Sequenzen der Satelliten. Dann wird das generierte Signal mit dem empfangenen Signal chipweise multipliziert. Die daraus resultierende Sequenz wird chipweise aufaddiert. Die erhaltene Zahl gibt die Anzahl der Übereinstimmungen. Ein Beispiel zeigt Abbildung 1.3. Hierbei ist das oberste Signal das vom Satelliten empfangene. Der Beginn ist mit dem grünen Block gekennzeichnet. Das zweite Signal ist das vom Empfänger generiert. Und in der letzten Zeile ist das Ergebnis der Kreuzkorellation dargestellt.

Abbildung 1.3: Kreuzkorrelation

Wenn diese durch Verschiebung des generierten Signals maximal wird, liegt eine exakte Übereinstimmung vor. Nun kann man die Zeit, die das Signal vom Satelliten bis zum Empfänger gebraucht hat, ablesen.

Wie oben beschrieben, funktioniert die Positionsbestimmung in der Praxis erstmal nicht, da die Uhr des Empfängers nicht synchron zu den Uhren in den Satelliten läuft. Mit der Kreuzkorrelation kann man also nur die relative Signallaufzeit zur Empfängeruhr bestimmen. Hat man jedoch den Start einer PRN-Sequenz ermittelt, können die Navigationsdaten ausgelesen werden. Sind hinreichend viele Satelliten sichtbar, kann man mit den enthaltenen Zeitstempel die Empfängeruhr neu stellen, so dass diese synchron läuft. Jetzt lässt sich die Kreuzkorrellation anwenden und man kann durch Verfolgen der Änderungen des Sequenzstartes jederzeit die aktuellen Signallaufzeiten bestimmen. Aus den bekannten Positionen der Satelliten (errechenbar durch die empfangenen Bahndaten) kann man die Entfernungen zu den Satelliten berechen und daraus die Position des Empfängers.

Geschwindigkeitsmessung Mit dem GPS-System ist es ebenfalls möglich, die aktuelle Geschwindigkeit zu messen. Dies kann über folgende approximierte Formel geschehen:

$$v = \frac{\mathrm{d}x}{\mathrm{d}t} \simeq \frac{x(t_2) - x(t_1)}{t_2 - t_1}$$

In diese Formel gehen Positionsfehler (dazu mehr im folgenden Paragrafen) direkt in die Geschwindigkeit ein.

Eine weitere Möglichkeit besteht durch Ausnutzung des Dopplereffektes. Bekannt ist diese Phänomen bei Schallwellen (z.B. beim Martinshorn). Es gilt ebenfalls für elektromagnetische Wellen. Es gilt, dass die Frequenzverschiebung proportional zur relativen Geschwindigkeit von Sender und Empfänger ist. Diese Daten sind alle bekannt. Somit lässt sich die aktuelle Geschwindigkeit mit recht hoher Genauigkeit berechnen.

Genauigkeit Wie oben beschreiben, ist ein chip (des C/A-Codes) genau eine Mikrosekunde lang. Dies entspricht multipliziert mit der Lichtgeschwindigkeit einer Länge von ca. 300 m. Moderne Empfänger können einen chip bis auf einen Prozent genau ermitteln. Hieraus folgt eine Genauigkeit von ca. 3 m. Dieser Wert wird durch weitere Faktoren verschlechtert. Hier ist vorallem die Verzögerung des Signales in der Ionosphäre zu nennen. In dieser Atmospärenschicht werden elektromagnetische Wellen gebrochen, wodurch sich der Weg verlängert. Diesen Fehler kann man nicht herausrechnen. Mit weiteren kleineren Fehlerquellen ergibt sich eine Genauigkeit von ca. ± 15 m.

NMEA 0183 Die NMEA (siehe [23]) engagiert sich für die Ausbildung und den Fortschritt der Marine-Elektronikindustrie und dem Markt, den diese bedient. Es handelt sich dabei um eine nicht auf Profit ausgelegte Vereinigung von Herstellern, Vertreibern, Ausbildungsinstitutionen und anderen mit Interesse an diesem Markt.

Die NMEA hat einen Standard (NMEA 0183; siehe [22]) für die Übertragung von Daten der Marinegeräten definiert. Hierzu zählen auch GPS-Geräte. Dieser Standard sieht vor, dass der Sender seine Daten im RS-232-Standard (vom PC als Datenformat der COM-Schnittstelle bekannt) ausgibt. Die Datenrate beträgt 4800 baud. Die Daten werden im ASCII[4]-Format übertragen. Dabei sind alle druckbaren Zeichen sowie CR[5] und LF[6] erlaubt und die Daten werden in der Form von Sätzen übertragen. Jeder dieser Sätze beginnt mir dem Zeichen *$*. Darauf folgt eine fünf Zeichen langer Header, der sich in den standardisierten Sätzen aus einer zwei Zeichen langen Senderkennung und einer drei Zeichen langen Satzkennung zusammensetzt. Dann folgt eine Reihe von Datensätzen, die mit Kommas unterteilt werden. Schliesslich wird der Satz mit einer optionalen Prüfsumme und einer CR/LF abgeschlossen. Jeder Satz kann inklusive des führenden *$* und den beiden CR/LF bis zu 82 Zeichen enthalten. Ist ein Datenfeld in einem Satz zwar vorgesehen aber nicht verfügbar, so wird er einfach weggelassen, das dazugehörige Komma zur Trennung der Datensätze wird aber ohne Leerzeichen beibehalten. Durch Zählen der Kommas kann ein Empfänger dann aus jeden Satz die entsprechenden Informationen richtig zuordnen. Die meist optionale Prüfsumme besteht aus einem * und zwei Hexadezimalzahlen, die sich durch ein (bitweise) XOR

[4]American Standard Code for Information Interchange = Amerikanischer Standard Code für Informationsaustausch

[5]Carriage Return = Wagenrücklauf

[6]Line Feed = Neue Zeile

aller Zeichen zwischen dem *$* und dem * berechnen. Bei manchen Sätzen ist die Prüfsumme notwendig.

1.2 Das Kartenmaterial

Als Ausgangsbasis standen uns digitale topographische Kartenelemente des Landesvermessungsamtes in Bonn zur Verfügung, welche von der Projektgruppen-Leitung käuflich erworben wurden. Das Kartenmaterial stellt eine insgesamt 288 qkm große Fläche des Dortmunder Straßennetzes dar, wobei die einzelnen Kartenelemente jeweils 1 qkm große Kacheln zeigen. Ausgeliefert wurden die digitalen Kacheln im TIFF-Format in einer Auflösung von jeweils $2000 \cdot 2000$ Pixeln. Digitale topographische Karten bietet das Landesvermessungsamt in zwei verschiedenen Maßstäben an, 1:10.000 und 1:50.000 (siehe [16]). Die Karten der Projektgruppe stammen aus dem Kartenwerk DTK-10-V-NRW und haben, wie aus dem Namen ersichtlich, einen Maßstab von 1:10.000. Inhaltlich gliedert sich das Kartenwerk DTK-10-V-NRW in zwei Teile. Der erste Teil besteht aus den bereits erwähnten TIFF-Dateien, die die eigentliche Karte darstellen und in den einzelnen Dateinamen den String *DTK10G* enthalten. Der zweite Teil beinhaltet TFW-Dateien, welche durch den Text *DTK10RS* gekennzeichnet sind. Eine Kachel besteht aus einem Paar von TIFF-Datei und TFW-Datei, wobei die TFW-Datei lediglich Straßennamen kleinerer Straßen beinhaltet, für welche es in der Hauptkarte keinen Platz zur Bezeichnung gab. Die Projektgruppe arbeitete ausschließlich auf den TIFF-Dateien. Das geodätische Bezugssystem des Kartenmaterials ist das Potsdam-Datum (Zentralpunkt: Rauenberg), welches Gauß-Krüger-Koordinaten und das Besselellipsoid verwendet. Praktischerweise beinhalten die Dateinamen zusätlich die Gauß-Krüger-Koordinaten des unteren linken Pixels der jeweiligen Kachel. Beispielsweise hat der untere linke Pixel der Kachel *08_DTK10G_606_700.tif* den Gauß-Krüger Rechtswert (2)606000 und den Hochwert (5)700000. Die zu Beginn stehende *08* ist lediglich eine zusätzliche Numerierung der Kachel. Zur Veranschaulichung stellt Abbildung 1.2 die Anordnung der 288 uns vorliegenden Kartenelemente des Dortmunder Raumes dar.

Die in der linken Spalte stehenden Zahlen geben den jeweiligen Hochwert der Kachel an, die untenstehenden Ziffern den Rechtswert. Sollte das Projekt später fortgesetzt werden, kann die Karte jederzeit um weitere Kacheln ergänzt werden, da das Landesvermessungsamt für den gesamten Raum NRW die DTK-10-V-Karten anbietet.

1.3 Gauß-Krüger-Transformation

Das Programm des PDAs stellt die Routen auf den Karten des Landungsvermessungsamtes dar. Diese Karten sind im Gauß-Krüger-Format (GK) angegeben, d.h. sie benutzen als Achsen die GK Rechts- und Hochwerte. Da wir bei einigen Funktionen wie

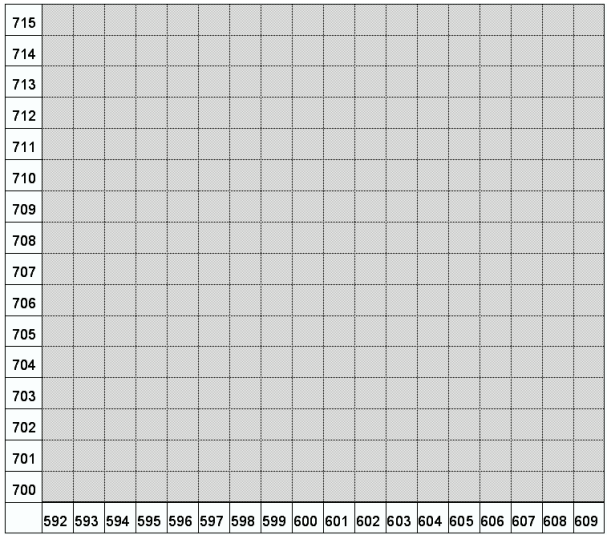


Abbildung 1.4: Kachelung der 288 Kartenelemente

dem Loggen GPS-Daten erhalten, ergab sich die Notwendigkeit der Umrechnung von GPS in Gauß-Krüger-Koordinaten.

1.3.1 Gauß-Krüger-System

Das GK-System wurde von Karl Friedrich Gauß und Johann Krüger entwickelt. Es ermöglicht jedem Punkt der Erde eine eindeutige Zuordnung über seine metrischen Rechts- und Hochwert. Der Rechtswert, vergleichbar mit dem Y-Wert eines kartesischen Koordinatensystems, gibt die Lage des Punktes in West-Ost Richtung an. Hierfür wird die Erde in 3° breite Meridianstreifen aufgeteilt. Die Berechnung erfolgt mit Hilfe einer Multiplikation und dem jeweiligen Mittelmeridian. Der Hochwert entspricht der X-Achse im kartesischen Koordinatensystem und entspricht der Nord-Süd Ausdehnung. Hierfür wird der Abstand in Metern vom Äquator gemessen. Die GPS-Punkte und GK unterscheiden sich in den Punkten Maßstab, Richtung des Koordinatensystems und Koordinatenursprung.

1.3.2 Mathematische Umrechnung der Koordinaten

Die von uns verwendete Transformation von GPS-Daten zu Gauß-Krüger-Koordinaten beruht auf einem Algorithmus zur Berechnung der Gauß-Krüger-Koordinaten von Ottmar Labonde (siehe [14]).

Im Folgenden wollen wir nun die elementaren Rechenschritte näher erläutern.

Wie oben beschrieben unterscheidet sich das GK System in drei Punkten. Daher ist es notwendig eine Änderung des Maßstabes, eine Rotation um die X-, Y-, Z-Achsen sowie eine Verschiebung des Koordinatensystems durchzuführen. Dies sind die drei Da-

tumsverschiebungskonstanten, die Achsenrotationsparameter sowie der Skalierungsfaktor. Aus diesen 7 Parametern ergibt sich auch der Name für das Kernstück der Umrechnung, der als 7-Parameter-Helmert-Transformation bekannt ist. Ihre Angabe ist ausschlaggebend für die Exaktheit der umgerechneten Werte. Ein weiterer wichtiger Faktor für die Genauigkeit ist die Nachkommastellenanzahl von PI.

Zuerst werden die Geographischen Koordinaten in das kartesische Koordinatensystem mit Ursprung Erdmittelpunkt überführt. Die Beziehung zwischen den kartesischen Koordinaten X_q, Y_q und Z_q und den ellipsoidischen Koordinaten ,sowie dem Querkrümmungsparameter N, ergibt sich durch:

$$X_q = (N + H) \cdot \cos\varphi \cdot \cos\lambda$$
$$Y_q = (N + H) \cdot \cos\varphi \cdot \sin\lambda$$
$$H = (b^2/a^2 + H) \cdot \sin\varphi$$

Danach erfolgt die eigentliche Transformation nach den Regeln der analytischen Geometrie. Das Ergebnis ist ein lokal angepasstes Ellipsoid, hier das Bessel-Ellipsoid, bestehend aus den neuen X-, Y- und Z-Koordinaten. Dieses Ellipsoid bezeichnet man als Referenzellipsoid, also ein abgeflachtes, symmetrisches Ellipsoid, das als Annäherung an die ideale Erdfigur einer bestimmten Region dient.

Danach wird das Bessel-Ellipsoid in die geographischen Daten im Potsdam-Datum umgerechnet. Als Datum kann man sich einen Referenzpunkt vorstellen. Wenn man sich auf dasselbe Datum bezieht, kann man problemlos ebene Koordinaten in geografische umrechnen und umgekehrt. Die astronomischen Messungen für das bis heute im Einsatz befindliche wichtigste Referenzsystem in Deutschland reichen ins 19. Jahrhundert zurück. Das geodätische Datum der deutschen Landesvermessung hat mehrere Bezeichnungen, jedoch dieselbe Bedeutung. Die gängigsten sind DHDN, Rauenberg-Datum und Potsdam-Datum. Im letzten Schritt werden die Koordinaten in ein ebenes, rechtwinkliges Koordinatensystem übertragen, das Gauß-Krüger-System. Die Berechnung erfolgt im Bessel-Ellipsoid selbst und Ergebnis sind Rechts- und Hochwert, die für die Routendarstellung auf der Stadtkarte Dortmund benötigt werden. (siehe [8]).

Wir erreichen mit dem Algorithmus für NRW eine Genauigkeit von ± 1 Meter, womit natürlich unser aktuelles Projekt Dortmund eingeschlossen ist. Durch einfache Veränderungen der Bezugssysteme lässt sich der Algorithmus für andere Regionen modifizieren.

Kapitel 2

Das Simulationsmodul

- Maik Drozdzynski • Andreas Gaubatz • Miguel Liebe -

SUMOiTT steht für **S**imulation of **U**rban **MO**bility **i**ntegrated **T**raffic **T**racer und bezeichnet die in dieser PG entwickelte Verkehrssimulationsumgebung zur Erzeugung künstlicher GPS-Spurdaten (GPS-Traces) im NMEA 0183 Format (siehe [22]). Der Einsatz bzw. die Entwicklung einer solchen Umgebung ist erforderlich, da im Rahmen dieser PG sehr große GPS-Spurdaten für die MapGeneration-Gruppe benötigt werden und es müßig wäre, die benötigten Datenmengen manuell, beispielsweise mit einem Garmin-Empfänger, zu sammeln.

2.1 Anforderungen

Die generierten Spurdaten sollten gewissen Ansprüchen genügen. In erster Linie müssen sie realistische Daten liefern. Dies bedeutet insbesondere, die Gegebenheiten der Strasse (zulässige Höchstgeschwindigkeit, Einbahnstraßen, Brücken etc.), den Zeitpunkt der simulierten Fahrt, die Ungenauigkeit der GPS-Messung und viele andere Faktoren zu berücksichtigen. Aufgrund der Komplexität dieser Aufgabe kann diese Forderung mitunter nur partiell erreicht werden. Die Wichtigkeit der Realitätstreue der generierten Daten wird anhand des folgenden Beispiels klar. Bekanntlich gibt es Straßen und Autobahnen, die in den Morgenstunden und am späten Nachmittag regelmäßig überfüllt sind. Wenn nun die Simulationssoftware Spurdaten erzeugen würde, die vollkommen unabhängig von der Tageszeit gleich hohe Geschwindigkeiten beinhalten würde, könnte eine essentielle Stärke des Navigationssystems nicht getestet werden. Es würden keine merklichen Unterschiede zwischen den Antworten zu Kürzeste-Wege-Anfragen bestehen, die einerseits um 8 Uhr Morgens und andererseits um 2 Uhr Nachts gestellt werden. Die Stärke des Navigationssystems besteht ja gerade in der Benutzung der dynamischen und zeitabhängigen Karten, in welchen implizit die Information über den Verkehrsfluss enthalten ist. Eine Möglichkeit der Realisierung für das Verkehrsverhalten besteht in der Angabe von Wahrscheinlichkeitsverteilungen für gewisse Aspekte einer Autofahrt. Dazu zählen Länge der Fahrt (typischerweise gibt

es mehr kurze als lange Fahrten), Zeitpunkt des Fahrtantritts (typischerweise gibt es tagsüber mehr Fahrten als Nachts), Fahrstil des Fahrers (i. d .R. gibt es mehr rücksichtsvoll fahrende Verkehrsteilnehmer als Raser) und vieles mehr. Auch die Simulation des Stauverhaltens vor Ampeln und auf Autobahnen ist ein schwieriges Problem.

2.2 Die Ausgangslage

Es wurde eine topographische Karte der Stadt Dortmund (TIFF-Format im Maßstab 1:10.000) von der Projektgruppenleitung erworben. Diese Karte ist in 288 Kacheln gleicher Größe unterteilt. Es ist ohne große Probleme möglich, diese Kacheln zu einer großen Gesamtkarte zusammenzufügen. Wichtig ist auch, dass die Kacheln farblich nicht verrauscht sind. Dies macht eine für uns erforderliche Weiterverarbeitung möglich. Um eine geeignete Vekehrssimulation durchzuführen, müssen in einem ersten Schritt die Kacheln bzw. die Pixelkarten vektorisiert werden, um einen gerichteten und bewerteten Straßengraphen mit Zusatzinformationen (Spuranzahl einer Straße, maximal zulässige Geschwindigkeit) zu generieren. In einem zweiten Schritt soll auf Basis des generierten Straßengraphen eine möglichst realitätsgetreue Verkehrssimulation ermöglicht werden. Hierbei sollen die Fahrzeuge GPS-Spurdaten in einem geeigneten Format ausgeben.

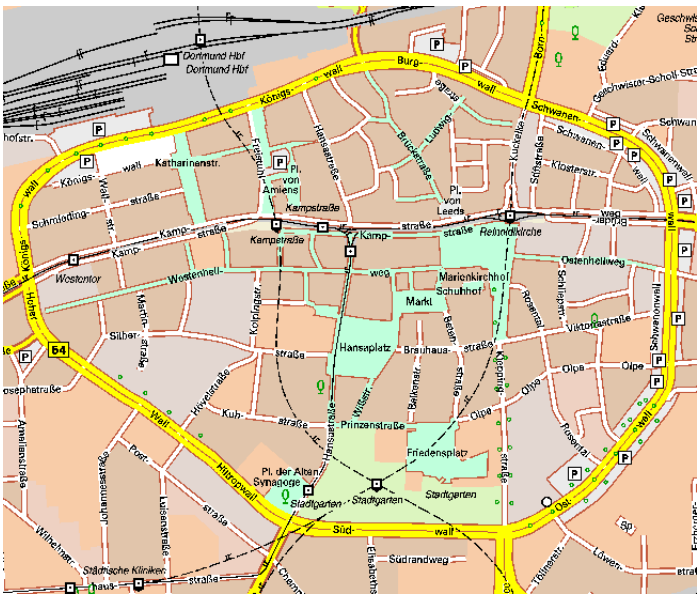


Abbildung 2.1: TIFF-Straßenkacheln im Rohformat. Hier: Dortmunder Ring

2.3 Vom Bitmap zum Graphen

2.3.1 Die Idee

Wie bereits angesprochen, stehen der Projektgruppe Kacheln im TIFF-Format zur Verfügung. Die grundlegende Idee besteht darin die Straßenflächen zu extrahieren und darauf aufbauend den Graph mit einem geeigneten Tracking-Algorithmus zu generieren. Problematisch hierbei ist jedoch die unterschiedliche Farbgebung der Straßen auf dem Kartenmaterial. So sind normale Straßen in der Stadt weiß, Autobahnen rot und Landstraßen gelb. Daher ist zunächst der Entwurf eines Algorithmus von Bedeutung, welcher alle Straßen erkennt und diese aus den Kacheln extrahiert. Aufgrund dieser Gegebenheiten ist es sinnvoll iterativ vorzugehen. Die suksessive Anwendung der graphischen Filter ermöglicht es die Straßenflächen so zu bereinigen, dass die Vektorisierung durchgeführt werden kann.

2.3.2 Die Algorithmen

Extrahierung der Straßenflächen Jedem Bildpunkt (Position i, j) des verfügbaren Kartenmaterials ist ein Farbwert zugeordnet. Die Farbe selbst ist als Tripel von rotem, grünem und blauem Farbanteil gegeben (RGB-Farbskala). Vorteilhafterweise sind die Karten nicht verrauscht, was bedeutet, dass sich die farbigen Flächen nicht aus einer Vielzahl unterschiedlicher RGB-Werte ergeben, sondern alle einen eindeutigen Farbwert haben. Aufgrund dieser Eigenschaft ist es möglich, zunächst die für die Straßengraphgenerierung relevanten Straßenflächen vom Rest zu trennen. Zu diesem Zweck müssen vom Benutzer lediglich die Farbwerte der gewünschten Straßen angegeben werden. Ein einfacher Algorithmus setzt die Farbwerte derjenigen Pixel auf schwarz, welche die vorgegebenen Farben haben. Alle Pixel, die nicht eine der vordefinierten Straßenfarben haben werden schließlich auf weiß gesetzt. Die Laufzeit der Extrahierung beträgt $O(n \cdot m)$, wobei n im folgenden stets die Bitmapbreite und m die Höhe bezeichnet.

Die Anwendung dieses Algorithmus reicht nicht zur Generierung des Straßengraphen aus. Zum einen gibt es schwarze Schriftzüge, welche Straßennamen, Ortsteile und andere wichtige Orte angeben und innerhalb der Straßen verlaufen oder die Straßen schneiden. Darüber hinaus sind auch alle Straßenbahn- und Eisenbahnstrecken in schwarz markiert. Man könnte nun einerseits die Farbe schwarz als Straßenfarbe definieren. In diesem Fall wählt man allerdings auch gleichzeitig alle Bahnstrecken als Straßenfläche aus. Andererseits könnte man die Farbe schwarz nicht auswählen, würde somit aber weiße Lücken auf den Straßenflächen erhalten. Der größte Teil der Straßen ist in den TIFF-Dateien als weiße Fläche gegeben. Bei der Auswahl dieser Farbe werden dann allerdings auch Sportplatze und Parkplatzmarker und andere kleinere Dinge ausgewählt. Die Extraktion der Straßenflächen (siehe Abbildung 2.2) ist also nicht trivial und bedarf einiger manueller und auch automatisierter Nachbearbeitungen. Die manuelle Nachbearbeitung besteht aus einfachen Mal- bzw Radierwerkzeugen, wie

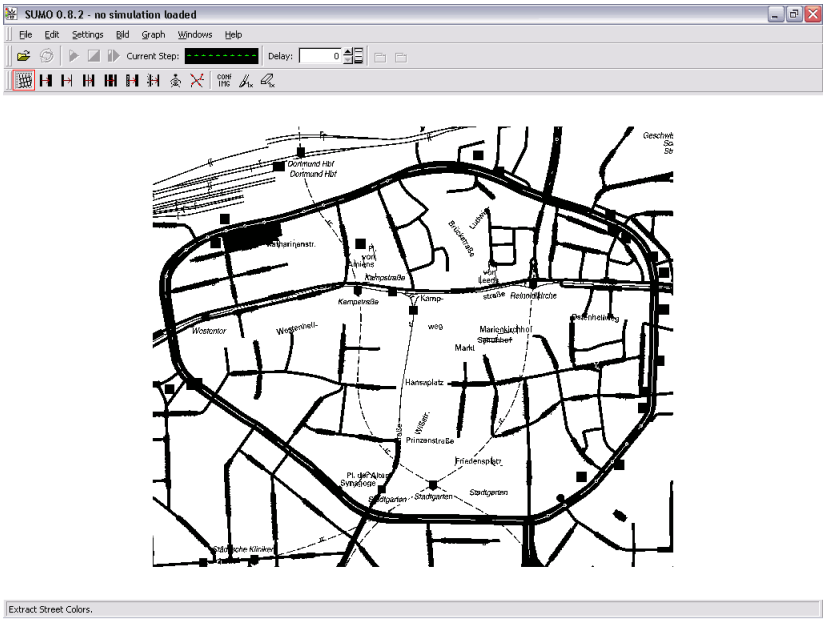


Abbildung 2.2: Die Extraktion der Straßenflächen

sie aus jedem Bildbearbeitungsprogramm bekannt sind. Zu den automatisierten Nachbearbeitungsmethoden zählen:

- Erosion
- Dilatation
- Morphologisches Öffnen
- Morphologisches Schliessen
- Lücken schliessen
- Fragmente entfernen

Erosion Für zwei Mengen A und B in Z^2 ist die Erosion von A bezüglich B, bezeichnet mit $A \ominus B$, definiert als

$$A \ominus B = \{z|(B)_z \subseteq A\},$$

wobei die Menge Z^2 die zweidimensionale Menge der Pixel eines Rasterbildes darstellt. Eine Erosion von A und B besteht also aus allen Punkten z, für die gilt, dass B um z verschoben (translatiert) in A enthalten ist. Die Menge B wird in diesem Zusammenhang als *strukturierendes Element* bezeichnet. Die Form und Größe des strukturierenden Elementes ist für das Ergebnis der Erosion entscheidend. Aufgrund der Aufgabenstellung ist die Wahl eines symmetrischen strukturierenden Elementes

notwendig, da ansonsten die Straßenflächen verzerrt werden. Weil durch die Erosion Pixel vom Rand der schwarzen Straßenflächen aus gelöscht werden, darf das strukturierende Element darüber hinaus nicht zu groß sein, da unter Umständen dünne Straßenflächen komplett gelöscht werden. Aus diesem Grund wurde ein $3 \cdot 3$ Pixel großes strukturierendes Element gewählt, wobei der Benutzer durch mehrfache Erosion die Straßenflächen stetig verdünnen kann. Auf diese Weise werden bereits ein Großteil der störenden Schriftzüge, sowie Straßenbahn- und Eisenbahnschienen eliminiert. Die Laufzeit des Algorithmus beträgt $O(n \cdot m \cdot s(B))$, wobei $s(B)$ die Anzahl der Pixel des strukurierenden Elements ist.

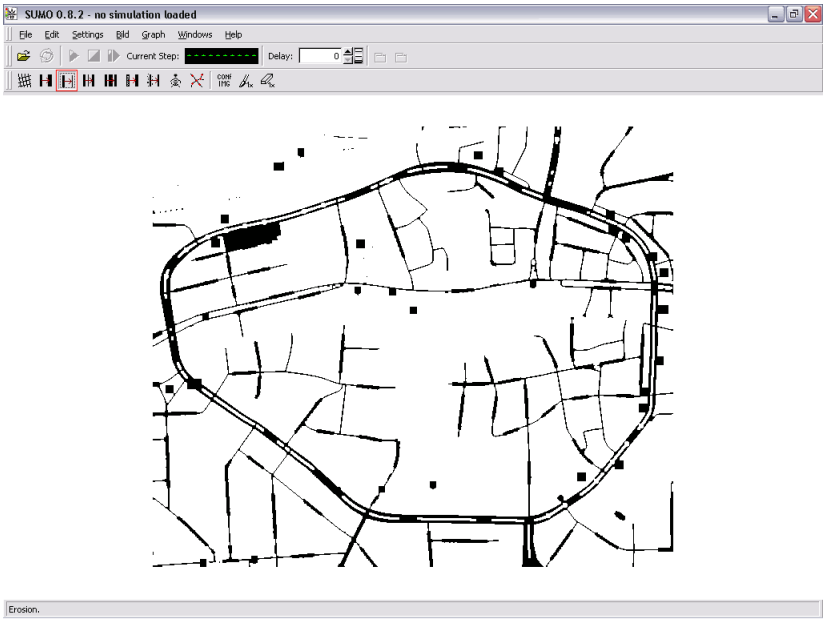


Abbildung 2.3: Erosion des 0/1 Bitmaps

Dilatation Für zwei Mengen A und B in Z^2 ist die Dilatation, bezeichnet mit $A \oplus B$, definiert als

$$A \oplus B = \{z | (\hat{B})_z \cap A \neq \emptyset\},$$

wobei $\hat{B}$ die Reflektion von B ist. Sie ist folgendermaßen definiert :

$$\hat{B} = \{w | w = -b, b \in B\}.$$

Somit sind Erosion und Dilatation komplementäre Operationen und es gilt die Gleichung

$$(A \ominus B)^c = A^c \oplus \hat{B}.$$

Das Komplemet von A ist die Menge der Elemente, die nicht in A enthalten sind, also

$$A^c = \{w | w \notin A\} = Z^2 - A.$$

Auch bei der Dilatation wurde ein $3 \cdot 3$ Pixel großes strukturierendes Element gewählt. Durch die Benutzung des Dilatationsfilters können kleine Lücken innerhalb der Straßenflächen verkleinert oder geschlossen werden. Bei der Benutzung muss der Anwender darauf achten, dass eng beieinander liegende Straßenflächen nicht zu einer Großen Gesamtfläche verschmolzen werden. Um nun weitere Lücken zu schließen kann der Filter `Lücken schließen` verwendet werden. Die Laufzeiten des Dilatations- und des Erosionsalgorithmus sind identisch.

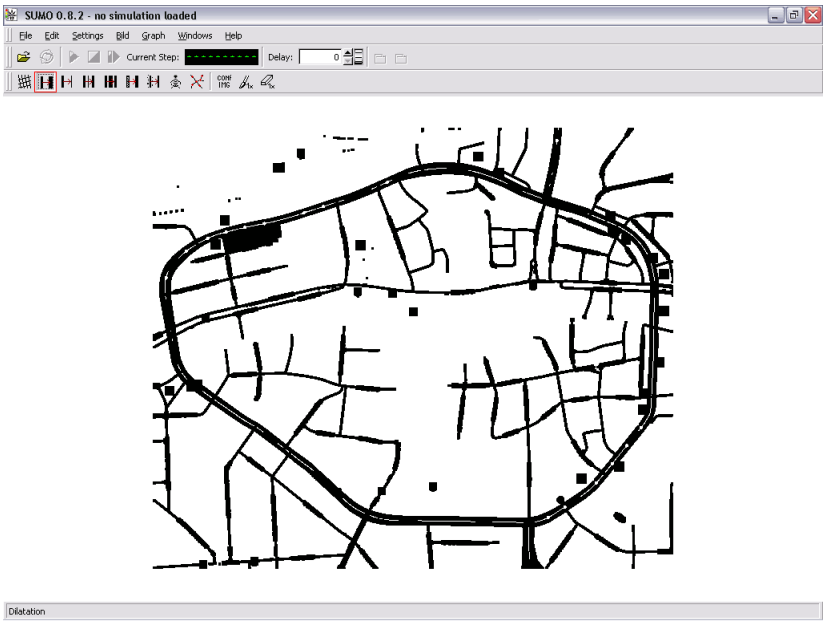


Abbildung 2.4: Dilatation des 0/1 Bitmap

Morphologisches Öffnen Das morphologische Öffnen einer Menge A durch das strukturierende Element B (beide aus Z^2), bezeichnet mit $A \circ B$, ist definiert als

$$A \circ B = (A \ominus B) \oplus B$$

Es ist also nichts anderes, als eine Erosion von A und B, unmittelbar gefolgt von einer Dilatation des Erosionsergebnisses. Durch diese Operation werden schmale Verbindungen zwischen schwarzen Flächen aufgebrochen, die Konturen werden abgerundet und kleine Vorsprünge eliminiert.

Morphologisches Schließen Das morphologische Schließen einer Menge A durch das strukturierende Element B (beide aus Z^2), bezeichnet mit $A \bullet B$, ist definiert als

$$A \bullet B = (A \oplus B) \ominus B$$

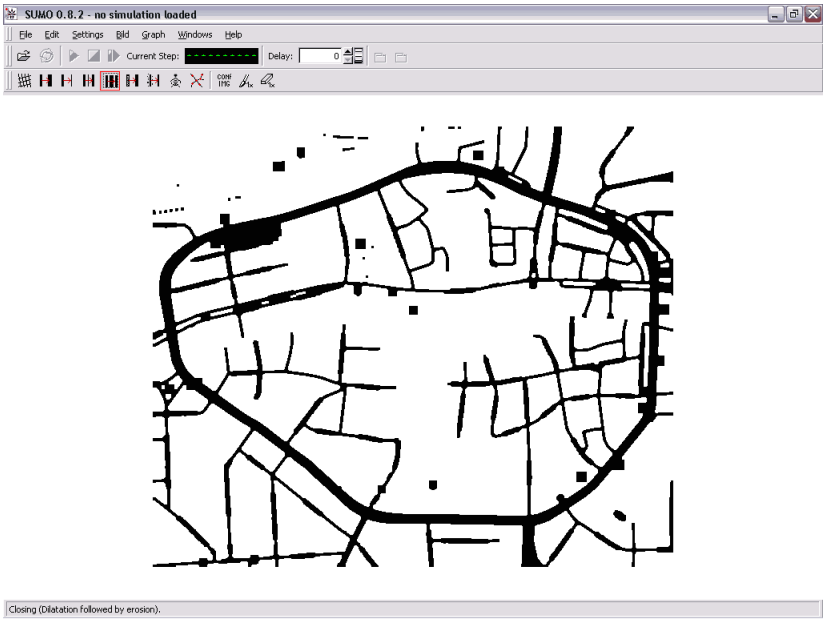


Abbildung 2.5: *Morphologisches Öffnen*

Durch diese Operation werden ebenfalls Konturen geglättet, allerdings werden im Gegensatz zum morphologischen Öffnen schmale Verbindungen zwischen schwarzen Flächen verstärkt, und kleine Lücken und Einbuchtungen gefüllt.

Sowohl das morphologische Öffnen als auch das morphplogische Schließen haben eine Laufzeit von $O(2c \cdot n \cdot m) = O(n \cdot m)$.

Lücken schließen Ein Problem welches nicht allein durch Dilatationen gelöst werden kann ist die Eliminierung von *Lücken* innerhalb breiter Straßenflächen, z.B. bei Autobahnen. Zwar können solche Lücken durch mehrfache Dilatationen geschlossen werden, allerdings werden dann viele Straßenflächen verschmolzen. Abhilfe schafft in diesem Zusammenhang der Algorithmus `Lücken schließen`, welcher lediglich für alle weißen Pixel überprüft, ob dieser mindestens $i \in \{1, 2, \ldots, 8\}$ schwarze Nachbarpixel haben. Die Anzahl i soll vom Benutzer festgelegt werden können, wobei sich ein Wert von $i = 5$ als praktikabel erwiesen hat. Auch hier beträgt die Laufzeit $O(n \cdot m)$

Fragmente entfernen Durch die Erosion werden die Straßenbahnschienen nicht ganz eliminiert. Außerdem befinden sich unter anderem Sportplätze und verschiedene Kartensymbole auf dem s/w-Bitmap. Ziel dieses Algorithmus ist es, diese schwarzen Stellen zu finden und weiß zu färben.

Es wird jedes Pixel des Bitmaps untersucht. Findet sich ein schwarzes Pixel x, wird in weiteren Schritten die Umgebung von x analysiert. Es wird die Eigenschaft ausgenutzt, dass Straßenzüge stets aufeinanderfolgende schwarze Pixel besitzen. Sind

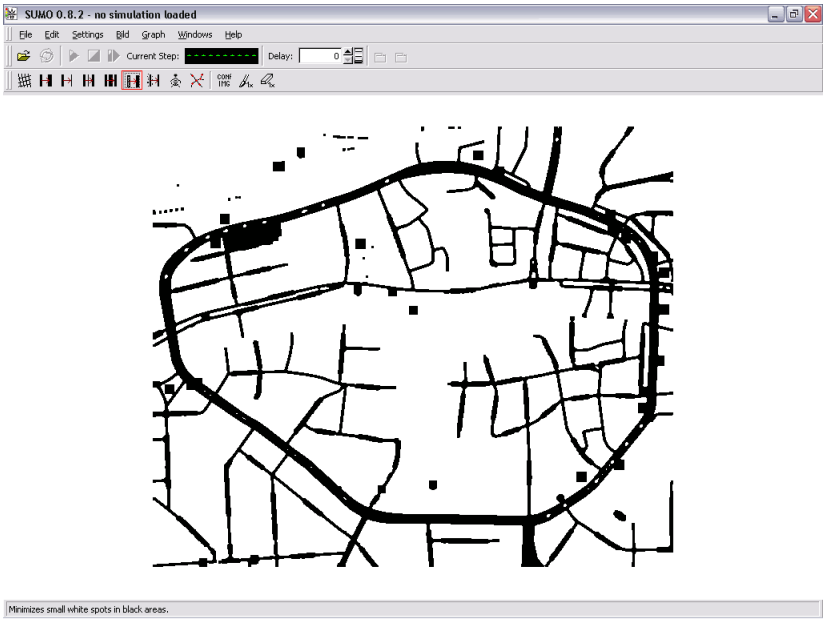


Abbildung 2.6: *Lücken schließen*

nun alle umgebenden Pixel um x weiß, kann man davon ausgehen, dass x isoliert von anderen schwarzen Pixeln ist und weiß gefärbt werden kann.

In der Methode wird ein ungerader Integerwert i größer 3 übergeben, der die maximale Größe eines Quadrates angibt, welches um x herum betrachtet wird. In einer Schleife wird iterativ das Quadrat von Kantenlänge 3 bis i vergrößert. Bestehen die Kanten eines Quadrates zu einem Zeitpunkt nur aus weißen Pixeln, so wird der Flächeninhalt weiß gefärbt und die Iteration abgebrochen. Ist dies nicht der Fall wird das Quadrat entsprechend der Iteration (bis zur maximalen Kantenlänge i) vergrößert. Die Laufzeit beträgt $O(n \cdot m \cdot i)$.

Skelettierung Nachdem die Straßenflächen durch die vorgestellten Filter sowie durch manuelle Nachbearbeitung sauber extrahiert wurden, führt ein Skelettierungsalgorithmus zu einer Darstellung, aus welcher dann im nächsten Schritt ein Straßengraph entwickelt werden kann. Das Skelett einer s/w-Pixeldatei ist, anschaulich gesagt, eine Menge von dünnen Linien, welche die Mittelachsen der ursprünglichen schwarzen Flächen darstellen. Auf die Straßenflächen bezogen bedeutet dies, dass das Straßenskelett die Mittellinien der ursprünglichen Straßenflächen darstellen soll. In der mathematischen Morphologie (siehe [13]) ist das Skelett von A, bezeichnet mit $S(A)$, definiert als

$$S(A) = \bigcup_{k=0}^{K} S_k(A)$$

mit

$$S_k(A) = (A \ominus kB) - (A \ominus kB) \circ B$$

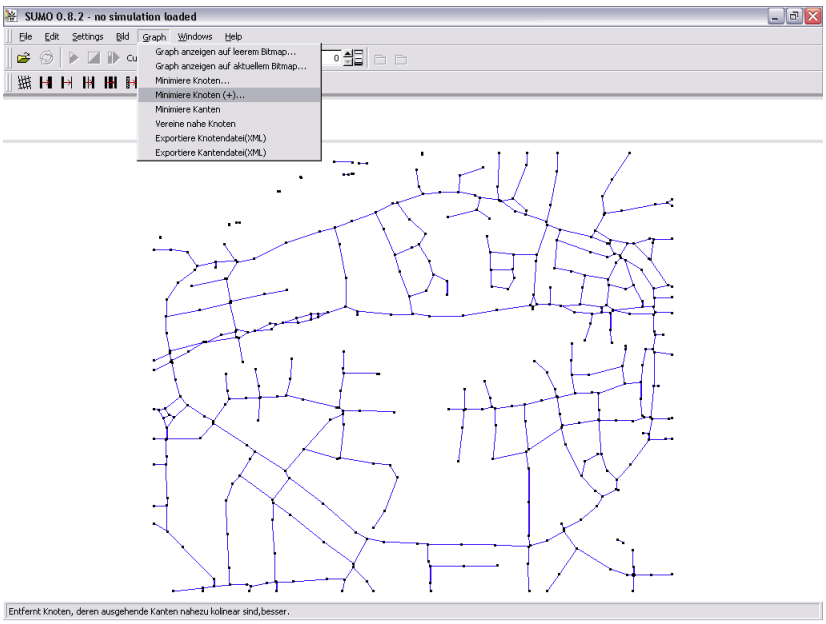


Abbildung 2.7: Fragmente entfernen

wobei B das strukturierende Element ist und

$$(A \ominus kB) = (\ldots(A \ominus B) \ominus B) \ominus B \ominus \ldots) \ominus B$$

gilt. K ist der letzte iterative Schritt bevor A zu einer leeren Menge erodiert, also

$$K = \max\{k|(A \ominus kB) \neq \emptyset\}$$

Eine erste Implementierung dieser klassischen Definition eines Skelettes hat nicht das gewünschte Ergebnis gebracht. Durch obige Definition wird nicht der Zusammenhang des Skeletts garantiert. Dies hatte zur Folge, dass gerade an Straßenkreuzungen (und anderen Stellen) das Skelett zerbricht und eine Grapherstellung aus dem Skelett sehr schwierig oder unmöglich wird. Eine Lösung bietet ein heuristischer Ansatz welcher von Blum 1967 (siehe [11]) vorgestellt wurde. Dieser basiert auf der sogenannten Mittelaxentransformation (MAT). Die MAT einer Region R mit Grenze G ist folgendermaßen definiert. Für jeden Punkt p in R finde man alle nächsten Nachbarn auf der Grenze G. Falls p mehr als einen nächsten Nachbarn hat gehört er zu der Mittelachse, also dem Skelett. Eine noch anschaulichere Definition einer MAT kann über ein Steppenfeuer gemacht werden. Man stelle sich eine zusammenhängende Fläche homogenen und trockenen Steppengrases vor, welches gleichzeitig an seinen Grenzen entzündet wird. Die MAT dieser Grasregion besteht aus allen Punkten, welche von mindestens zwei Brandfronten gleichzeitig erreicht wird. Eine direkte Umsetzung dieser Idee führt zu ineffizienten Algorithmen, da die Distanzen von jedem inneren Punkt zu allen Randpunkten berechnet werden müssten. Der in unserer Implementierung benutzte Algorithmus geht folgendermaßen vor: Die zu skelettierende s/w-Grafik wird in

2 Basisschritten bearbeitet, welche dann sukzessive wiederholt werden, bis keine Veränderungen mehr gemacht werden. Dabei seien die betrachteten Pixel nach folgendem Schema angeordnet, wobei das aktuelle Pixel das mittlere Pixel p_1 sei:

p_9	p_2	p_3
p_8	p_1	p_4
p_7	p_6	p_5

In Schritt 1 werden alle schwarzen Pixel (zwecks späterer Löschung) markiert, wenn folgende Bedingungen erfüllt sind:

1) $2 \leq N(p_1) \leq 6$
2) $T(p_1) = 1$
3.1) $p_2 \wedge p_4 \wedge p_6 = 0$
4.1) $p_4 \wedge p_6 \wedge p_8 = 0$

wobei $N(p_1) = p_2 + p_3 + \cdots + p_9$ und $T(p_1)$ entspricht der Anzahl der 0/1-Übergänge in der Sequenz $p_2, p_3, p_4, \ldots, p_9, p_2$. Sind diese Bedingungen für alle Pixel überprüft worden werden diese auf dem Originalbild gelöscht. Das manipulierte Bild ist die Eingabe für den nächsten Basisschritt. In Schritt 2 werden wieder alle schwarzen Pixel (zwecks späterer Löschung) markiert, wenn nun folgende Bedingungen erfüllt sind:

1) $2 \leq N(p_1) \leq 6$
2) $T(p_1) = 1$
3.2) $p_2 \wedge p_4 \wedge p_8 = 0$
4.2) $p_2 \wedge p_6 \wedge p_8 = 0$

Nach dem zweiten Schritt wird entsprechend dem ersten Basisschritt verfahren. Bedingung *1)* wird verletzt, wenn p_1 einen, sieben oder acht schwarze Nachbarn hat. Im Fall von nur einem Nachbarn darf p_1 offensichtlich nicht gelöscht werden, da dieser Pixel sich am Ende einer Pixelkette befindet. Würde man schwarze Pixel mit 7 oder 8 schwarzen Nachbarn löschen, würde man diese Region erodieren. Bedingung 2 verhindert, dass das Skelett bei der Erzeugung zerfällt, da so keine Pixel gelöscht werden, die auf einer 1-Pixel dünnen Linie liegen. Ein Pixel, das die Bedingungen *3.1)* und *4.1)* sowie die Bedingungen *1)* und *2)* erfüllt ist entweder ein Pixel auf der Ost- oder Südgrenze, oder ein Nordwest-Eckpunkt. In diesen Fällen gehört das Pixel nicht zum Skelett und sollte entfernt werden. Auf gleiche Weise lässt sich der zweite Basisschritt interpretieren, wobei Bedingungen *3.2)* und *4.2)* zusammen mit den ersten beiden Bedingungen das Löschen von Pixeln auf der Nord- und Westgrenze, sowie von Südost-Eckpunkten veranlassen. Für Nordost-Eckpixel gilt $p_2 = 0$ und $p_4 = 0$, und somit sind sowohl Bedingungen *3.1)* und *4.1)* als auch *3.2)* und *4.2)* erfüllt. Gleiches gilt für Südwest-Eckpunkte, mit $p_6 = 0$ und $p_8 = 0$. Die Laufzeit der Skelettierung beträgt $O(n \cdot m \cdot d_{max}/2)$. Anschaulich gesprochen ist d_{max} die Dicke der dicksten Straße.

Dieser Wert geht nur zur Hälfte in die Komplexität ein, da sich der Algorithmus von zwei Seiten der Straßenmittelachse nähert.

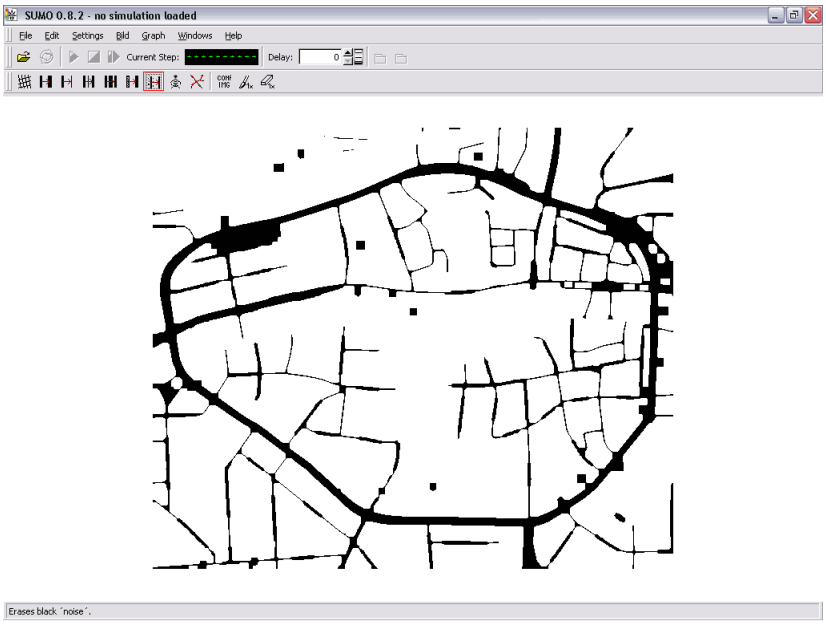


Abbildung 2.8: Skelettierte Straßenkarte

2.3.3 Code-Beispiel: *Die Skelettierung*

Im Anschluss an die Skelettierung folgt noch ein Algorithmus, der das Skelett maximal verdünnt. Dabei enthält ein maximal verdünntes Skelett keine Pixel, die entfernt werden könnten, ohne dass das Skelett zerfällt. Eine Ausnahme bilden Punkte, die lediglich einen schwarzen Nachbarn haben. Erst nach der maximalen Verdünnung ist das Skelett für die Vektorisierung geeignet.

2.3.4 Die Erzeugung des Straßengraphen

Nachdem das Skelett auf geeignete Weise vorbereitet wurde, kommt nun ein Trackingalgorithmus zum Einsatz, welcher einen zusammenhängenden und gerichteten Graphen erzeugt. Der Algorithmus basiert auf der Sweepline-Technik. Die Pixel werden spaltenweise (Sweepline) betrachtet. Sobald ein Pixel gefunden wird, welcher zum Skelett gehört, wird die Subroutine `Pixelcounter` aufgerufen, welche die Zusammenhangskomponente bearbeitet, die zu dem gefundenen Pixel gehört. `Pixelcounter` erzeugt an allen Sackgassenendpunkten Knoten. Darüber hinaus werden auf Pixelketten in regelmäßigen Abständen (vom Benutzer anzugeben) weitere Knoten und Kanten mit ihren Vorgängern erzeugt. Sobald ein Kreuzungspixel erreicht wird, wird ebenfalls ein Knoten erzeugt und die mindestens zwei neuen Nachbarn des

Pixel in eine Liste geschrieben. Diese Liste wird dann iterativ abgearbeitet, wobei für jeden Nachbarn `Pixelcounter` rekursiv aufgerufen wird. Dabei war insbesondere zu beachten, dass nicht alle in den Listen gespeicherten Nachfolger eines Kreuzungspixels aufgerufen werden müssen. Es kann vorkommen, dass man durch die Traversierung des Skelettes auf diese Weise zu Kreuzungsknoten gelangt, die bereits zuvor besucht wurden und deren Nachbarliste noch nicht vollständig abgearbeitet wurde. Eine spezielle farbliche Markierung der noch nicht abgearbeiteten Nachfolgerpixel eines Kreuzungspixels löst dieses Problem. Erst, wenn auf der durch den Sweeplinealgorithmus gefundenen Zusammenhangskomponente der Graph erzeugt wurde, traversiert die Sweepline weiter das Bild und sucht die nächste Komponente. Dies funktioniert allerdings nur, wenn im Algorithmus `Pixelcounter` die bereits besuchten Pixel entsprechend markiert werden, damit sie von der Sweepline im folgenden ignoriert werden. Setzt man einen Knotenzähler ein, der die zu einer Komponente gehörigen Knoten zählt, lassen sich Zusammenhangskomponenten finden, die möglicherweise wenige Knoten haben und gelöscht werden können (Skelette von Parkplätzen. Die minimale Anzahl der Knoten der Zusammenhangskomponenten, die akzeptiert werden, muß vom Benutzer angegeben werden.

2.3.5 Die Datenstruktur für den Graphen

An dieser Stelle wurde auf die Verwendung bereits bestehender Datenstrukturen für Graphen verzichtet. Die Datenstruktur ist recht einfach gehalten und besteht aus den Klassen `Vertex`, `Edge` und der Hauptklasse `Graph`. In letzterer sind die gängigen Methoden für Graphmanipulationen implementiert. Zusätzlich wurde eine Methode nach dem Prinzip der Tiefensuche entwickelt, die einen zufälligen Pfad durch den Graphen mit beliebig vielen Knoten zurückliefert. Dies ist wichtig, um eine statische Simulation auf dem Graphen zu erzeugen. Statisch heißt, dass GPS-Traces auf dem Pfad nur an Knoten ausgegeben werden und nicht etwa auf den Kanten. Dies ist sinnvoll, um große Datenmengen für Testzwecke an andere Untergruppen weiterzugeben. Authentischen Charakter haben diese Traces jedoch nicht.

In der Klasse `Vertex` findet sich eine Methode, die zu jeder Gauß-Krüger-Koordinate die entsprechenden $longitude$- und $latitude$-Werte des zugehörigen GPS-Datums zuordnet.

Die Knoten und Kanten werden in jeweils einem Vektor gespeichert. Entsprechende Templates werden von `Fox Toolkit` bereitgestellt. Zu jedem Knoten werden Attribute für die x- und y-Koordinaten verwaltet. Zudem besitzt jedes Knotenobjekt zwei Vektoren, um Adjazenzen zum Vorgänger- und Nachfolgerknoten auszudrücken. Kantenobjekte haben einen Startknoten s und einen Zielknoten t. Mit diesen beiden Attributen werden Kanten eindeutig identifiziert. Es ist außerdem sinnvoll, die Länge einer Kante zu verwalten. Sie errechnet sich aus dem Euklidischen Abstand l von s und t. Mit entsprechenden `Get()`-Methoden für die Koordinaten der Knoten erhält man folgende Formel:

$$l = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Mit einem Klick auf den Menüeintrag *Graph* → *Graph anzeigen*, ist es möglich, den Graphen anzuzeigen.

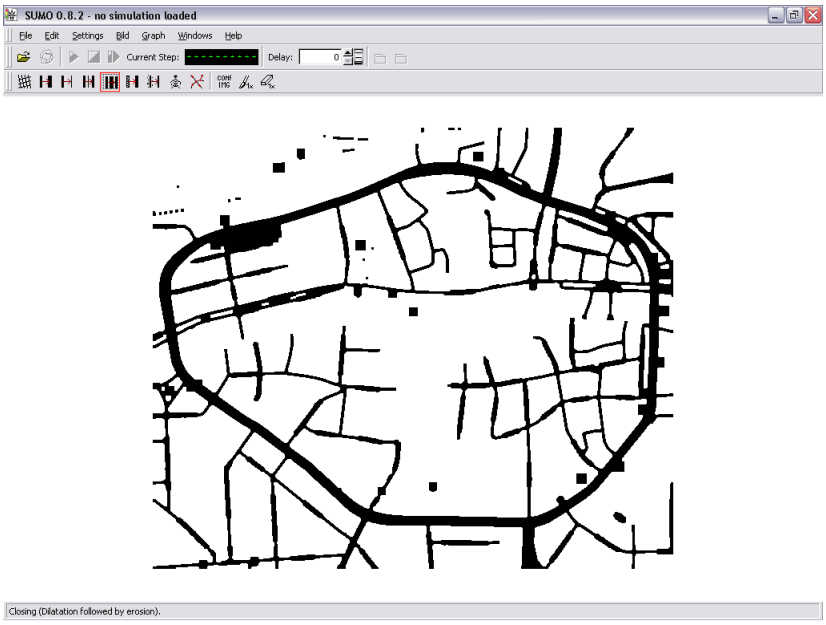


Abbildung 2.9: Der Straßengraph vor der Bearbeitung

Anhand der gelben Linien kann man etwa die Abweichung zu den Straßen erkennen. Je geringer der Knotenabstand gewählt wird, desto genauer ist der Graph. Es gilt, ein gutes Mittelmaß zwischen Speicherplatz und Genauigkeit zu finden.

2.3.6 Die Vereinfachung des Straßengraphen

Der durch den Trackingalgorithmus erzeugte Graph hat sehr viele Knoten. Hochgerechnet auf extrem große Straßenkarten wären dies zu viele Daten. Viele der Knoten und Kanten sind überflüssig. Zum Teil, weil Knoten auf geraden Straßenzügen liegen können, oder weil Knoten sich unter Umständen sehr dicht bei anderen Knoten befinden (tritt häufig bei Kreuzungen auf). In einem solchen Fall kann man die Knoten miteinander verschmelzen. So kann zum Beispiel von einer Kachel die Menge der Knoten um mehr als die Hälfte reduziert werden, ohne dass die Kanten des Graphen das Straßenbild zu sehr verzerren.

Löschen überflüssiger Knoten In diesem Zusammenhang ist ein Knoten als überflüssig zu bezeichnen, wenn er sich auf einem geraden Straßenzug befindet. Typischerweise hat er dann Grad 2. Es dürfen natürlich keine Knoten aus kurvenreichen Straßenverläufen herausgelöscht werden.

In einer Iteration werden stets eine Folge von drei Knoten x, y, z und die zugehörigen Kanten $a = (x, y)$ sowie $b = (y, z)$ betrachtet. Bei diesem Algorithmus bedienen

wir uns eines Kolinearitätstests, der auf die Kanten a und b angewendet wird. Sind die Kanten parallel, so liegt der Knoten y auf einer Geraden und kann bedenkenlos herausgelöscht werden. Gleiches passiert, wenn die Kanten zwar nicht parallel sind, aber eine gewisse Kolinearitätstoleranz (mit ϵ bezeichnet) nicht überschritten wird. Es wird die nächste Iteration mit den Knoten x, z und v begonnen, wobei v der nächste Knoten auf dem Straßenzug ist. Wird die Toleranz überschritten, d.h., die Kanten liegen nicht annähernd auf einer Geraden, kann der Knoten y nicht entfernt werden. Es wird die nächste Iteration mit den Knoten y, z und v begonnen, wobei auch in diesem Fall v der nächste Knoten auf dem Straßenzug ist.

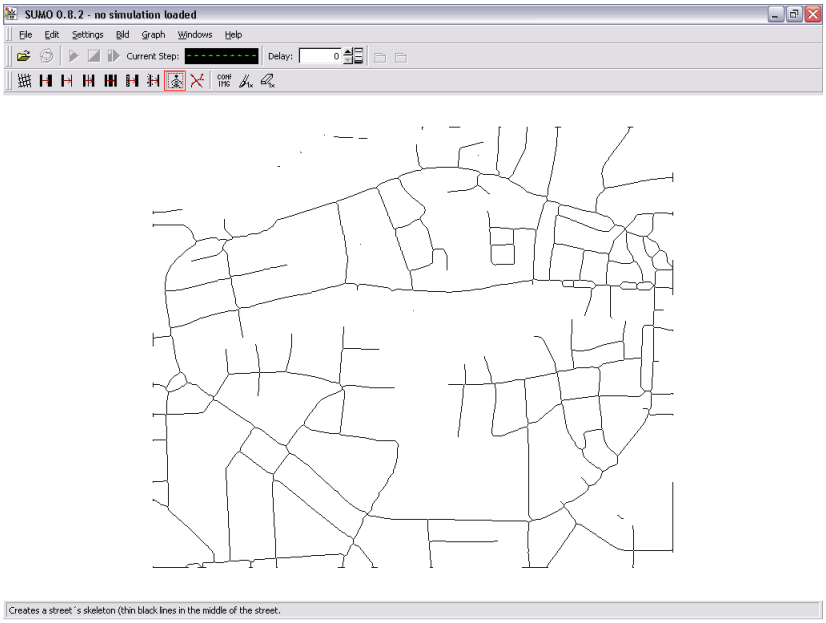


Abbildung 2.10: Der Straßengraph nach Löschen überflüssiger Knoten

Die Güte des Ergebnisses ändert sich mit der Variation von ϵ. Wählt man es relativ klein, werden verhältnismäßig wenige Knoten gelöscht und die Kurven gut modelliert. Ist ϵ hingegen groß, so werden zwar viele Knoten gelöscht, aber die Kurven werden unter Umständen nicht hinreichend modelliert, so dass Verzerrungen des Straßenbildes möglich sind (Straßen könnten plötzlich durch Vorgärten verlaufen). Es galt, einen hinreichenden Wert für ϵ zu finden.

Verschmelzung nahe beieinander liegender Knoten Zum Auffinden derjenigen Knoten, welche zu nah beieinander liegen, wird folgendermaßen vorgegangen.

In einer Schleife schaut man sich jede einzelne Kante e des Graphen an. Hier wird nun ausgenutzt, dass jeder Kante bei ihrer Konstruktion die Länge als Attribut übergeben wird. Die Länge der Kante versteht sich hier als Euklidischer Abstand der Start- bzw. Endknoten. Ist eine Kante sehr kurz, liegen offensichlich die zugehörigen Start-

und Endknoten nahe beieinander. Der Methode wird ein Integer-Wert $tolerance$ übergeben, der die minimal zulässige Länge einer Kante bestimmt.

Nehmen wir im Folgenden an, die Kante e sei zu kurz. Der Startknoten wird mit s bezeichnet und der Endknoten mit t. Durch die Verschmelzung entsteht ein neuer Knoten m, der in der Mitte von s und t liegen soll. Die Kante e wird gelöscht.

Nun betrachten wir den Knoten s (Für den Knoten t verläuft der Vorgang analog). In einem Hilfsarray werden alle Nachfolgeknoten x von s gespeichert. Alle Kanten (x, s) und (s, x) werden gelöscht. Nun müssen lediglich neue Kanten (x, m) und (m, x) hinzugefügt werden.

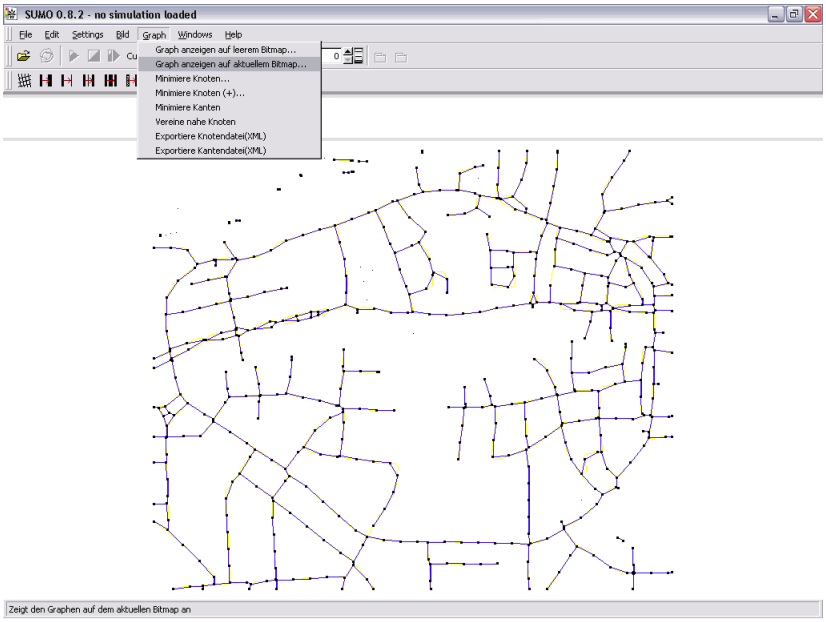


Abbildung 2.11: Der Straßengraph nach Verschmelzung nahe beieinander liegender Knoten

Nun ist der iterative Teil abgeschlossen. Es ist natürlich möglich, einige Algorithmen mehrmals durchzuführen, wie zum Beispiel eine Dilatation oder eine Erosion. Hingegen darf eine Extrahierung der Straßenflächen nur einmal durchgeführt werden, genau wie die Skelettierung und die Vektorisierung. Es wurden für jeden Algorithmus Flags implementiert. Sind die jeweiligen Flags $true$, so darf der Algorithmus im Laufe der Iteration noch durchgeführt werden. Sind bestimmte Flags $false$, ist es nicht mehr möglich, diesen Algorithmus aufzurufen. Dieser Sachverhalt wird dem Benutzer durch ein Pop-Up Fenster verdeutlicht, wenn er dennoch versucht, bspw. eine Extrahierung der Straßenflächen nach einer Vektorisierung durchzuführen.

Bevor der Benutzer eine Straßenkarte vektorisiert, hat er in einem Konfigurationsmenü die Möglichkeit, bestimmte Attribute für die einzelnen Algorithmen einzustellen, wie z.B. die Anzahl der Wiederholungen der Erosion oder Toleranzwerte für das Verschmelzen der Knoten. Weitere Einstellungsmöglichkeiten sind folgender Abbil-

dung zu entnehmen.

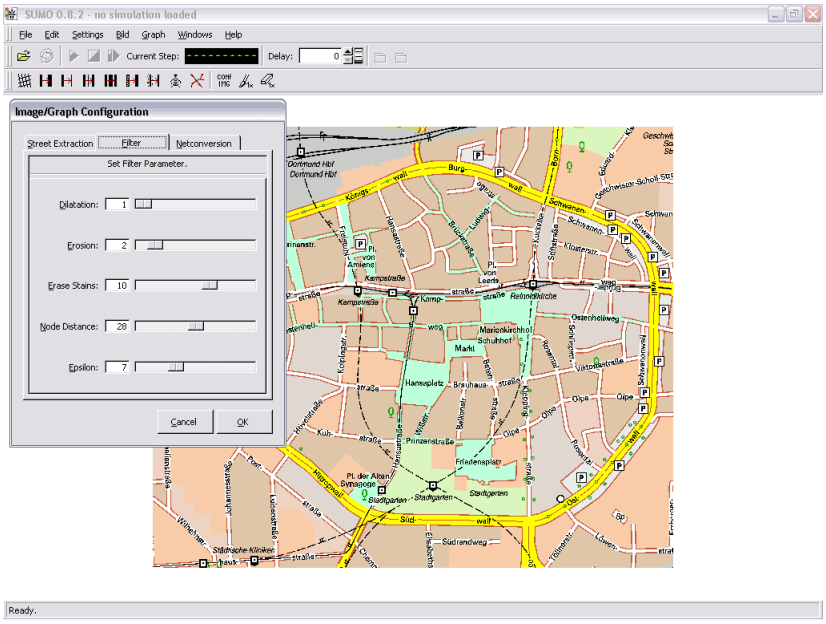


Abbildung 2.12: Konfigurationsmenü für Filter

2.4 Vom Graphen zur Verkehrssimulation

Um nun erfolgreich eine authentische Verkehrssimulation durchführen zu können, haben wir uns im Verlauf der PG dazu entschlossen, unsere bis dato geleistete Arbeit in das open-source-Projekt *SUMO* zu integrieren.

2.4.1 Das open-source Projekt SUMO

SUMO ist ein Akronym und steht für Simulation of Urban Mobility. SUMO hat es sich mit seinen Entwicklern zur Aufgabe gemacht, eine portable, mikroskopische Simulationsumgebung für den Straßenverkehr zu entwerfen, um auf sehr großen Straßennetzen simulieren zu können.

Im Wesentlichen wurde SUMO 2000 von den Mitarbeitern von (ZAIK) und vom *Institute of Transport Research* im *German Aerospace Centre* ins Leben gerufen. Die Hauptaufgaben bestehen in der Optimierung von Ampelkreuzungen, Analyse des Fahrzeugverhaltens und Routingberechnungen.

Zu den wichtigsten Programmteilen gehören:

- command-line simulation (SUMO)
- simulation with a graphical user interface (GUI)

- network builder/converter (NETCONVERT)
- network builder/generator(NETGEN)
- OD-matrix to trip definitions converter (OD2TRIPS)
- a router using a dynamic user assignement approach (DUA-ROUTER)
- a router using junction ratio descriptions (JTR-ROUTER)

Zur Zeit liegt SUMO in der Version 0.8.0.2 vor.

2.4.2 Aus dem Graph wird ein SUMO-Straßennetz

SUMO erzeugt Straßennetze auf Basis von Graphen, deren Informationen in XML-Dateien gespeichert sind. Es werden im Wesentlichen drei XML-Dateien zur Generierung benötigt. Zum einen zwei ähnliche Dateien, in denen Knoten- bzw. Kanteninformationen gespeichert sind und eine Datei in der die Routen der virtuellen Vekehrsteilnehmer enthält. Zur Laufzeit der Simulation ist es nicht mehr möglich, aktiv in die Simulation einzugreifen. Für das Parsen der XML-Dateien wird *XERCES* benötigt. Es war also nötig, in der Klasse `Graph` eine Erweiterung dahingehend vorzunehmen, so dass die oben genannten XML-Dateien im gerechten Format erzeugt werden. Der folgende Auszug aus den XML-Dateien *node.sumo.xml* (siehe Abbildung 2.13) und *edge.sumo.xml* (siehe Abbildung 2.14 skizziert deren Struktur.

```
<nodes>
 <node id="0" x="0.0" y="220.0" type="priority" />
 <node id="1" x="61.0" y="234.0" type="priority" />
 <node id="2" x="98.0" y="232.0" type="priority" />
 .
 .
 <node id="n" ... />
</nodes>
```

Abbildung 2.13: Beispieldatei: Knoten

Sind die XML-Dateien korrekt formatiert, sieht das unserem Straßengraphen äquivalente Sumonetz wie folgt aus.

2.4.3 Nachbearbeitung des SUMO-Straßennetzes

Nach der erfolgreichen Vektorisierung, dem Vereinfachen des Straßengraphen, dem Export der für SUMO erforderlichen XML-Dateien muss nun noch das Straßennetz

```
<edges>
 <edge id="0" fromnode="0" tonode="1" priority="78"
                      nolanes="1" speed="50.000" />
 <edge id="1" fromnode="0" tonode="2" priority="78"
                      nolanes="1" speed="50.000" />
 <edge id="2" fromnode="2" tonode="n" priority="78"
                      nolanes="1" speed="50.000" />
 .
 .
 <edge id="n" ... />
</edge>
```

Abbildung 2.14: Beispieldatei: Kanten

als solches bearbeitet werden. Standardmäßig entspricht eine Spur einer Kante und die zulässige Höchstgeschwindigkeit liegt bei 50 km/h. Um eine realitätsnahe Simulation durchzuführen, sollte das Straßennetz eine gute Abbildung der Wirklichkeit darstellen. Demzufolge haben nicht alle Straßen zwei Spuren, was dem Standardwert der Straßenerzeugung entspricht. Es ist wünschenswert, auf der SUMO-Oberfläche direkt eine Kante zu markieren und Attribute zu verändern. Wie bekommt man jedoch Informationen darüber, wie viele Spuren jede einzelne Straße des betrachteten Kartenausschnitts besitzt? Abhilfe versprach die folgende Seite im Internet. Hier sind Satellitenfotos von allen Städten des Ruhrgebietes zu finden. Es ist möglich, diese Fots heraus- bzw. heranzuzoomen, so dass man die Anzahl der Spuren auf einer beliebigen Straße in guter Qualität erkennen kann. Insbesondere ließ sich auf diese Art und Weise herausfinden, welche Kreuzungen Ampelkreuzungen sind. Es ist also möglich, die einzelnen Kanten per Hand zu annotieren und das Straßennetz real darzustellen.

Problemstellung Die visuelle Darstellung des Straßengraphen in SUMO ist losgelöst von der Datenstruktur, die den originären Graphen repräsentiert. Daraus ergibt sich die folgende Problemstellung. Verändert man das Straßennetz in der Visualisierung, beispielsweise mit den oben genannten Methoden, so verändert sich auch tatsächlich nur die Darstellung. Die Datenstruktur bleibt davon unberührt. Die Datenstruktur ist von Seiten der Entwickler nur auf Effizienz ausgelegt und nicht auf Benutzerfreundlichkeit. Für uns als Projektgruppenteilnehmer war es schwierig, durch diese Strukuren Erkenntnisse zu gewinnen. Daniel Krajzcewicz, einer der Entwickler von SUMO, hat uns seine Hilfe angeboten und hat die Datenstruktur soweit vereinfacht, dass es von dort an möglich war, den SUMO-Graphen zu manipulieren.

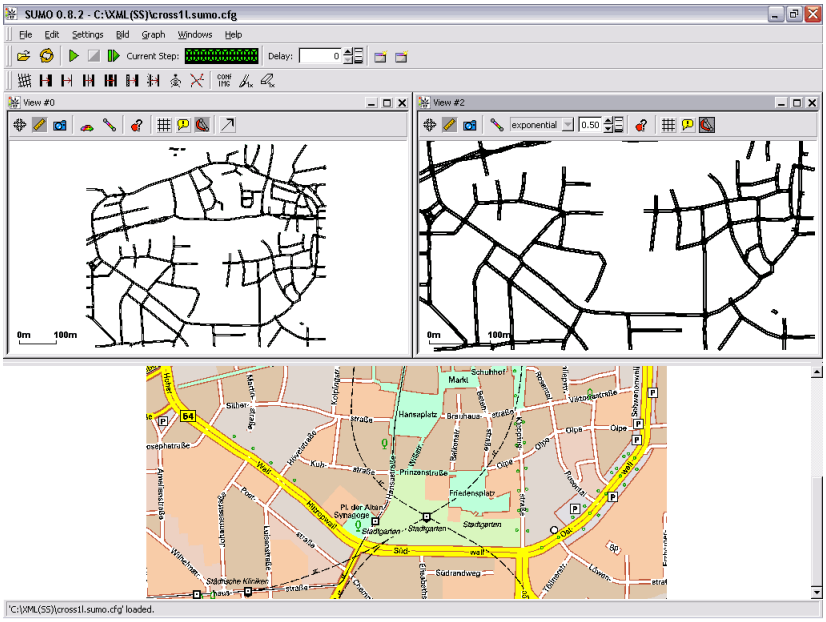


Abbildung 2.15: Das SUMO-Straßennetz (o. l.) insgesamt und ausschnittweise (o. r.) im Vergleich zur Ursprungskarte (u.)

2.4.4 Der Verkehr fließt

Wie bereits weiter oben angesprochen, werden die Wege, die ein Fahrzeug im Rahmen der Simulation zurücklegt, zuvor in einer XML-Datei definiert. Es ist also zur Simulationslaufzeit nicht mehr möglich, diese Daten zu verändern. Zusammen mit der Routingdatei und den Dateien für die Knoten und Kanteninformationen wird in der Konsole mit dem SUMO-Unterprogramm `NetConvert` eine Datei *meineSimulation.sumo.cfg* erzeugt, die sich direkt mit dem Menüeintrag *open* in Sumo öffnen lässt.

Das Unterprogramm `Duarouter` hilft bei der Generierung der einzelnen Routen und somit bei der Erstellung der Routingdatei. In der Konsole lässt sich angeben, wie viele Autos pro Zeiteinheit (d.h. pro Simulationsschritt) erzeugt werden sollen. Es werden randomisiert Start- und Endkanten ausgewählt und ein Weg zwischen ihnen berechnet. All diese Informationen werden schließlich in die Routingdatei geschrieben.

2.4.5 Erweiterung der Klasse vehicle von Sumo

Damit wir dem Projektgruppenziel gerecht werden, müssen GPS-Spurdaten aus der laufenden Verkehrssimulation gewonnen werden. Allerdings sollten sie nunmehr nicht statischer Natur sein, sondern dynamisch (beispielsweise sollte anhand des Spurdatums ein Abbremsverhalten des Fahrzeuges erkennbar sein). Hierzu sind einige Manipulationen der Klasse `MSVehicle` von SUMO nötig. Damit wir eine korrekte Ausgabe der GPS-Koordinaten erhalten, müssen wieder die Gauß-Krüger-Koordinaten kon-

vertiert werden. Allerdings werden die Fahrzeugpositionen nicht mit den Koordinaten des Koordinatensystems des Canvas bestimmt, sondern mit den Kanten, auf denen sich das Fahrzeug befindet. Aufgrund dessen ist es allerdings nicht möglich, GPS-Spurdaten zu erzeugen. Es ist notwendig, die Klasse `MSVehicle` von SUMO derart zu erweitern, dass Fahrzeuge über ihre absolute Position lokalisiert werden.

2.4.6 Export der GPS-Traces

Wie bereits in den einleitenden Kapiteln diskutiert, werden an die von uns genertieren Daten spezielle Anforderungen gestellt. So muss neben der Authentizität der einzelnen Datensätze auch das gültige Format eingehalten werden. Die Spurdaten, welche an die Gruppe *MapGeneration* weitergeleitet wurden, sehen folgendermaßen aus:

```
$GPRMC,174512,A,5149.388979,N,00738.045083,E,,,05122004,,
$GPGGA,174512,5149.388979,N,00738.045083,E,,,,0.0,,,,,
```

Abbildung 2.16: NMEA Beispiel

Interpretation Die GPS-Spurdaten sollen in einem ASCII-File gespeichert werden. Das übernimmt eine Methode der Klasse `MSVehicle`. Der Schreibvorgang geschieht bei sogenannten *log-Points*. In der Realität sind dies die Zeitpunkte, bei denen ein GPS-Empfänger Verbindungen zum Satelliten aufnimmt und so letztlich den Trace erzeugt. In unserer Simulation kann man den zeitlichen Abstand der log-Points selber definieren (jede Sekunde, alle fünf Sekunden, ...). Jedes Fahrzeug schreibt während seiner Existenz auf der Karte beispielsweise alle fünf Sekunden seine augenblickliche (GPS-) Position in das Textfile. Ist das Fahrzeug an seinem Ziel angekommen, erhalten wir einen gültigen NMEA-Datensatz im GPRMC- bzw. GPGGA-Format.

2.5 Ausblick

2.5.1 Vergleichbare Arbeiten

TU München Mit der Erkennung von Straßen aus Luft- und Satellitenbildern beschäftigte sich auch die TU München (siehe [21]). In *Semantic Objects and Context for Road Extraction* und den Vor- sowie Folgearbeiten werden verschiedene Ansätze für eine Straßenerkennung beschrieben. Als Hauptpunkt dieser Arbeiten kann der Multi-Resolution-Ansatz genannt werden. Dabei werden Straßen mit Hilfe von Bildern verschiedener Auflösungen des gleichen Ausschnittes verwendet. In hoher Auflösung werden Straßen z. B. mittels paralleler Linien erkannt, während bei tiefer Auflösung Straßen nur als dünne Linien auftauchen und erkannt werden. Des Weiteren

werden auch Einflüsse der Umgebung auf die Erkennung beschrieben (*Context based road extraction)*. Diese teilweise sehr komplexen Algorithmen liefern sehr gute Resultate, sind jedoch nur bedingt auf gedruckte Karten übertragbar. Siehe auch `http://www9.informatik.tu-muenchen.de/projects/roads.html`

AIS Des Weiteren ist im Rahmen einer Diplomarbeit an der Philosophischen und Naturwissenschaftlichen Universität Bern im Jahr 2001 erschienen (siehe [2]). Im Wesentlichen beschäftigt sich hier Arik Dasen mit der Extraktion von Straßen aus gedruckten Karten. Hierbei steht keine anschließende Verkehrsplanung oder Verkehrssimulation im Vordergrund. Vielmehr konzentriert sich die Arbeit auf die Erkennung von Kartensymbolen wie z.B. Parkplätze, Legenden oder anderen relevanten Symbolen. Siehe auch `www.carve.ch/uni/diplomarbeit/diplom_dasen.pdf`

Sumo bei INVENT An diesem Projekt beteiligen sich viele namhafte deutsche Unternehmen(Opel, Audi , BMW, MAN ...), um insbesondere Entwicklungen in der Verkehrsflussoptimierung und in der Vekehrssicherheit voranzutreiben. Dabei stehen drei zentrale Projekte im Vordergrund, die sich im Weiteren in acht Teilprojekte untergliedern:

Hauptprojekte

- Fahrerassistenz, aktive Sicherheit
- Verkehrsmanagement 2010
- Verkehrsmanagement in Transport und Logistik

2.5.2 Blick in die Zukunft

Da das uns zur Verfügung stehende Kartenmaterial unverrauscht ist, war die Implementierung von Filtern die das Kartenmaterial entrauschen nicht nötig. Um zum Beispiel auch eingescannte Pläne bearbeiten zu können wäre die Implementierung von Rauschfiltern eine sinnvolle Ergänzung.

Gerade in Zusammenarbeit mit dem Projekt SUMO ergeben sich interessante Erweiterungsmöglichkeiten. In SUMO gibt es mehrere Möglichkeiten Routeninformationen in ein gegebenes Straßennetz zu integrieren. Einerseits können Routen per Zufall erzeugt, und auf den zugehörigen Straßennetzen simuliert werden, was natürlich nicht zu realistischen Simulationen führt. Andere Möglichkeiten sind die Simulationsumgebungen entweder aus geläufigen Formaten wie Visum, Vissim oder Artemis, in das SUMO-Format zu konvertieren, oder Routen in XML-Dateien per Texteingabe auf recht mühsame Art und Weise zu definieren. Es existiert zur Zeit noch kein grafisches Werkzeug, mit welchem Routen auf Straßennetzen bequem eingegeben werden können. Auch diese Lösung dieser Aufgabe könnte eine weitere sinnvolle Ergänzung des Systems darstellen.

Algorithmus 1 Skelettierung

Eingabe:
*R*asterbild I mit Breite b und Höhe h
Ausgabe:
*S*kelett von I

repeat
{Im ersten von zwei Durchläufen des Bildes I werden die schwarzen Pixel dahingehend untersucht, ob sie entweder auf der Ost- oder Südgrenze liegen, oder ob es sich um Nordwest-Eckpunkte handelt. In diesen Fällen werden sie zwecks späterer Löschung markiert.}
for $i = 1$ **TO** b **do**
for $j = 1$ **TO** h **do**
if $p(i,j)$ =SCHWARZ **AND**
{Bedingung 1}
$2 \leq$ *zähleSchwarzeNachbarn(*$p(i,j)$*)* ≤ 6 **AND**
{Bedingung 2}
zähle0/1Übergänge($p(i,j)$*)*=1 **AND**
{Bedingung 3.1}
($p(i,j-1)$ =WEIß **OR** $p(i+1,j)$ =WEIß **OR** $p(i,j+1)$ =WEIß) **AND**
{Bedingung 4.1}
($p(i+1,j)$ =WEIß **OR** $p(i,j+1)$ =WEIß **OR** $p(i-1,j)$ =WEIß) **then**
markiere $p(i,j)$
end if
end for
end for
setze alle markierten Pixel auf weiß, und lösche die Markierungen
{Im zweiten Durchlauf werden die schwarzen Pixel dahingehend untersucht, ob sie entweder auf der Nord- oder Westgrenze liegen, oder ob es sich um Südost-Eckpunkte handelt. Auch in diesen Fällen werden sie zur späteren Löschung markiert.}
for $i = 1$ **TO** b **do**
for $j = 1$ **TO** h **do**
if $p(i,j)$ =SCHWARZ **AND**
{Bedingung 1}
$2 \leq$ *zähleSchwarzeNachbarn(*$p(i,j)$*)* ≤ 6 **AND**
{Bedingung 2}
zähle0/1Übergänge($p(i,j)$*)*=1 **AND**
{Bedingung 3.2}
($p(i,j-1)$ =WEIß **OR** $p(i+1,j)$ =WEIß **OR** $p(i-1,j)$ =WEIß) **AND**
{Bedingung 4.2}
($p(i,j-1)$ =WEIß **OR** $p(i,j+1)$ =WEIß **OR** $p(i-1,j)$ =WEIß) **then**
markiere $p(i,j)$
end if
end for
end for
setze alle markierten Pixel auf weiß, und lösche die Markierungen
until mindesten ein Pixel wurde markiert

Kapitel 3

Das MapGeneration-Modul

- Mohamed Bettahi • René Brüntrup • Seung-Jun Hong • Björn Scholz -

Abbildung 3.1: Schrittweiser Aufbau eines Kartenteils

Bisherige elektronische Karten sind in ihrer Herstellung und Aktualisierung aufwendig und damit teuer. Es gibt jedoch mittlerweile eine große Anzahl relativ günstiger Geräte, mit denen sich GPS-Daten aufzeichnen lassen. Dazu gehören u.a. spezielle GPS-Geräte, die nur dem Zweck dienen, die aktuelle Position anzuzeigen und aufzuzeichnen (z. B. [10, 27]), PDAs mit extern angeschlossenem GPS-Empfänger und auch die immer häufiger werdenden PDAs mit integriertem GPS-Empfangsteil. Damit liegt die Idee nahe, aus einer großen Menge gesammelter GPS-Daten automatisch eine Karte zu errechnen bzw. eine bestehende Karte zu aktualisieren.

Das MapGeneration-Projekt verfolgt genau diese Idee. Ein Server nimmt über das Internet eine große Menge an GPS-Spuren entgegen und errechnet aus diesen in einer

internen Datenbank eine Karte. Obwohl die Idee grundsätzlich auf unterschiedliche Arten von Wegekarten anwendbar ist, konzentrieren wir uns zunächst auf Straßenkarten.

Unser Ansatz ist sowohl zum Erstellen digitaler Karten von bisher nicht erfaßten Regionen, als auch zum Aktualisieren bestehender Karten geeignet. Neben der einfachen und kostengünstigen Erstellung und Aktualisierung bietet der Einsatz von GPS-Daten den Vorteil, dass detailierte Fahrzeitinformationen zur Verfügung stehen. Heutige Karten (z. B. [12, 15, 17]) bieten für gewöhnlich lediglich eine Unterscheidung nach Straßentypen (Autobahn, Landstraße, usw.), die für eine Fahrzeitabschätzung jedoch nicht ausreichend sind. Speichert man zusätzlich die Fahrzeitinformationen nach Wochentagen und Uhrzeiten getrennt, so lassen sich wesentlich genauere Fahrzeitschätzungen abgeben. Eine Fahrt durch das Kamener Kreuz ist z.B. nachts wesentlich schneller möglich als an einem Werktag Nachmittag um 17:00 Uhr.

Der folgende Abschnitt *MapGenerator* beschreibt den Server, inklusive dem darin verwendeten Algorithmus zur Erstellung der Straßenkarte. Im Kapitel *MapGenerator Gui* wird anschließend die Administrator-Oberfläche beschrieben. Im Ausblick finden sich zuletzt weitere Ideen und Möglichkeiten, die bisher nicht realisiert wurden.

3.1 MapGenerator

Der MapGenerator nimmt als Server GPS-Spuren entgegen und integriert die daraus gewonnenen Informationen in eine interne Straßenkarte. Da der MapGenerator im normalen Betrieb keine Interaktivität erfordert ist er als Konsolenprogramm ausgelegt.

Als Eingangsdaten verwendet der MapGenerator GPS-Spuren. Eine GPS-Spur besteht aus einer Reihe von Koordinaten, die jeweils mindestens durch Längen-, Breitengrad und Zeit gegeben sind. Falls in den Eingangsdaten Höheninformationen vorhanden sind, werden diese auch berücksichtigt. Als Format für die Spuren wird zur Zeit nur NMEA 0183 (siehe [22]) unterstützt. NMEA 0183 ist ein Standard der *National Marine Electronics Association* (siehe [23]), der von praktisch allen GPS-Geräten unterstützt wird.

Aus diesen GPS-Spuren soll eine Straßenkarte erstellt werden. Der MapGenerator verwendet als Straßenkarte einen gerichteten Graphen (siehe Abbildung 3.2). Dabei entspricht jede Kante einer Fahrtmöglichkeit. Eine gewöhnliche Straße besteht aus zwei Kanten, einer für jede Fahrtrichtung, Einbahnstraßen naheliegenderweise aus nur einer Kante. Mehrspurige Straßen bestehen ebenfalls nur aus zwei Kanten, da die Ungenauigkeit des GPS-Signals keine direkte Unterscheidung mehrerer Spuren erlaubt. Kreuzungsknoten sind alle Knoten, die über mehr als eine eingehende und/oder ausgehende Kante verfügen.

Für eine detailierte Straßenkarte reichen diese Informationen aber noch nicht aus. Lange kurvige Straßen ohne Abbiegemöglichkeiten würden durch eine gerade Linie dargestellt. Daher werden zu jeder Kante Detailinformationen gespeichert. Diese enthalten eine Reihe von GPS-Punkten, die den Straßenverlauf genauer approximieren.

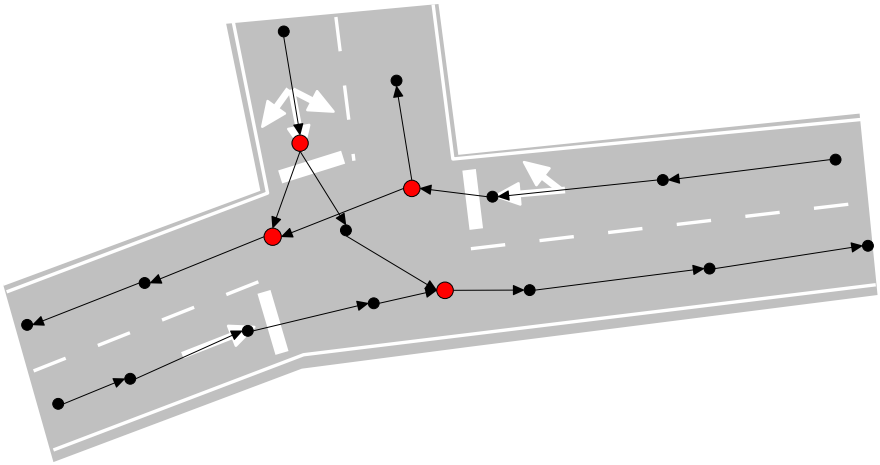


Abbildung 3.2: Kartenformat der Karte in der Datenbank

Zu jedem einzelnen dieser kürzeren Stücke wird außerdem die durchschnittliche Fahrzeit gespeichert.

Zur Verwaltung der Daten wurde die Welt in Kacheln konstanter Größe (bezüglich des geographischen Koordinatensystems) aufgeteilt. Eine Kachel ist $1/100° \cdot 1/100°$ groß, dies entspricht in der Gegend um Dortmund einer Grösse von ca. 550 m $\cdot$ 1100 m. Innerhalb der Kacheln werden die Knoten (in unserem Falle sind das GPS-Punkte) gespeichert. Die Kanten werden einzeln und separat von den Knoten als Liste von Knoten-IDs bespeichert. Durch eine zusätzliche Zuordnung von Kanten-IDs zu den Knoten, lassen sich zu Kanten und Knoten immer die passenden Gegenstücke finden. Die Kachelung hat ihre Vorteile, u. a. in der einfachen Parallelisierung (siehe unten und 3.1.3) und in der Suche nach dem besten Clusterknoten (siehe 3.1.4).

Um die Daten zu speichern, verwendet der MapGenerator eine Datenbank, die über ODBC (siehe [25]) angebunden wird. ODBC hat den Vorteil, dass wir uns während der Implementationsphase nicht auf eine Datenbank festlegen müssen. Momentan wird eine MySQL-Datenbank verwendet, diese läßt sich jedoch leicht und ohne dass das Projekt neu kompiliert werden muss, gegen z. B. eine PostgreSQL-Datenbank austauschen.

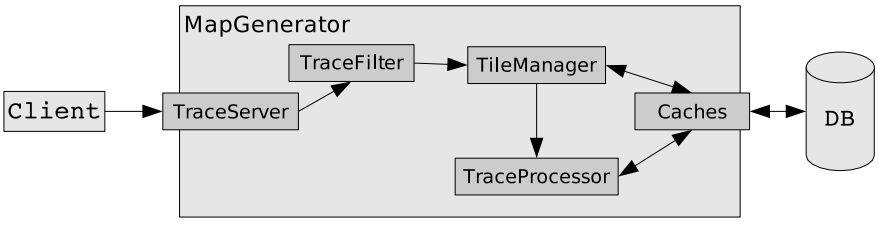


Abbildung 3.3: Datenfluss vom Eingang der Daten bis zur Datenbank

Um nun die Spuren in die Karte zu integrieren, werden diese in mehreren unabhän-

gigen Komponenten, ähnlich einer Prozessorpipeline, Schritt für Schritt verarbeitet. In Abbildung 3.3 sind die wichtigesten Komponenten und der Datenfluss zwischen ihnen dargestellt. Im Folgenden wollen wir der Reihe nach kurz auf die Komponenten eingehen. Detailiertere Informationen folgen dann in den anschließenden Abschnitten.

Der TraceServer nimmt die Daten über eine TCP/IP-Verbindung (siehe [6]) entgegen. Es wird kein weiteres Protokoll verwendet, so dass die NMEA-Daten mit einem beliebigen Programm (z. B. netcat) an den Server geschickt werden können. Nachdem eine GPS-Spur empfangen wurde, wird sie an den TraceFilter übergeben.

Der TraceFilter wendet mehrere Filter auf die Spuren an. Dabei sollen Fehler, die in GPS-Daten typischerweise auftreten können, soweit wie möglich erkannt und korrigiert werden. Unbrauchbare Spuren oder Teilspuren werden verworfen. Alle anderen Spuren werden an den TileManager übergeben.

Der TileManager ist für mehr Dinge zuständig, als sein Name vermuten läßt. Zum einen organisiert er das Starten der TraceProcessoren. Jeder Trace benötigt eine bestimmte Menge von Kacheln für seine Abarbeitung. Für die Zeit, in der ein Trace bearbeitet wird, sind diese Kacheln gesperrt. Traces, die unabhängig voneinander bearbeitet werden können, weil sie auf unterschiedlichen Kartenteilen liegen, können parallel gestartet werden. Außerdem ist der Tilemanager noch für eine Reihe von Operationen zuständig, bei denen mehrere TraceProcessoren synchronisiert werden müssen (siehe 3.1.3).

Der MapGenerator ist darauf ausgelegt eine große Anzahl von Traces sehr schnell zu verarbeiten. Jede Programmkomponente ist als unabhängiger Thread realisiert, so dass hierdurch bereits, ähnlich einer Prozessorpipeline mehrere Spuren parallel bearbeitet werden können. Da allerdings das Clustern wesentlich länger dauert als alle anderen Bearbeitungsschritte, ist dieser Vorgang durch das gleichzeitige Starten mehrerer TraceProcessoren ebenfalls parallelisierbar.

Die Parallelisierbarkeit hat zwei Vorteile. Zum einen können die verschiedenen Aufgaben auf verschiedene Prozessoren oder virtuelle Prozessoren (zum Beispiel beim Pentium IV) verteilt werden. Außerdem muss davon ausgegangen werden, dass der zugrundeliegende Datenspeicher – die Datenbank – im Vergleich zum Hauptprogramm sehr langsam arbeitet. Durch mehrere Prozesse auf einem Prozessor können die Latenzzeiten der Datenbank effektiv zur Berechnung verwendet werden.

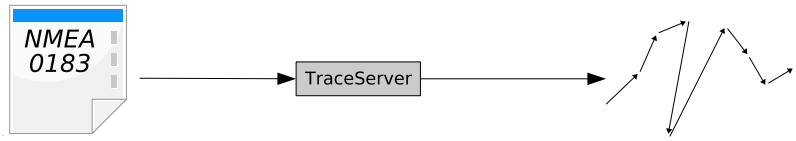
Wie bereits angesprochen ist die Datenbank ein Flaschenhals des Systems. Die Datenbanken werden daher nicht direkt, sondern über getrennte Caches für die verschiedenen Datenelemente – Kacheln und Kanten – angesprochen. Die Hauptalgorithmen brauchen nur sehr wenig Hauptspeicher, so dass in den Caches eine sehr große Anzahl von Elementen vorgehalten werden kann.

Wie besonders unter Linux üblich wird der MapGenerator über eine Konfigurationsdatei konfiguriert. Diese ist komplett in XML geschrieben und erlaubt die Konfiguration aller Parameter des Systems, vom Suchradius im Clusteralgorithmus, über die maximale Anzahl an TraceProcessoren bis zum TCP/IP-Port für den Dateneingang.

Da das Programm auf einen hohen Datendurchsatz ausgelegt ist, haben wir uns für C++ als Programmiersprache entschieden. Zumindest aus heutiger Sicht naheliegen-

derweise, verwenden wir Linux als Entwicklungsplattform. Alle verwendeten Bibliotheken und der MapGenerator selbst sind jedoch plattformunabhängig, so dass sich das System auf einer großen Anzahl von Plattformen benutzten lassen sollte.

3.1.1 TraceServer



Der TraceServer ist die erste Komponente der Verarbeitungskette. Er stellt sowas wie das Tor zur Außenwelt dar; hier werden über eine TCP/IP-Verbindung GPS-Spuren entgegen genommen. Diese GPS-Spuren, im Weiteren als *Trace* bezeichnet, müssen im NMEA 0183 Format vorliegen. NMEA 0183 ist ein durch die NMEA definiertes und ursprünglich zur Kommunikation zwischen Marinegeräten entwickeltes, heutzutage aber auch von gängigen GPS-Geräten unterstütztes, Übertragungsformat auf ASCII-Ebene.

Auf die TCP/IP-Verbindung wird kein weiteres Protokoll aufgesetzt, so dass zur Übertragung von Daten bereits ein kleines Programm wie *netcat*[1] ausreicht.

Technischer Ablauf Der Server öffnet einen Port, dessen Nummer in der Konfigurationsdatei eingestellt werden kann. Clients, die sich auf diesem Port verbinden, schicken ihre Daten und können durch das Beenden der Verbindung das Ende des Datenstroms signalisieren.

Die empfangenen Daten werden im Weiteren auf ihre syntaktische Korrekheit gemäß NMEA 0183 überprüft und die Informationen, wie z. B. Längengrad, Breitengrad und Zeit, der einzelnen GPS-Punkte extrahiert. Sollte die Extraktion der Daten – aus welchen Gründen auch immer – fehlschlagen, so wird dieser Trace verworfen. Andernfalls werden die gewonnen Informationen in einer entsprechenden Datenstruktur gespeichert und zur Weiterverarbeitung an den TraceFilter geschickt.

3.1.2 TraceFilter

Der TraceFilter erhält seine *Rohdaten* vom TraceServer. Die Rohdaten entsprechen exakt dem aufgezeichneten Trace. Durch Fehler des GPS-Empfängers während der Aufzeichnung, z. B. bedingt durch schlechten/fehlenden Empfang, können sich im Trace

[1] z. B. http://netcat.sourceforge.net/

falsche GPS-Punkte befinden. Beim verwendeten Gerät zur Aufzeichnung von Testdaten, waren z.B. sowohl vereinzelt als auch Sequenzen von GPS-Punkten mit den Koordinaten 0°/0° zu finden, obwohl wir leider keinen Urlaub im *Golf von Guinea* vor der afrikanischen Küste gemacht haben.

Der TraceFilter ist dafür zuständig Fehler zu erkennen und entsprechend zu behandeln. Zur Fehlererkennung werden momentan die folgenden Merkmale benutzt:

Es wird die Zeit zwischen zwei aufeinanderfolgenden GPS-Punkten betrachtet. Sollte diese zu groß sein, wird davon ausgegangen, dass es sich um zwei unterschiedliche Traces handelt und der ursprüngliche Trace wird aufgetrennt. Hier ergibt sich aber sofort ein Problem: Was geschieht, wenn ein längerer Tunnel, wie z.B der St.-Gotthard-Tunnel mit einer Länge von ca. 17 km (benötigte Zeit: ca. 15-25 min), durchfahren wird und somit verständlicherweise keine GPS-Daten vorliegen? Der Wert für den maximalen Zeitunterschied muss also wohlüberlegt sein und darf unter Umständen nicht der einzige Faktor zur Erkennung von Lücken sein.

Als weiteres Merkmal wird die durchschnittliche Geschwindigkeit, die zwischen zwei benachbarten GPS-Punkten erzielt wurde, ermittelt und gegen einen konfigurierbaren Wert, von z. B. 250 km/h, geprüft. Wird dieser überschritten, wird der GPS-Punkt gelöscht. Mit dieser Methode können auch Seqenzen von unmöglichen GPS-Punkten erkannt und gelöscht werden.

Hieran schließt sich der Test auf die mittlere Beschleunigung an. Dafür gilt ähnliches wie für den Geschwindigkeitstest: Es werden die Geschwindigkeiten zweier benachbarter Teilstücke und damit einhergehende Beschleunigung ermittelt und gegen einen frei konfigurierbaren Wert getestet. Ein sinnvoller Wert liegt im Bereich von ± 2 G $\simeq \pm 20$ m/s^2. Sollte dieser Wertebereich unter- bzw. überschritten werden, wird auch hier der verursachende GPS-Punkt gelöscht.

Veranschaulichen lässt sich dieser Sachverhalt an Abbildung 3.4. Hier liegt Punkt A erkennbar weit entfernt vom eigentlichen Trace. Durch den Test auf die Geschwindigkeit könnte dieser Punkt nicht als Ausreißer erkannt werden, da diese u. U. unter dem angesprochenen Schwellwert liegen kann. Betrachtet man hingegen die (positive wie negative) Beschleunigung, die benötigt wird um den Punkt A in der gegebenen Zeit zu erreichen bzw. zu verlassen, so ist die Chance größer, diesen Punkt als Ausreißer zu entlarven.

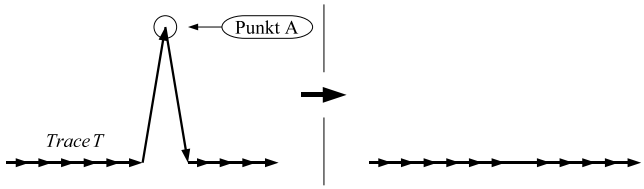


Abbildung 3.4: Einzelner Ausreißer

Da der Filter jeden Punkt einzeln betrachtet, ist es mit dem gleichen Mechanismus möglich Sequenzen von Ausreißern zu erkennen. Dieses wird anhand der Abbil-

dung 3.5 erklärt. Hier liegen die Punkte A - E augenscheinlich zu weit vom eigentlichen Trace entfernt. Punkt A scheitert am Geschwindigkeits- und/oder Beschleunigungstest und wird gelöscht. Um nun zu Punkt B zu gelangen, ist eine noch größere Geschwindigkeit/Beschleunigung nötig. Also wird er auch gelöscht. Dieses Verfahren wird bis zum Punkt E einschließlich fortgesetzt. Das Löschen dieses Punktes ist in der Abbildung aus Platz- und Redundanzgründen (siehe dazu Abbildung 3.4) nicht mehr aufgenommen worden. Nach dem Löschen einer Reihe von Ausreißern kann es erneut notwendig werden den Trace aufgrund einer zu großen Lücke aufzutrennen.

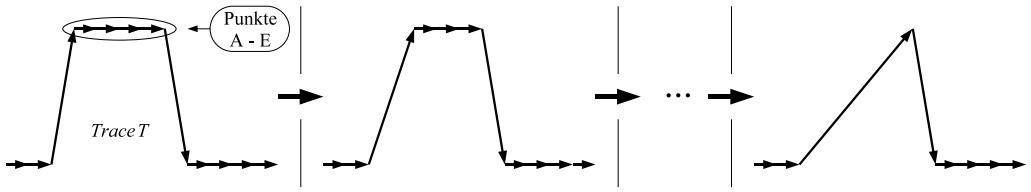


Abbildung 3.5: Sequenz von Ausreißern

Durch diese Massnahmen lassen sich die gröbsten Fehler, große Lücken und Ausreisser, behandelt. Für weitere Fehler, die sich z. B. im Grenzbereich der bereits angesprochen Merkmale befinden, können durch den modularen Aufbau sehr einfach weitere Filter eingebaut werden. Zu nennen sind hier Filter, die z. B. die Winkeländerung in Verbindung mit der herrschenden Geschwindigkeit betrachten. Und falls weitere Eingabequellen vorhanden sind, wie ein Tachometer, Beschleunigungssensoren u. ä., so können diese Informationen mit Hilfe von Kalmannfiltern dazu dienen, den Trace zu verifizieren bzw. korrigieren.

Wenn der Trace vollständig gefiltert wurde, wird er an den TileManager übergeben, der über die weitere Verarbeitung entscheidet.

3.1.3 TileManager

Der TileManager erhält vom TraceFilter die nunmehr *sauberen* Traces, berechnet für jeden Trace die benötigten Kacheln und speichert sie zwischen. Zusätzlich verwaltet er alle momentan von den TraceProcessoren benutzten Kacheln. Erfüllt ein Trace, im weiteren als *der* Trace bezeichnet, gewisse Voraussetzungen, so kann dieser Trace weiterverarbeitet, d. h. in die bestehende Karte eingearbeitet werden. Voraussetzungen sind z. B. genügend freie Systemressourcen und dass alle von dem Trace benötigten Kacheln unbenutzt sind.

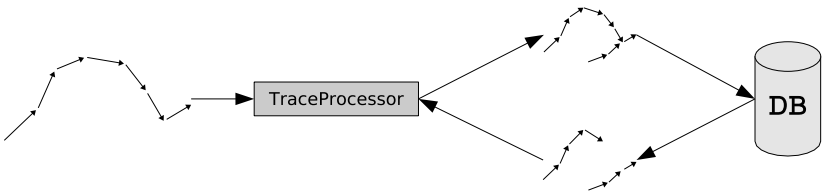
Hierzu werden die von dem Trace benötigten Kacheln exklusiv reserviert. Dadurch wird bereits vielen Konflikten mit anderen TraceProcessoren aus dem Weg gegangen.

Der Trace wird mittels des in 3.1.4 näher beschriebenen Clusteralgoritmus in die bestehende Karte eingearbeitet und danach werden alle benötigten Kacheln wieder freigegeben.

Das allgemeine Kachelkonzept und die Reservierung von Kacheln, erlaubt es, dass mehrere Traces durch TraceProcessoren parallel verarbeitet werden können. Eine gute Skalierbarkeit wird erreicht. Doch jeder Vorteil bringt zumeist Nachteile mit sich: In diesem Fall müssen Änderungen an Kanten, die sich auch über Kacheln jenseits der reservierten erstecken können, synchronisiert werden. Die initiale Reservierung stellt zwar sicher, dass sich der Trace nur innerhalb der reservierten Kacheln bewegt. Sie kann jedoch nicht garantieren, dass auch alle betrachtenen Kanten dies tun. Ohne Synchronization kann es zu Kollisionen zwischen den TraceProcessoren kommen, wenn diese auf gleiche Informationen lesend wie schreibend zugreifen.

Beispielhaft soll hier die Aufteilung einer Kante betrachtet werden: Bei der Entstehung einer neuen Kreuzung muss die ehemals durchgehende Kante aufgetrennt werden, da laut unseren Spezikationen einen Kante nur zwischen zwei Kreuzungen existieren darf. Dazu werden alle laufenden Traceprocessors vom TileManager benachrichtigt, dass die Kacheln, die von dieser Kante betroffen sind, zur Zeit nicht bearbeitet werden dürfen. Betroffene TraceProcessoren stellen daraufhin die Arbeit ein, der TileManager erfüllt seine Aufgabe (indem er jetzt die Kante aufspaltet), die TraceProcessoren werden über das Resultat (die neue ID der Kante) benachrichtigt und können dann weiterarbeiten. Dieser Prozess arbeitet mit wechelseitigem Ausschluss, d. h. nur ein TraceProcessor kann die entsprechende Methode im TileManager aufrufen.

3.1.4 TraceProcessor



Der TraceProcessor stellt die zentrale Verarbeitungseinheit dar. Hier wird ein nunmehr gefiltert (siehe 3.1.2) und durch den TileManager (siehe 3.1.3) ausgewählter Trace in die bereits in der Datenbank bestehende Karte integriert. Dies geschieht durch einen inkrementellen Clusteralgorithmus, der nun im Weiteren vorgestellt und veranschaulicht wird.

Eigenschaften des Clusteralgorithmus Es wird immer nur ein einziger Trace in die bestehende Karte, die auch leer seien darf, eingefügt. Im Gegensatz zum Ansatz von Schrödl (siehe [26]) wird also keine initiale Karte benötigt. Ausserdem ermöglicht dieses inkrementelle Aufbauen der Karte eine leichte Erweiterbarkeit der Karte und zum anderen hält es die Menge an möglichen Fällen überschaubar. Dies macht den Clusteralgorithmus einfacher, verständlicher und universell.

Durch den gewählten Kachelansatz (siehe 3.1) erhält man eine gewisse Lokalität. Dies hat an mehreren Stellen einen Vorteil. Hier sind u. a. die Suche nach benachbarten Knoten zu nennen. Die kleinen Kacheln garantieren, dass nur eine kleine Menge von Knoten untersucht werden muss.

Der Clusteralgorithmus führt die Änderungen, die von ihm ausgeführt werden, in einer speziellen Datenstruktur mit. Dieser sogenannte TraceLog wird benötigt, um das korrekte Arbeiten des Algorithmus zu verifizieren. Hierfür wurde ein GUI-Element entwickelt, das den TraceLog darstellen kann (siehe 3.2.2).

Benötigte Funktionen Der eigentlichen Clusteralgorithmus verwendet einige Funktionen, die kurz vorgestellt werden sollen. Wo möglich, wird die Motivation für die Funktion bereits hier angegeben. Ansonsten wird im Zuge des Clusteralgorithmus näher darauf eingegangen.

`BesterClusterknoten` Diese Funktion wird in jeder Iteration des Clusteralgorithmus benutzt, um, wie der Name bereits suggeriert, einen besten Clusterknoten zu suchen.

Was zeichnet nun diesen Clusterknoten aus? Zum einen muss sich dieser in einer definierten Umgebung zu dem gegebenen Knoten befinden (also z. B. in einem Radius $r \leq 30$ m) und die Orientierung darf sich nicht zu stark unterscheiden (z. B. max. Winkeldifferenz $|\alpha_{max}| \leq 20°$). Welche konkreten Werte für diese Eigenschaften benutzt und wie diese gewichtet werden, wurde auf Testdaten versucht zu ermitteln. Hier liegt jedoch noch eine Optimierungsmöglichkeit. Ein Beispiel wird in Abbildung 3.6 gegeben.

Um eine Laufzeit für diese Funktion zu ermitteln, muss die Implementierung näher betrachtet werden: Bislang werden bis zu vier Kacheln vollständig mittels brute-force durchsucht, d. h. jeder Knoten wird explizit betrachtet (vier Kacheln sind es dann, wenn der gegebene Knoten sehr nahe in der Ecke einer Kachel liegt). Der Test, ob ein Knoten als potenzieller bester Clusterknoten in Frage kommt, benötigt konstante Zeit. Da dieser Algorithmus als Variante einer Nächsten-Nachbarn-Suche angesehen werden kann, wird davon ausgegangen, dass ein Voronoi-Diagramm (siehe [9]) die Zeitkomplexität senkt. Es wird daher eine generelle Laufzeit von $O(t_{suche})$ angenommen.

`Verschmelzen` Jedesmal, wenn ein bester Clusterknoten gefunden wurde, muss dieser mit dem gegebene Tracepunkt verschmolzen werden. Dies geschieht gemäß den vergebenen Gewichten zu den Knoten. Diese Operation läßt sich in konstanter Zeit durchführen.

`Trennen` Eine in der Karte vorhandene Kante wird an einem gegebenen Knoten aufgetrennt. Sollte dieser Knoten eine Start- oder Endknoten sein, bleibt die Kante unangetastet. Ansonsten entstehen zwei neue Kanten. Aufgrund der benutzten Datenstruktur muss den Knoten der abgetrennten Kante die neue ID

dieser Kante mitgeteilt werden. Wie sich dies auf die Laufzeit auswirkt, wird im Abschnitt *Zeitkomplexität* erläutert.

`Verbinden` Diese Operation verbindet zwei Knoten. Hierzu wird entweder eine bestehende Kante um den angegeben Knoten erweitert, zwei Kanten verbunden, indem die zweite Kante nach der ersten eingefügt wird, oder eine neue Kante mit den angegebenen Knoten erzeugt. Da bei dem Verbinden zweier Kanten ebenfalls eine Umnummerierung der Knoten stattfinden muss, wird auf diese Operation betreffend der Laufzeit ebenfalls im Abschnitt *Zeitkomplexität* weiter eingegangen.

`NächsterTracepunkt` Wenn die Bearbeitung des aktuellen Tracepunktes abgeschlossen ist und dieser in die Karte integriert wurde, muss ein neuer Punkt zur Verarbeitung ausgewählt werden. Hier hängt es nun vom Ergebnis der Bearbeitung diese aktuellen Tracepunktes ab, welcher Punkt ausgewählt wird. Siehe hierzu Algorithmus 2. Die Laufzeit ist offensichtlich $O(1)$.

Algorithmus 2 NächsterTracepunkt

Eingabe:
GPS-Spur T
aktueller Tracepunkt t_i
aktueller Knoten n_i
Ausgabe:
nächster Tracepunkt t_{i+1}

{$nachfolger(n_i)$ ermittelt den Nachfolger von n_i in $kante(n_i)$}
if $nachfolger(n_i)$ existiert **then**
  $d \leftarrow$ `Abstand`$(n_i, nachfolger(n_i))$
else
  $d \leftarrow C$ {definierte Entfernung, z. B. 30m}
end if
$t_{i+1} \leftarrow$ interpolierter Tracepunkt, der von t_i um d entfernt ist.
return t_{i+1}

Inkrementeller Clusteralgorithmus Der Pseudo-Code des benutzten Clusteralgorithmus ist in Algorithmus 3 angegeben. Im weiteren werden die Fälle des Algorithmus erklärt: Zuerst wird ein bester Clusterknoten c ermittelt (siehe Abbildung 3.6). Falls dieser vorhanden ist, wird durch Verschmelzen mit dem aktuellen Tracepunkt t_i ein neuer Knoten n_i erzeugt. Nun muß ermittelt werden, ob sich die Knoten n_{i-1} und n_i auf unterschiedlichen Kanten befinden. Sollte dies der Fall sein, müssen diese Kanten an den entsprechenden Stellen aufgetrennt werden und auf die Knoten die `Verbinden`-Funktion anwenden werden. Dieser Fall ist in Abbildung 3.7 dargestellt.

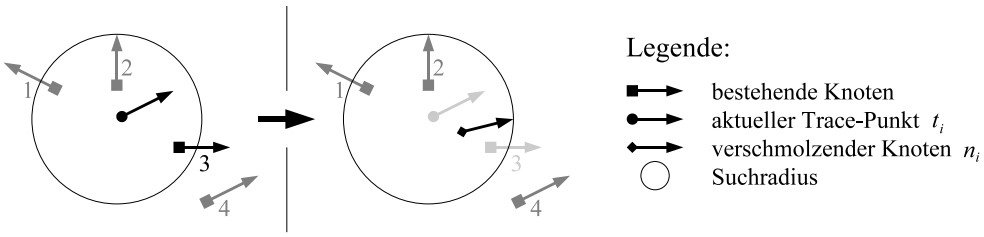


Abbildung 3.6: Suche nach dem besten Clusterknoten

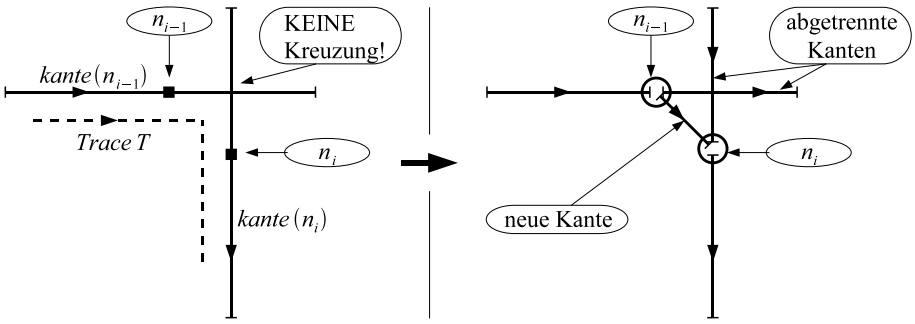


Abbildung 3.7: Aufspalten von zwei bestehenden Kanten

Falls kein Clusterknoten gefunden wurde, muß unterschieden werden, ob eine Kante zu n_{i-1} existiert: In diesem Fall wird diese Kante an der Stelle n_{i-1} aufgespalten und wiederum n_{i-1} und n_i verbunden. Zwei Bespiele hierzu sind in den Abbildungen 3.8 und 3.9 gegeben. Anderenfalls wird eine neue Kante zu n_i erstellt.

Die Bearbeitungsschritt ist damit abgeschlossen und es wird `NächsterTracepunkt` aufgerufen. Ausgehend von dessen Ergebnis werden entweder die Statusvariablen gesichert und in die nächste Iteration gegangen oder der Algorithmus beendet.

Zeitkomplexität Wenn man die Kanten als Adjazenzlisten verwaltet, benötigen alle Operationen abgesehen von der Clusterknoten-Suche und des Umnummerierens konstante Zeit.

Der aufmerksame Leser wird nun einwenden, dass die Kacheln und somit auch

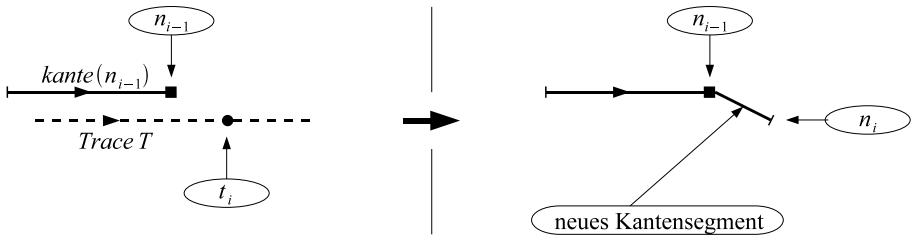


Abbildung 3.8: Anfügen eines Knoten an eine bestehende Kante

Algorithmus 3 Clusteralgorithmus

Eingabe:
GPS-Spur T

$i \leftarrow 0$
$t_i \leftarrow beginn(T)$
loop
  $c \leftarrow$ `BesterClusterknoten`(t_i)
  if c existiert **then**
    $n_i \leftarrow$ `Verschmelzen`(t_i, c)
    if $kante(n_{i-1}) \neq kante(n_i)$ **then**
      `Trennen`$(kante(n_{i-1}), n_{i-1})$
      `Trennen`$(kante(n_1), n_i)$
      `Verbinden`(n_{i-1}, n_i)
    end if
  else {c existiert nicht}
    $n_i \leftarrow t_i$
    if $kante(n_{i-1})$ existiert **then**
      `Trennen`$(kante(n_{i-1}), n_{i-1})$
      `Verbinden`(n_{i-1}, n_i)
    else
      `NeueKante`(n_i)
    end if
  end if
  $t_{i+1} \leftarrow$ `NächsterTracepunkt`(T, t_i, n_i)
  if t_{i+1} existiert **then**
    $t_i = t_{i+1}$
    $n_{i-1} = n_i$
    $i = i + 1$
  else {Ende des Traces erreicht}
    exit loop
  end if
end loop

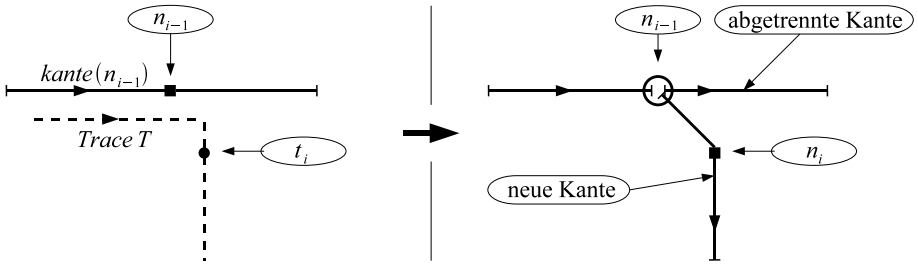


Abbildung 3.9: Aufspalten von einer bestehenden Kante

die Knoten sowie die Kanten aus einer Datenbank gelesen werden müssen, was einen Zugriff auf einen sekundären Speicher bedeutet. Als Konsequenz daraus würden alle im Abschnitt *Benötigte Funktionen* beschriebenen Operationen nicht mehr ausschlaggebend sein und der Algorithmus würde langsam werden. Daher wurde zum einen vor die Datenbank ein Cache geschaltet und zum anderen der TraceProcessor als Thread realisiert, so dass während des Zugriffs auf die Datenbank andere Arbeitsschritte ausgeführt werden können.

Betrachtet man nun die anfallende Datenmenge für Kanten und Knoten, so gehen wir davon aus, dass das komplette Straßennetz von Deutschland eine für heutige Server übliche Hauptspeichermenge nicht überschreitet. Somit behält die oben getroffene Aussage ihre Gültigkeit.

Wie viele Iterationen sind möglich bzw. nötig? Da die Funktion `NächsterTracepunkt` die gegebene GPS-Spur ausnahmslos vorwärts abarbeitet, werden maximal linear (in der Länge der Spur) viele Schritte ausgeführt. Wenn l die Länge der Spur ist, benötigt man $O(l)$ viele Iterationen.

Betrachten wir nun die Komplexität einer Iteration: Hierfür muß die Suche nach dem besten Clusterknoten und die bereits mehrfach erwähnte Umnummerierung von Knoten betrachtet werden. Die Clusterknoten-Suche wurde bereits mit $O(t_{suche})$ abgeschätzt. Um die Komplexität der Umnummerierung abschätzen zu können, müssen einige Vorbetrachtungen angestellt werden. Gegegeben seien GPS-Spuren mit einer Gesamtlänge von L. Nach der Verarbeitung befinden sich in einer inital leeren Karte $O(L)$ viele Knoten. Im worst-case kann nun an jedem Tracepunkt eine Umnummerierung (durch ein Aufspalten oder Verbinden) stattfinden. Da jedes Mal maximal allen vorhandenen Knoten eine neue ID zugewiesen werden muss, ergibt sich hierfür eine Laufzeit von $O(L^2)$.

Nun braucht man in diesem Fall keine worst-case Analyse anstellen, da die Eingangsdaten bekannt sind: Das real vorhandene Strassennetz. Dieses Netz zeichnet sich durch eine begrenzte und annähernd konstante Anzahl von Kreuzungen aus. Sind alle Kreuzungen einmal in der Karte verzeichnet, so wird keine Umnummerierung mehr stattfinden. Der Aufwand der für diese Kreuzungen betrieben wird, kann also als eine amortisierte Konstante angesehen werden. Das Verbinden von Kanten kann ebenfalls vernachlässigt werden.

Somit läßt sich die Zeitkomplexität des kompletten Algorithmus mit $O(L \cdot t_{suche})$ abschätzen. t_{suche} ist nicht von L, sondern nur von der Größe einer Kachel abhängig. Eine Kachel konstanter Größe kann aufgrund der Beschaffenheit dieses Algorithmus nur eine begrenzte, vorhersagbare Anzahl an Knoten aufnehmen. Da t_{suche} um Grössenordnungen kleiner ist als L, liegt hier annähernd ein Linearzeit-Algorithmus vor.

3.2 MapGeneration GUI

Im Laufe des Projektes hat sich gezeigt, dass zum Überwachen und Testen des Servers unbedingt eine grafische Benutzeroberfläche notwendig ist: Ohne grafische Oberfläche steht dem Server nur die Textausgabe auf der Konsole zur Verfügung. Mit einer Oberfläche hingegen können Statusinformationen, Verarbeitungsinformationen und auch der Inhalt der Datenbank, also die aktuelle Karte, übersichtlich dargestellt werden.

Für den täglichen Betrieb ist es wichtig, eventuell auftretende Probleme und Leistungsengpässe zu erkennen. Dazu müssen Fehler und grundsätzliche Statusinformationen, wie z. B. die Anzahl der aktuell arbeitenden TraceProcessoren oder die Anzahl noch auf Abarbeitung wartender Spuren, einsehbar sein. Diese Informationen lassen sich nicht sinnvoll durch Textausgaben auf der Konsole darstellen. Die grafische Oberfläche ermöglicht eine detailierte Darstellung aller dieser Meldungen und Statusvariablen des Servers.

In der Testphase waren jedoch zunächst die Kartendarstellungfunktionen besonders wichtig, die eine grafische Oberfläche bieten kann. Wie bereits im Kapitel 3.1.4 (TraceProcessor) gezeigt, ist die Berechnung der Straßen relativ kompliziert, so dass eine genaue Darstellung dieses Prozesses unumgänglich ist, um Fehler erkennen und verstehen zu können. Der TraceLogViewer zusammen mit den vom TraceProcessor gespeicherten Verarbeitungsinformationen ermöglicht dazu eine detailierte, schrittweise Betrachtung der Berechnungen.

Zum intensiveren Testen werden große Datenmengen an den Server geschickt, so dass sich bereits umfangreiche Kartenstücke ergeben. Um in diesen großen Bereichen problematische Stellen finden zu können, muss es eine Möglichkeit geben, die gesamte Karte darzustellen, wie sie aktuell in der Datenbank gespeichert ist. Diese Aufgabe löst der MapViewer, mit dem sich schnell und bequem die gesamte Erdkarte überblicken läßt, der aber auch eine Vergrößerung bis auf einzelne Kreuzungen erlaubt.

Die nächsten beiden Abschnitte beschäftigen sich im Detail mit dem TraceLogViewer und dem MapViewer.

3.2.1 MapViewer

Der MapViewer stellt die aktuelle, vom Server generierte Karte dar. Dazu verbindet er sich direkt mit der Datenbank des Servers. Da nur ein lesender Zugriff auf die Datenbank notwendig ist, kann der Viewer problemlos auch bei laufendem Server verwendet werden.

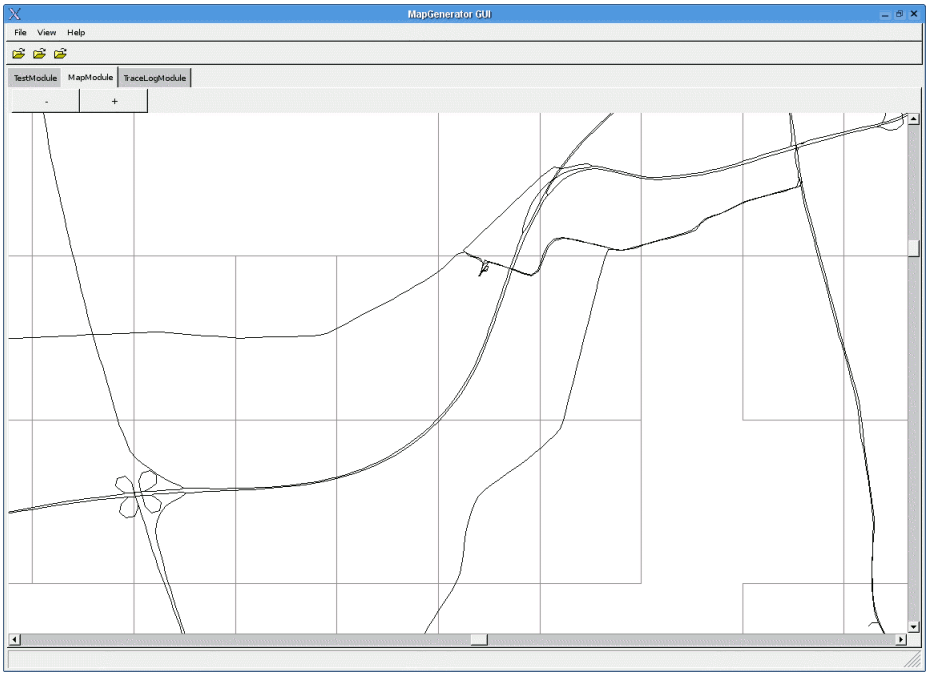


Abbildung 3.10: Hauptansicht des MapViewers

Abbildung 3.10 zeigt den MapViewer mit einem Kartenteil in mittlerer bis starker Vergrößerung. Zu sehen sind das Autobahnkreuz Münster-Süd und Teile Münsters (in der rechten oberen Ecke). Durch die Knöpfe am oberen Rand der Karte und über das Mausrad läßt sich die Vergrößerung von einer Sicht auf die gesamte Erdkarte bis hinunter auf einzelne Straßen regulieren.

Wie im Kapitel über den Server beschrieben, rechnet der Server nur mit GPS-Koordinaten. Um die Daten zweidimensional darzustellen, wird im MapViewer die weltweit funktionierende UTM-Transformation (siehe [4]) verwendet.

Dem aufmerksamen Leser wird aufgefallen sein, dass bei der Darstellung der gesamten Erdkarte $36.000 \cdot 18.000 = 648.000.000$ Kacheln zu berücksichtigen sind. Es ist vollkommen klar, dass es unmöglich ist, alle 648 Millionen Kacheln aus der Datenbank zu laden, und selbst das einzelne Abfragen der Verwendung aller Kacheln, ohne die Daten innerhalb der Kacheln zu laden, dürfte für einen durchschnittlich geduldigen Benutzer unzumutbar sein.

Daher lädt der MapViewer zu Beginn eine Liste aller verwendeten Kacheln. Wenn die Karte relativ dicht mit Straßen belegt ist, kann auch diese Operation eine größere Datenübertragung notwendig machen. Bei einer *willkürlich geschätzten* Nutzung von 1% der Kacheln ergibt sich z.B. eine Datenmenge von $6.480.000 \cdot 4$ Byte = 25.920.000 Byte (4 Byte für die ID der Kachel), das sind knapp 25 Megabyte. Bei einer schnellen Netzwerkverbindung – die für den Betrieb des Viewers sowieso notwendig ist – sollte diese Übertragung zumindest noch erträglich sein.

Die Liste der verwendeten Kacheln wird dann in zwei Bitfeldern unterschiedlicher

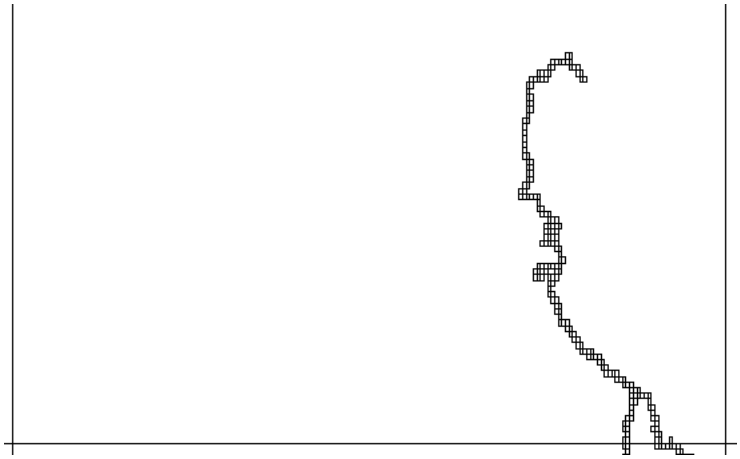


Abbildung 3.11: Mittlere Zoomstufe mit als verwendet gekennzeichneten Kartenbereichen

Auflösung zusammengefaßt, um zum einen eine schnellere Abfrage nach der Nutzung der Kacheln zu erlauben und zum anderen Speicher zu sparen. In den geringeren Auflösungsstufen wird dann nur die Nutzung von größeren Kachelblöcken abgefragt und es werden nur die Blöcke angezeigt in denen laut Bitfeld Kacheln verwendet werden. Erst in relativ hohen Vergrößerungsstufen wird auf die Datenbank zugegriffen, um die Daten einzelner Kacheln zu laden. Abbildung 3.11 zeigt den Teil einer großen Kachel, die eine Breite von 10° hat. Innerhalb dieser Kachel ist der nordwestliche Teil Deutschlands u.a. mit den Autobahnen A31 und A45 zu erkennen. Jedes der kleinen Kästchen steht für einen Block von fünf mal fünf Kacheln, von denen mindestens eine verwendet wird. Ein solcher Block ist in den dargestellten Breitengraden grob geschätzt 2,8 km breit und 5,6 km hoch.

Um beim Zugriff auf die Datenbank in den höheren Vergrößerungsstufen keine Verzögerungen und damit ein Ruckeln der Grafik zu verursachen, finden diese Zugriffe im Hintergrund statt. Die Daten der Kacheln werden dann dargestellt, sobald sie geladen werden konnten.

Durch die Kombination dieser Maßnahmen kann der Benutzer die Karte in allen Vergrößerungsstufen schnell bewegen. In den Vergrößerungsstufen, in denen einzelne Kacheln dargestellt werden, kann es lediglich dazu kommen, dass einzelne Kacheln erst verspätet dargestellt werden.

3.2.2 TraceLogViewer

Der TraceProcessor speichert in jedem Berechnungsschritt Informationen in eine Datei, die dazu verwendet werden können den genauen Ablauf der Berechnungen darzustellen. Diese Dateien können im TraceLogViewer geladen und dann Schritt für Schritt betrachtet werden.

In den Dateien, die der TraceProcessor erzeugt, werden zu Beginn alle von den

Änderungen eventuell betroffenen Kacheln und Kanten abgelegt. Danach folgt eine Reihe von einzelnen Veränderungen wie z.B. das Anlegen eines neuen Punktes oder das Auftrennen einer Kante. Damit sind alle notwendigen Informationen vorhanden, so dass der Berechnungsablauf, auch ohne dass eine Verbindung zur Datenbank benötigt wird, dargestellt werden kann. Die aktuelle Datenbank muss also keinen Bezug zu den Aufzeichnungen haben, die Aufzeichnungen lassen sich problemlos archivieren.

Beim Laden der Dateien liest der TraceLogViewer zunächst die benötigten Kacheln und Kanten ein. Die Darstellung dieser Karteninformationen übernimmt anschließend das selbe Darstellungssystem, dass auch im MapViewer verwendet wird.

Mit den einem alten Kassettenrekorder nachempfundenen Knöpfen oberhalb der Kartendarstellung kann nun in kleinen ($<$ und $>$) oder großen ($\ll$ und $\gg$) Schritten durch die Aufzeichnungen gesprungen werden.

Ein kleiner Schritt stellt dabei eine einzige Änderung dar. Der Aufbau einer Straße wird also durch die getrennten Schritte *Erstellen eines neuen Knotens* und *Hinzufügen eines Knotens zu einer Kante* dargestellt. Ein großer Schritt verdeckt diese Details und zeigt zum Beispiel direkt die Erweiterung einer Kante um einen Knoten oder das Verbinden einer Kante mit einer anderen zu einer T-Kreuzung.

Damit ist der TraceLogViewer das ideale Werkzeug, um die Entstehung von falschen Verbindung zu untersuchen. Mit den gewonnenen Erkenntnissen kann dann der TraceProcessor weiter verbessert werden.

3.3 Ausblick

Nachdem nun der bestehende Status Quo hinreichend beschrieben wurde, wollen wir uns in diesem Abschnitt mit Verbesserungen, Erweiterungen und dem weiteren Fortbestehenden des Projektes nach dem Ende der Projektgruppenzeit beschäftigen.

3.3.1 Verbesserungen & Erweiterungen

Zuerst soll auf performance-technische Veränderungen eingegangen werden:

- Der TraceServer sollte mehrere Verbindungen gleichzeitig erlauben. Dieses muss getestet werden.
- Es sollten weitere genauere Filter bereitgestellt werden. Es ließen sich z. B. lernende Filter realisieren.
- Die Ressourcen-Kontrolle im TileManager muss noch implementiert werden. Es sollen sowohl die Prozessorleistung als auch die Auslastung des Hauptspeichers mit einbezogen werden. Damit einhergehend soll der TileManager die Größe der Caches bestimmen.
- Die Parallelisierung der TraceProcessoren muss implementiert werden. Die Grundvoraussetzungen sind bereits gelegt.

- Die Suche nach dem besten Clusterknoten kann vermutlich durch die Benutzung eines Voronoi-Diagramms beschleunigt werden.

Zudem sind Erweiterungen der Funktionalität des Clusteralgorithmus geplant. Bisher werden nur neue Straßen in das System eingetragen. In der Realität kommt es aber auch vor, dass Straßen umgebaut oder komplett entfernt werden. Um diese Fälle zu berücksichtigen könnte man z. B. zu jeder Kante das Datum der letzten Spur, die zu dieser Kante hinzugerechnet wurde, speichern. Lange Zeit ungenutzte Kanten könnten dann regelmäßig entfernt werden.

Der Algorithmus wird in Zukunft Straßeneigenschaften, wie Typ (Autobahn, Ortsstraße), Spuranzahl, Ampelanlagen u. ä., erkennen. Dazu muss u.a. zu den Kanten die durchschnittliche Abweichung gespeichert werden. Daraus und mit der erzielten Geschwindigkeit ließe sich der Typ und die Spuranzahl ermitteln. Ampelanlagen könnte man durch regelmäßige, lange Haltezeiten an Kreuzungen erkennen.

Desweiteren sind noch Erweiterungen, die die Eingabe bzw. Ausgabe der Daten betreffen, denkbar:

- Bislang werden die aufgezeichneten GPS-Spuren mit dem kleinen Programm *netcat* an den TraceServer geschickt. Dies ist jedoch wenig komfortabel und dem Benutzer nicht zumutbar. Durch die einfache Schnittstelle lassen sich leicht andere Clients erstellen. Angedacht wurde eine Web-Oberfläche, mit der der Benutzer seine GPS-Spuren dem System schicken kann und – um es attraktiver zu machen – sieht, was seine Spuren bewirkt haben.

- Die intern verwaltete Karte mit den Zeitinformationen sollte Routenplanungs-Programmen zur Verfügung gestellt werden. Dazu muss die Karte in ein für diese Programme lesbares Format gebracht werden.

- Die interne Karte kann auch zum Zeichnen einer Karte verwand werden. Dies erfordert einen Rendering-Algorithmus.

3.3.2 Fortbestand des Projektes

Da wir der Ansicht sind, dass die bereits geleistete Arbeit nicht in der Versenkung verschwinden sollte, haben wir den erstellten Quelltext unter die AFL[2] gestellt. Diese Lizenz gibt zum einen den bestehenden Quelltext frei, erlaubt es aber auch darauf aufbauend proprietäre Software zu erstellen. Wir haben uns den OpenSource-Hoster BerliOS als Plattform für weitere Entwicklungen ausgesucht. Erreichbar ist dieses Projekt unter `http://mapgeneration.berlios.de`.

[2] Academic Free Licence (http://opensource.org/licenses/)

Kapitel 4

Das PDA-Modul

- Vanessa Faber • Anne Scheidler • Feng Wang -

Das Ziel der PDA-Gruppe bestand darin, ein Programm zu entwickeln, welches die Nutzung des Navigationssystems ermöglicht. Neben der grafischen Darstellung der Karte, den berechneten Routen und der Ermittlung des aktuellen Standortes wird eine Netzwerkanbindung benötigt, über welche die berechneten Routen empfangen werden können. Um die Datenbank der Map-Generation-Gruppe auch mit empirisch ermittelten Strassennetzen versorgen zu können, wird weiterhin eine Loggerfunktion benötigt, welche die Koordinaten gefahrener Strecken mit Hilfe einer seriell angeschlossenen GPS-Maus empfängt und abspeichert. Diese Strecken können ebenfalls über das Netzwerk versendet werden.

Im Folgenden soll das im Rahmen der Projektgruppe entstandene Programm *NaviSys* vorgestellt werden. Anschliessend werden die wichtigsten Funktionen dieses Programms aus implementationstechnischer Sicht genauer erläutert, bevor ein kleiner Ausblick über Zukunftsvisionen gegeben wird.

4.1 Programmbeschreibung

In diesem Abschnitt werden zunächst die Systemvoraussetzungen für das Programm *NaviSys* näher erläutert. Anschließend folgt eine benutzerfreundliche Beschreibung zur Bedienung des Programms.

4.1.1 Systemvoraussetzungen

Um zuverlässiges Arbeiten mit dem Programm zu garantieren, wird ein PDA mit dem Betriebssystem *Windows Mobile 2003 für Pocket PC Phone Edition* benötigt. Das PDA sollte mit einem *Intel StrongARM* kompatiblen Prozessor mit mindestens 200 Mhz ausgestattet sein. Außerdem werden neben mindestens 16 MB freiem Speicherplatz auf dem RAM noch 16 MB Speicherplatz auf einer Speicherkarte benötigt. Um die GPS-Maus ansprechen zu können, sollte ein serieller Anschluss vorhanden sein. Ebenso

ist ein Navigationsbutton erforderlich. Bei der grafischen Darstellung wird ein TFT-Touchscreen mit einer Auflösung von $240 \cdot 320$ Pixeln erwartet.

Um komfortabel Kartenmaterial zwischen Desktop-PC und PDA auszutauschen, wird ein Programm zum Synchronisieren der Daten zwischen PC und PDA benötigt. Hierfür wurde von der PDA-Gruppe das Programm *Active Sync* genutzt.

4.1.2 Die Bedienung

Das PDA-Programm *NaviSys* stellt dem Benutzer diverse Funktionen zur Verfügung, welche sowohl über das Menü *Start* als auch über die Toolbar-Buttons zu erreichen sind (siehe Abbildung *Das Programm NaviSys*).

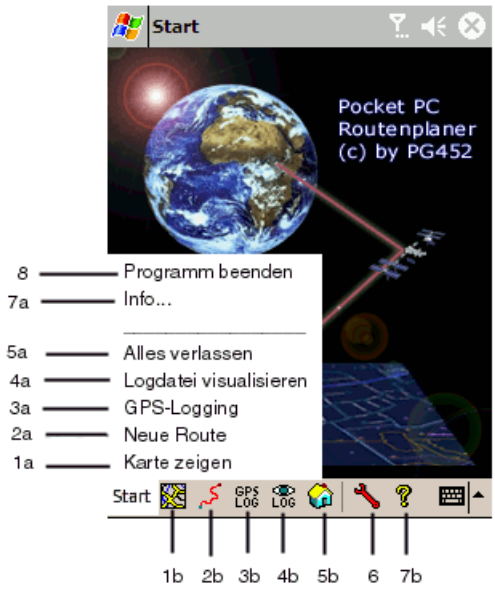


Abbildung 4.1: Das Programm NaviSys

1. Karte zeigen (1a/b) Mit Aufruf dieser Funktion wird die Straßenkarte in Abhängigkeit des aktuellen Standortes gezeichnet, sofern eine GPS-Maus augeschlossen ist. Ansonsten wird ein vordefinierter Bereich der Karte geladen.

Mithilfe des Navigationsbutton ist es möglich, Position und Auflösung der Karte zu verändern. Dem Betrachter stehen vier verschiedene Kartenauflösungen zur Verfügung, die durch mittiges Drücken auf diesen Button zyklisch verändert werden(*siehe Abbildung Zoomfaktor 0 bis 3*).

2. Neue Route (2a/b) Mit Aufruf dieses Menüpunktes gelangt der Benutzer zu folgender Oberfläche:

Hier können der Modus der Routenberechnung ausgewählt und Start-/ und Zielort eingegeben werden. Klickt man auf den Button *Suchen*, wird die Karte

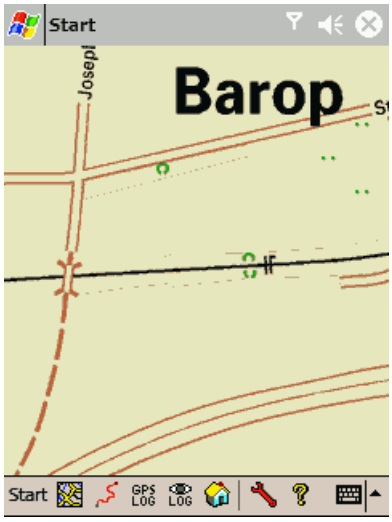


Abbildung 4.2: Zoomfaktor 0

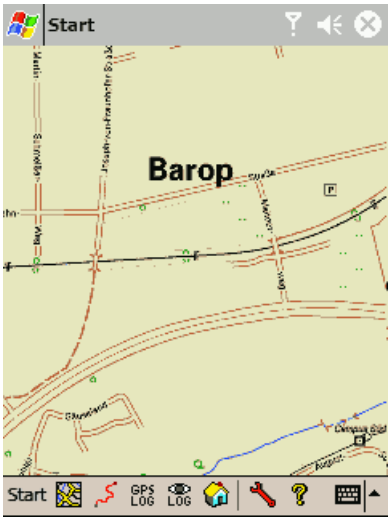


Abbildung 4.3: Zoomfaktor 1

aufgerufen, auf welcher man den Start- und Zielpunkt mit Hilfe von Fähnchen setzen kann.

Wurden die Angaben gemacht, kann der Benutzer durch Betätigen des Buttons *Übernehmen* seine Eingaben bestätigen und gelangt dann automatisch wieder zum vorher geladenen Fenster, in welchem nun unter Start und Ziel die eingegebenen Koordinaten stehen. Mit einem Klick auf *weiter* wird die Anfrage an den Routingserver versendet und nach Empfangen der berechneten Route wird diese auf der Karte gezeichnet.

3. GPS-Logging (3a/b) Wählt der Benutzer diesen Menüpunkt, öffnet sich folgendes Fenster:

Durch Drücken der Buttons *Start* und *Stop* kann der Benutzer das PDA alle empfangenen GPS-Koordinaten aufzeichnen lassen. Im Feld *GPS Aktuell* werden dem Benutzer der aktuelle Breiten- und Längengrad direkt angezeigt. Über *Daten Senden* werden die aufgezeichneten Strecken an den Datenbank-Server der Map-Generation-Gruppe geschickt und anschließend auf dem PDA automatisch gelöscht.

4. Logdatei visualisieren (4a/b) Hat der Fahrer bereits eine Strecke aufgezeichnet und möchte sich diese gerne ansehen, so kann er den Menüpunkt *Logdatei visualisieren* wählen.

5. Alles verlassen (5a/b) Über *Alles verlassen* gelangt der Benutzer wieder zurück in das Hauptmenü.

6. Tools (6) Über diesen Button gelangt der User in ein Menü, in welchem er sowohl

Abbildung 4.4: Zoomfaktor 2

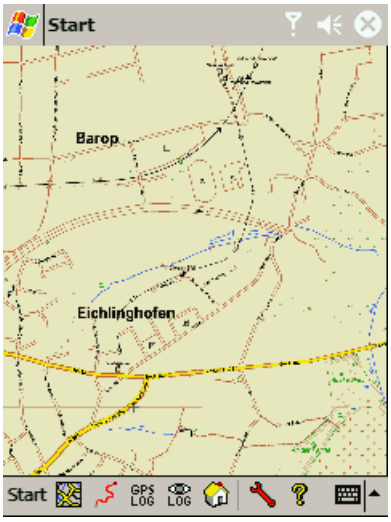


Abbildung 4.5: Zoomfaktor 3

den Speicherort der geloggten Fahrten angeben kann als auch den Ort zum Speichern der empfangenen Routen.

7. Info... (7a/b) Kurzinformation über die PG 452.

8. Programm beenden (8) Mit *Programm beenden* wird die Anwendung geschlossen.

4.2 Die Implementation

Das gesamte PDA-Programm wurde mit der Entwicklungsumgebung *Embedded Visual C++* von Microsoft in der Version 4.0 implementiert, welches zur Zeit von Microsoft zu Testzwecken kostenlos zum Download bereitgestellt wird. Als Software Developement Kit wurde das *Pocket PC 2003 SDK* genutzt, welches ebenfalls von Microsoft kostenlos zur Verfügung gestellt wird.

Die wichtigsten Gebiete der Implementation sind die Visualisierung, das Netzwerk, die Umsetzung der Logger-Funktion, die Verarbeitung der GPS-Daten und die Sprachausgabe. Im Folgenden sollen diese Hauptgebiete genauer erläutert werden.

4.2.1 Die Visualisierung

Wie bereits in der Einleitung angedeutet, gehören zum Bereich der Visualisierung sowohl die geeignete Darstellung der bereitgestellten Kartenelemente als auch die Anzeige der berechneten Routen und des aktuellen Standortes. Des weiteren wird schließlich ein Rahmenprogramm erforderlich, um die implementierten Funktionen nutzen zu

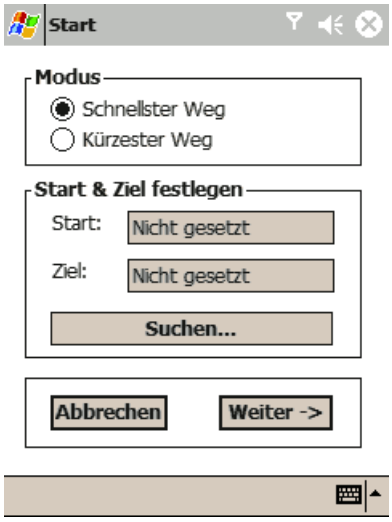


Abbildung 4.6: Die Routenanfrage

Abbildung 4.7: Setzen von Start- und Zielpunkt

können. Da die Microsoft GDI Funktionen (**G**raphics **D**evice **I**nterface) für die Darstellung von Karten, Routen und Bitmaps keine ausreichende Leistung liefern, wurde eine spezialisierte Grafikklasse geschrieben, welche direkten Zugriff auf den Grafikspeicher hat. Der Speicherzugriff wird über die GAPI (**G**ame **A**pplication **P**rogramming **I**nterface) realisiert.

Visualisierung der Kartenelemente Ausgangsbasis zur Visualisierung der Karten sind $18 \cdot 16$ Kartenelemente, die in der Auflösung von $2000 \cdot 2000$ Pixeln im TIFF-Format vorliegen. Diese Kacheln wurden zunächst auf eine Größe von $600 \cdot 600$ Pixeln reduziert, um den Speicheraufwand so gering wie möglich zu halten. Außerdem wurden die Kacheln ins Bitmap-Format konvertiert. Im Anschluss daran wurden die einzelnen Kartenelemente zu einer großen Gesamtkarte zusammengefügt und die Farbtiefe wurde auf 8 Bit mit lediglich 12 Farben reduziert. Um die Speichergröße noch weiter einzuschränken, wurde als Komprimierungsprogramm eine Win32-Anwendung implementiert. Dieses Programm erhält als Eingabe die gesamte Karte und gibt komprimierte Kacheln mit einer Größe von $1200 \cdot 1200$ Pixeln aus. Die PDA-Gruppe hat sich für die erneute Generierung von Kacheln entschieden, um das Programm möglichst universell zu halten. Sollte das Kartenmaterial zu einem späteren Zeitpunkt erweitert werden, muss die Gesamtkarte lediglich erneut mit Hilfe des Komprimierungsprogramms gekachelt werden und danach auf die Speicherkarte des PDA geschrieben werden.

Um den Komprimierungsalgorithmus beschreiben zu können, muss das Bitmap-Format genauer betrachtet werden. Eine Bitmap-Datei besteht aus vier verschiedenen Abschnitten, dem Bitmap-Header, dem Information-Header, der Farbpalette und den eigentlichen Daten (siehe [30]). Der Bitmap Header enthält die Dateisignatur *BM*, die

Abbildung 4.8: Eine berechnete Route

Dateilänge in Byte und den Abstand (Offset) zwischen Dateianfang und Datenanfang in Byte. Im Information-Header finden sich Informationen zum Bild selbst, z.B. die Höhe und Breite des Bildes, die horizontale und vertikale Auflösung in Pixel pro Meter, der Typ der Komprimierung und die Anzahl der benutzten Farben. Die Farbpalette definiert jede Farbe durch ihren Anteil an Rot, Grün und Blau. Die Daten enthalten zeilenweise die Pixelinformation des Bildes. Ausgangspunkt ist die linke untere Ecke des Bildes. Bei Bildern mit 1-, 4- oder 8-bit Farbinformation enthält der Pixelwert nicht direkt die Farbinformation, sondern einen Index auf eine Farbpalette. Die Kompression funktioniert nun folgendermaßen. Zunächst werden die für eine $1200 \cdot 1200$ Pixel große Kachel benötigten Informationen aus der erstellten Gesamtkarte eingelesen, die sich im eigentlichen Datenteil der Datei befinden. Da die Daten bei einem 8-bit-Bitmap lediglich aus Palettenindizes bestehen, wird die eigentliche Kompression nun nicht mehr schwierig. Für jeden darzustellenden Pixel existiert in der Originaldatei nur ein Byte, welches den Farbpalettenindex angibt. Beim Einlesen der Daten werden diese byteweise miteinander verglichen und gezählt. Stehen in der Originaldatei beispielsweise 10 aufeinanderfolgende identische Bytes, werden in die komprimierte Datei nur zwei Bytes geschrieben. Das erste Byte steht für die Anzahl der Pixel (10 identische Byte), das zweite Byte beinhaltet den Index des Farbtabelleneintrags. Die bei der Kompression entstandenen 72 Kacheldateien werden anschließend auf die Speicherkarte des PDA kopiert. Die neu generierten Kacheldateien beinhalten jedoch keinen sonst üblichen Dateiheader mehr und können daher nur mit dem speziell hierfür entwickelten Algorithmus zur Dekompression erkannt und schließlich dargestellt werden, da in den neuen Dateien zwar die Angaben des Farbindex vorhanden sind, jedoch die zuordnende Farbtabelle und die Bildgröße lediglich im Programm fix vordefiniert werden. Durch diese Vorgehensweise wird ein rund 100 MB großes Bitmap mit 8 Bit Farbtiefe auf knapp 10 MB reduziert.

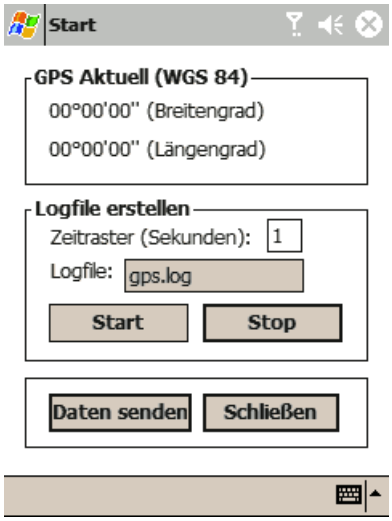


Abbildung 4.9: Das Daten-Loggen

Der Ansatz des von uns genutzten Komprimierungsalgorithmus ist in der Literatur unter der Bezeichnung RLE (**R**un **L**ength **E**ncoding) bekannt.

Visualisierung der komprimierten Kacheldateien Um die Kartenelemente auf dem PDA darzustellen, wird zunächst ermittelt, welche der Kacheln für den Visualisierungsbereich benötigt werden. Ausgangspunkt beim ersten Laden der Karte ist die oberste linke Kachel der Gesamtkarte. Werden nun Scrollvorgänge durchgeführt, verändert sich der Visualisierungsbereich und u.U. die darzustellende Kachel. Es gibt folgende Möglichkeiten für die Position des Visualisierungsbereiches:

Fall A Der benötigte Visualisierungsbereich befindet sich auf genau einer Kachel

Fall AB Der benötigte Visualisierungsbereich schneidet zwei nebeneinander liegende Kacheln

Fall AC Der benötigte Visualisierungsbereich schneidet zwei untereinander liegende Kacheln

Fall ABCD Der benötigte Visualisierungsbereich schneidet vier aneinandergrenzende Kacheln

Wurden bei der Suche nach relevanten Kacheln eine (*Fall A*) bis maximal vier Kacheln (*Fall ABCD*) gefunden, werden die komprimierten Kacheldateien dekomprimiert und vollständig in 1,4 MB große Arrays geschrieben. In den Grafikspeicher werden lediglich diejenigen Pixel geladen, welche für den darzustellenden Ausschnitt benötigt werden. Um die Grafik weiter zu beschleunigen, wurde zusätzlich eine Cache-Funktion implementiert, welche bereits dargestellte Kacheln zur Sicherheit noch im

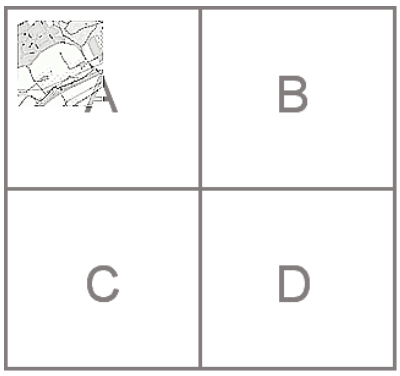


Abbildung 4.10: Fall A

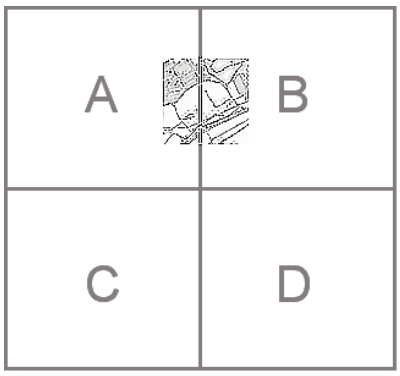


Abbildung 4.11: Fall AB

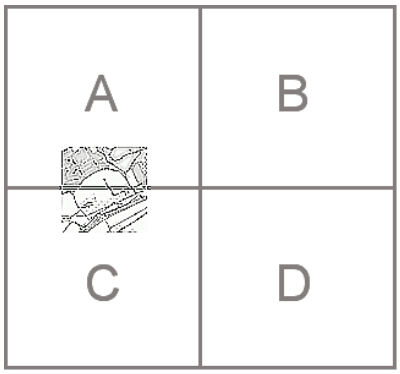


Abbildung 4.12: Fall AC

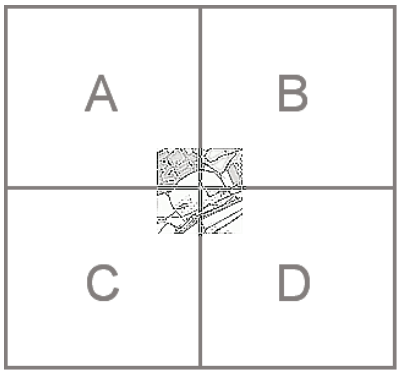


Abbildung 4.13: Fall ABCD

Speicher hält, um ggf. wieder schnell darauf zugreifen zu können. Dies ist der Fall, wenn z. B. eine Kachelbreite nach links gescrollt wird und danach wieder zurück.

Das Zeichnen der Routen Da die Koordinaten der Route einen gewissen Abstand voneinander haben, in der Visualisierung aber Strecken und Kurven jeglicher Form erwünscht sind, bedarf es einer Aufarbeitung der empfangenen Routen-Daten. Zunächst wird die kürzeste Verbindung zwei aufeinander folgender Koordinaten berechnet. An jedem Punkt dieser Strecke wird dann ein Kreis gezeichnet um weiche Konturen der Route zu garantieren und eine geeignete Linienstärke zu schaffen. Die Kreise werden dabei über Sinus-/ und Cosinusfunktionen berechnet. Um kostbare Rechenleistung einzusparen, werden die Sinus und Cosinunsfunktionen jedoch nicht für jeden Punkt neu berechnet. In einem Preprocessing werden zwei eindimensionale, 360 Felder große Arrays mit den Sinus- bzw. Cosinus-Werten der Winkel 1..360° gefüllt. Somit ist es möglich, die rechenintensiven Winkelfunktionen durch schnelle Speicherzugriffe über Indizierung zu ersetzen.

Algorithmus 4 Dekomprimieren

Eingabe:
$komprimiert$ {Eindimensionales Array mit komprimierten Daten}
$groesse$ {Grösse des Arrays mit komprimierten Daten}
Ausgabe:
$dekomprimiert$ {Eindimensionales Array mit dekomprimierten Daten}

$offset \leftarrow 0$
for $index1 = 0$ to $groesse$ **do**
  $pixelanzahl \leftarrow komprimiert_{index1}$
  $farbindex \leftarrow komprimiert_{index1+1}$
  for $index2 = 0$ to $pixelanzahl$ - 1 **do**
    $dekomprimiert_{index2+offset} \leftarrow farbindex$
    $index2 \leftarrow index2 + 1$
  end for
  $offset \leftarrow offset$ + $pixelanzahl$
  $index1 \leftarrow index1 + 2$
end for

4.2.2 Das Netzwerk

Verbinden mit dem Server Die Implementierung der Netzwerkanbindung wird durch zwei Klassen realisiert. Die MFC Klasse `CCeSocket` (abgeleitet von `CSocket`) stellt dabei das Herzstück der Netzwerkkommunikation dar, während die Klasse `CNetworkDlg` (abgeleitet von `CDialog`) die Schnittstelle zum Hauptprogramm bildet. Um eine Kommunikation mit dem Server zu ermöglichen, wird ein Socket erstellt, welches den ersten Verbindungsaufbau übernimmt. Bei erfolgreichem serverseitigen Akzeptieren der Verbindung können dann Daten über dieses Socket gesendet und empfangen werden.

Routenanfrage Das Programm ist so konzipiert, dass vor dem möglichen Herstellen einer Verbindung zum Server alle relevanten Daten vom Benutzer abgefragt wurden. Diese Daten werden dann in einer Zeichenfolge zusammengefasst und an den Server übermittelt. Nach der Übermittlung der Anfrage geht das Programm in den Listening-Mode und erwartet die berechnete Route. Die eingehenden Rohdaten werden in einer Datei gesammelt, um später für Sprachausgabe und Routendarstellung weiter verarbeitet zu werden. Sobald das Programm eine empfangene Zeichenfolge mit dem Inhalt `end` registriert, wird die Datei geschlossen, die Serververbindung abgebaut und das Programm springt zum Routing Modus.

Übermittlung der LOG-Datei Die durch einen Log-Vorgang erstellte LOG-Datei wird zeilenweise an den Server übermittelt.

4.2.3 Die Logger-Funktion

Das Ansteuern des Seriellen Anschluss befindet sich in der Klasse `Serial`. Die Klasse `Serial` beinhaltet gleichzeitig die Loggerfunktion. Die GPS Mouse wird an den seriellen Port des PDAs angeschlossen. Über die Mouse werden nun die NMEA 0183 Datensätze empfangen und über die Klasse `Serial` verarbeitet bzw. gespeichert. Die Notwendigkeit ergibt sich zum einen durch das Anzeigen der aktuellen Position und zum anderen durch das Sammeln realer NMEA 0183 Traces. Die Klasse gibt ihre Daten in Form der aktuellen Position zum einen an die Unterklasse `AnaNMEA` der Konverter Klasse und zum anderen werden die abgespeicherten Traces mit Hilfe der TCP/IP Struktur an die Gruppe *MapGeneration* geschickt. Zuerst wird der serielle Port initialisiert. Hierfür wird der serielle Port geöffnet und danach die entsprechenden Parameter eingestellt. Die Parameter befinden sich in der DCB- und Timeout-Strukturen. DCB legt als ersten Parameter die Baud Rate auf CBR4800 fest. Die Byte-Größe wird auf 8 gesetzt und wir legen keine Parität fest. Zuletzt wird ein Stoppbit angegeben. In der Timeout-Struktur legen wir Parameter für die `read`-Funktion fest. Das Format des aufgenommenen NMEA 0183 Strings ist ASCII. Die Methode der Funktion Datenaufnahme wird als Thread realisiert. Der ganze Vorgang wird automatisch mit Hilfe einer unendlichen Schleife wiederholt, bis das Programm beendet wird. Dazu werden folgenden Funktionen realisiert: Jedes Mal werden 1000 ASCII Zeichen aus der GPS Maus gelesen und in einem Array gespeichert. Dann wird das Array nach Dollarzeichen durchgesucht, welches das erste Zeichen eines NMEA-0183 Strings ist. Die Zeichen, die nach dem Symbol *$* aufgenommen worden sind, werden solange in einen String geschrieben, bis das Sonderzeichen CR/LF gelesen wird, das das Ende eines kompletten NMEA 0183 Strings bezeichnet. Nun wird der ausgelesene NMEA 0183 String in einer externen TXT-Datei gespeichert, die dann wie oben beschrieben weiterverarbeitet bzw. über das Netzwerk verschickt werden kann. Anschließend wird mit Hilfe der Funktion `sleep(100)` der Lesevorgang für 100 Millisekunden blockiert. Dann beginnt der Vorgang wieder von vorne. Der NMEA 0183 String wird dadurch alle 100 Millisekunden aktualisiert. Beim Programmstart wird die Methode `start` der Klasse `Serial` aufgerufen, um diesen Thread einzuschalten.

4.2.4 Berechnung der GK-Koordinaten und Pixelkoordinaten

Zur Darstellung der aktuellen Position auf der Karte wird eine Konvertierung der GPS-Koordinaten in Gauss-Krüger-Koordinaten und eine anschließende Umrechnung in Pixelkoordinaten notwendig. Die Klasse `GPS2GK` bildet hierbei den Kern der Transformation. Der Algorithmus stammt von Ottmar Labonde (siehe [14]) und läuft unter dem Namen *WGSDHDN3*. Weitere Informationen sind dem Abschnitt 1.3 zu entnehmen.

4.2.5 Die Sprachausgabe

Das Ziel der Funktion Sprachausgabe ist es, sprachliche Informationen für den Fahrer anzubieten. Mit der Funktion der Sprachausgabe wird die Situation vermieden, dass der Fahrer seinen Blick von der Straße abwenden muss, um auf das PDA zu schauen. Die akkustische Vermittlung der Informationen stellt sicher, daß die Aufmerksamkeit des Fahrers nicht durch die Navigationsanweisungen des PDAs beeinträchtigt wird.

Wir verwenden den Dateityp WAV, weil er von fast allen Betriebsystemen und Programmierumgebungen unterstützt wird. Zur Erzeugung der WAV Datei wird eine Software mit TTS Technik (**T**ext**T**o**S**peech) benutzt. TTS leistet die Umwandlung von Text in natürlich klingende Sprache. Es werden viele Produkte mit TTS Technik auf dem Markt angeboten. Es ist jedoch kein einziges davon kostenlos. Daher haben wir eine Demosoftware verwendet, um die WAV Dateien zu erzeugen. Wir haben die Demosoftware *Elan SaySo* von der Firma *Acapela Gruppe* benutzt, da sie die beste Tonqualität unter allen von uns getesteten Produkten bietet und viele Optionen zulässt. Die WAV-Dateien werden auf dem PDA gespeichert.

Das PDA bekommt den Trace vom Server in Form von Informationseinheiten die aus den Gauß-Krüger-Koordinaten und zusätzlichen Meldungsinformationen bestehen. Vor und nach jeder Kreuzung enthält der Trace eine Meldungsinformation. Vor der Kreuzung gibt es immer einen String *CRE* oder *CLI* . Nach der Kreuzung folgt ein String, der aus *DRE* bzw. *DLI* und einer Zahl als Angabe zur Distanz bis zur nächsten Kreuzung besteht. Das Hauptprogramm stellt sicher, dass zu jeder Kreuzung und in bestimmten Entfernungen vor der Kreuzung die Funktion *Sprachausgabe* aufgerufen wird. Die mit der TTS-Technik erstellten Meldungen werden mit der API Funktion `Play Sound` abgespielt.

An der Stelle des Strings CRE wird *Sofort nach rechts abbiegen* abgespielt, an der Stelle des Strings CLI *Sofort nach links abbiegen*. Nach der dem Auslesen des Strings CRE oder DLI überwacht das Hauptprogramm ständig, ob die nächste Koordinate des Traces erreicht wird. Bei jeder Koordinate des Trace wird die neue Distanz zur folgenden Kreuzung berechnet. Die neue Distanz ergibt sich aus der Differenz der alten Distanz und des seitdem zurückgelegten Weges. Danach wird überprüft, ob die neue Distanz in einem von mehreren Intervallen liegt, in denen, eine Meldung erfolgen soll. Ist die Entfernung zur Kreuzung 390 - 410 m, 290 - 310 m, 190 - 210 m oder 90 - 110 m, wird eine WAV-Datei abgespielt, die den Fahrer auf die bevorstehende Abbiegungsmöglichkeit hinweist. Wenn die vorherige Koordinate auch schon im betreffenden Intervall enthalten war, dann wird die wiederholte Meldung unterdrückt. Wenn an der Koordinate 309 m vor der Kreuzung *in 300 m links* abgespielt wird, dann ist es nicht mehr nötig, dies an den Koordinaten 302 m und 291 m vor der Kreuzung zu wiederholen.

4.3 Ausblick

Innerhalb eines Jahres ist im Rahmen dieser Projektgruppe ein zuverlässig laufendes PDA-Programm entstanden, welches die oben erklärten Funktionen zur Verfügung stellt.

Für zukünftige Weiterentwicklungen gibt es jedoch einige Gesichtspunkte, anhand derer das Programm verbessert und weiterentwickelt werden kann und sollte. Beispielsweise wird zum jetzigen Zeitpunkt lediglich auf Kartenmaterial des Dortmunder Raumes gearbeitet. Um das System global nutzen zu können, wäre es sinnvoll, eine Datenbank bereitzustellen, welche mindestens bundesweites Kartenmaterial zur Verfügung stellt. Da die Speicherkapazität des PDA beschränkt ist, sollte dieses Kartenmaterial individuell auf den Bedarf des Nutzers abgestimmt werden können, was zum Beispiel durch ein zusätzliches PC-Programm realisiert werden könnte, welches der User bei sich zu Hause installieren kann.

Ebenfalls sinnvoll wäre eine erweiterte Funktionsweise der Sprachausgabe. Z. B. könnte man für die interessantesten Städte Deutschlands oder sogar der Welt Touristeninformationen anbieten und Stadtrundfahrten mit Sprachausgabe realisieren.

Auch eine Suchfunktion nach Strassen oder zumindest Stadtteilen auf dem vorhandenen Kartenmaterial könnte durchaus sinnvoll sein, um die Start- und Zielangabe bei einer Routenanfrage zu vereinfachen.

Alles in Allem gibt es noch einige Funktionen, um welche das Programm NaviSys erweitert werden kann. Beim Implementieren des bis zum jetzigen Zeitpunkt stehenden Programms wurde hingegen sehr viel Wert auf Zuverlässigkeit und Geschwindigkeit gelegt und nicht auf die Vielfalt der Funktionen. Dies erscheint für die Zukunft vernünftig, da sich potentielle zukünftige Programmierer nicht mehr komplett in den bereits bestehenden Code einarbeiten müssen, sondern direkt mit Weiterenwicklungen beginnen können.

Kapitel 5

Das Routing-Modul

- Heiner Ackermann • Thomas Härtel -

Die Routinggruppe beschäftigte sich mit der Implementierung einer Serveranwendung, die zwei Anwenderschnittstellen zur Verfügung stellt.

1. **Berechnung kürzester Wege** Basierend auf einem Routinggraphen berechnen wir kürzeste und schnellste Wege zwischen einem Start-Ziel-Koordinatenpaar. Ein Anwender kann diese über eine TCP/IP-Schnittstelle an unseren Server schicken und erhält wahlweise eine Folge von GPS- oder Gauß-Krüger-Koordinaten, die einen Weg zwischen der Start- und der Zielkoordinaten beschreibt.

2. **Aktualisierung des zu Grunde liegenden Routinggraphen** Um kurzfristige Fahrzeitänderungen, wie zum Beispiel Staus oder Straßensperrungen, bei der Berechnung von kürzesten oder schnellsten Wegen berücksichtigen zu können, haben wir eine graphische Benutzeroberfläche entwickelt, über die Manipulationen am Routinggraph vorgenommen werden können.

Im nächsten Abschnitt 5.1 beschreiben wir zunächst die zugrunde liegende Datenbasis des Routinggraphen.Wir beschreiben in welchem Format uns diese Daten durch die MapGeneration-Gruppe zur Verfügung gestellt werden, und wie wir daraus den Routinggraphen erzeugen.

In Abschnitt 5.2 beschreiben wir die Benutzerschnittstellen. Wir beginnen mit der Schnittstelle zur PDA-Gruppe, über die ein Benutzer Anfragen an unseren Server schicken kann. Anschließend folgt die Beschreibung der graphischen Oberfläche und ihrer Funktionalität, über die Aktualisierungen am Routinggraph vorgenommen werden können.

Im Abschnitt 5.3 beschreiben wir sequentiell den Programmablauf einer Benutzeranfrage. Dabei gehen wir zunächst auf die Berechnung nächster Nachbarn mit Voronoidiagrammen ein. Danach beschreiben wir, wie wir kürzeste und schnellste Wege mit dem Algorithmus von Dijkstra berechnen und welche Speed-Up-Techniken wir dazu in Betracht gezogen haben. Abschließend beschreiben wir eine Nachverarbeitung der Ausgabe, in der wir Abbiegevorschläge an Kreuzungen berechnen.

5.1 Der Routinggraph

Der Routinggraph wird uns von der MapGeneration-Gruppe zur Verfügung gestellt. Er ist ein gerichteter Graph, der vom konkreten Straßenverlauf abstrahiert, da zur Berechnung kürzester und schnellster Wege lediglich die Fahrzeit beziehungsweise die Entfernung zwischen Abbiegemöglichkeiten von Bedeutung ist. Die Knoten des Graphen sind in die Ebene eingebettet und beschreiben Koordinaten an denen ein Fahrzeug seine Fahrt in mehrere Richtungen fortsetzen kann. Die gerichteten Kanten sind mit 96 Fahrzeitinformationen und der Kantenlänge annotiert. Dabei beschreibt die i-te Fahrzeitinformation die Fahrzeit, die ein Fahrzeug benötigt, um die Kante im i-ten Zeifenster zu überqueren. Dazu haben wir die 24 Stunden eines Tages in 96 Zeitfenster zu je 15 Minuten eingeteilt. Zu jeder Kante existiert außerdem ein Eintrag in einer Datenbank, der den exakten Verlauf der Kante als Folge von GPS-Koordinaten beschreibt. Der exakte Verlauf einer Kante wird bei der Ausgabe des berechneten Weges berücksichtigt.

Der Routinggraph wird von der MapGeneration-Gruppe erzeugt und in einem speziellen Format in 2 Textdateien abgelegt und uns so zur Verfügung gestellt. Wir beschreiben nun das Format der Textdateien. Der Graph wird in einer Datei mit dem Namen *graph.txt* gespeichert, die wie folgt strukturiert ist:

#Kommentar
N,$longitude, latitude$
E,$v_{Start}, v_{Ziel}, database\text{-}ID, length, t(1), ..., t(96)$

Die Datei kann mit beliebig vielen Kommantarzeilen beginnen, solange eine solche Zeile mit einem # Symbol beginnt. Es folgt eine Auflistung der Knoten. Eine Zeile, die einen Knoten beschreibt, beginnt mit *N*, gefolgt von der Längen- und Breitengradangabe (GPS-Koordinaten) des Knotens. Die beiden Werte werden durch ein Komma getrennt. Die Reihenfolge der Knoten definiert eine Nummerieung der Knoten, die wir bei der Definition der Kanten ausnutzen. Eine Kante wird durch eine Zeile beschrieben, die mit *E,* beginnt. Es folgt die Angabe des Start- und Zielknotens der Kante, deren Nummern sich auf die implizite Nummerierung der Knoten beziehen. Daran schließt sich eine Variable $database\text{-}ID$ an, unter der die Detailinformationen der Kante in der Datenbank abgerufen werden können. Die Detailinformationen beschreiben den exakten Verlauf einer Kante als Folge von GPS-Koordinaten. Es folgt die Länge der Kante in Metern und die 96 Fahrzeitinformationen. Alle Werte werden durch Kommata getrennt.

Die Detailinformationen der Kanten werden zurzeit noch in einer Textdatei abgespeichert, die das folgende Format hat.

$database\text{-}ID, longitude, latitude, ...$

Jede Zeile beginnt mit der $database\text{-}ID$ der Kante, die durch diese Zeile beschrieben wird, gefolgt von den Längen- und Breitengraden, die den genauen Verlauf der Kante beschreiben. Zu beachten ist, dass die Koordinaten von Start- und Zielknoten nicht Bestandteil der Detailinformationen einer Kante sind.

Wir haben den Routinggraphen mit der Bibliothek *LEDA Version 3.1.9* [18] implementiert. Knoten werden durch die Klasse `POINT` repräsentiert, Kanten durch die Klasse `SEGMENT`, die wir an unsere Anforderungen angepasst haben. Den Graph erzeugen wir mit Hilfe des parametrisierten Graphentemplates `GRAPH<POINT, SEGMENT>`.

5.2 Anwenderschnittstellen

5.2.1 Die Schnittstelle zur PDA-Gruppe

Unser System ist in der Lage, verschiedene kürzeste und schnellste Wegeanfragen zu beantworten. Es akzeptiert zudem sowohl GPS- als auch Gauß-Krüger-Koordinaten in der Eingabe und berechnet in Abhängigkeit des spezifizierten Koordinatensystems eine Ausgabe im selben Koordinatensystem. Wir unterstützen die folgenden Routinganfragen.

1. Berechnung eines *kürzesten* (S, T)*-Weges*.
2. Berechnung eines *schnellsten* (S, T)*-Weges*. Dabei unterscheiden wir zwischen 2 verschiedenen Varianten:
   (a) Berechnung eines schnellsten (S, T)-Weges zu einer *vorgegebenen Abfahrtszeit* t *am Knoten* S.
   (b) Berechnung eines schnellsten (S, T)-Weges zu einer *vorgegebenen Ankunftszeit* t *am Knoten* T.

   Offensichtlich sind die Varianten 2a und 2b symmetrisch. Variante 2b entspricht Variante 2a im Graphen G', indem alle Kanten umgedreht worden sind.

Routinganfragen können an unseren Server als String im folgenden Format geschickt werden.

$$Format, Anfragetyp, Zeit, Start, Ziel$$

Das Format ist entweder der String *GPS* oder *GK*, der angibt, in welchem Koordinatensystem Start- und Zielkoordinaten angegeben werden. Der *Anfragetyp* ist eine Zahl aus {*1,2,3*}, die angibt was für ein Weg berechnet werden soll. Dabei sind die drei Anfragetypen wie folgt codiert.

1 kürzester (S, T)-Weg.

2 schnellster (S, T)-Weg zur vorgegebenen Abfahrtszeit *Zeit*.

3 schnellster (S, T)-Weg zur vorgegebenen Ankunftszeit *Zeit*.

Im Fall *1* einer kürzesten Wegeanfrage kann die Variable Zeit auf einen beliebigen Wert gesetzt werden. Der String endet mit den Koordinaten von Start- und Zielknoten im spezifizierten Koordinatensystem, im Fall von *GPS* zunächst der Längengrad, dann der Breitengrad, im Fall von *GK* zunächst der Rechtswert, dann der Hochwert. Alle Werte werden durch Kommata getrennt. Es folgt ein Beispiel.

$$GPS, 1, 0, 50.1553980, 8.6413480, 50.1513820, 8.6250820$$

Der Server berechnet zu jeder Anfrage eine Route und schickt diese über die bestehende Netzwerkverbindung an den Anfragesteller zurück. Die Serverantwort besteht aus einer Folge von GPS- oder Gauß-Krüger-Koordinaten, die den kürzesten bzw. schnellsten Weg zur Anfrage exakt beschreiben. Dazu laden wir die Detailinformationen der Kanten, über die der Weg führt, aus der Datenbank und verschicken diese als String über die Netzwerkverbindung. Zusätzlich markieren wir Koordinaten, an denen ein Fahrzeug seine Fahrt in mehrere Richtungen fortsetzen kann und schlagen Abbiegerichtungen vor. Einer solchen Koordiante stellen wir den Sting *C*$\{$*Re,Li*$\}$ voran, wobei $\{$*Re,Li*$\}$ die Richtung an dieser Kreuzung angibt. Außerdem berechnen wir den Abstand zwischen aufeinanderfolgenden Kreuzungen in Metern und fügen diesen und die Richtung an der nächsten Kreuzung hinter jeder Kreuzung ein. Dies markieren wir durch den String *D*$\{$*Re,Li*$\}Dist$. Den Abstand vom Startpunkt des Weges zur ersten Kreuzung fügen wir vor der Wegbeschreibung ein. Es folgt ein Beispiel. Dabei bezeichnet $Dist_1$ den Abstand zur ersten Kreuzung C_1, deren Koordinate durch x_i, y_i angegeben wird.

$$D_1 ReDist_1, x_1, y_1, ..., C_1 Re, x_i, y_i, D_2 LiDist_2, x_{i+1}, y_{i+1}, ..., end$$

5.2.2 Das User-Interface zur Aktualisierung des Routinggraphen

Unser User-Interface zu Aktualisierung des Routinggraphen, nachfolgend Grapheditor genannt, wurde mit dem `wxWidgets GUI-Toolkit` entwickelt. Der Grapheditor stellt den gesamten Routinggraphen des Servers dar. Dieser kann per Netzwerkverbindung oder aus einer lokalen Textdatei geladen werden. Zur Fehlerbeseitigung können noch die Detailkanten nachgeladen, sowie berechnete Routen visualisiert werden.

Auf dem Routinggraphen können Kantenaktualisierungen vorgenommen werden. Es kann genau eine Kante ausgewählt oder ein ganzer Bereich markiert werden, um

mehrere Kanten gleichzeitig zu manipulieren. Für jede Routingkante können einzeln alle 96 Zeitfenster verändert werden. Bei Mehrfachauswahl kann eine relative Verlangsamung oder Beschleunigung der Fahrzeit für die gewählten Kanten pro Zeitfenster eingestellt werden. Auch die Knotenpositionen können bearbeitet werden, um beispielsweise Korrekturen am Graphen durch Ungenauigkeiten vorzunehmen. Durch die Änderungen an den Knoten werden auch die Längen der ein- und ausgehenden Kanten aktualisiert.

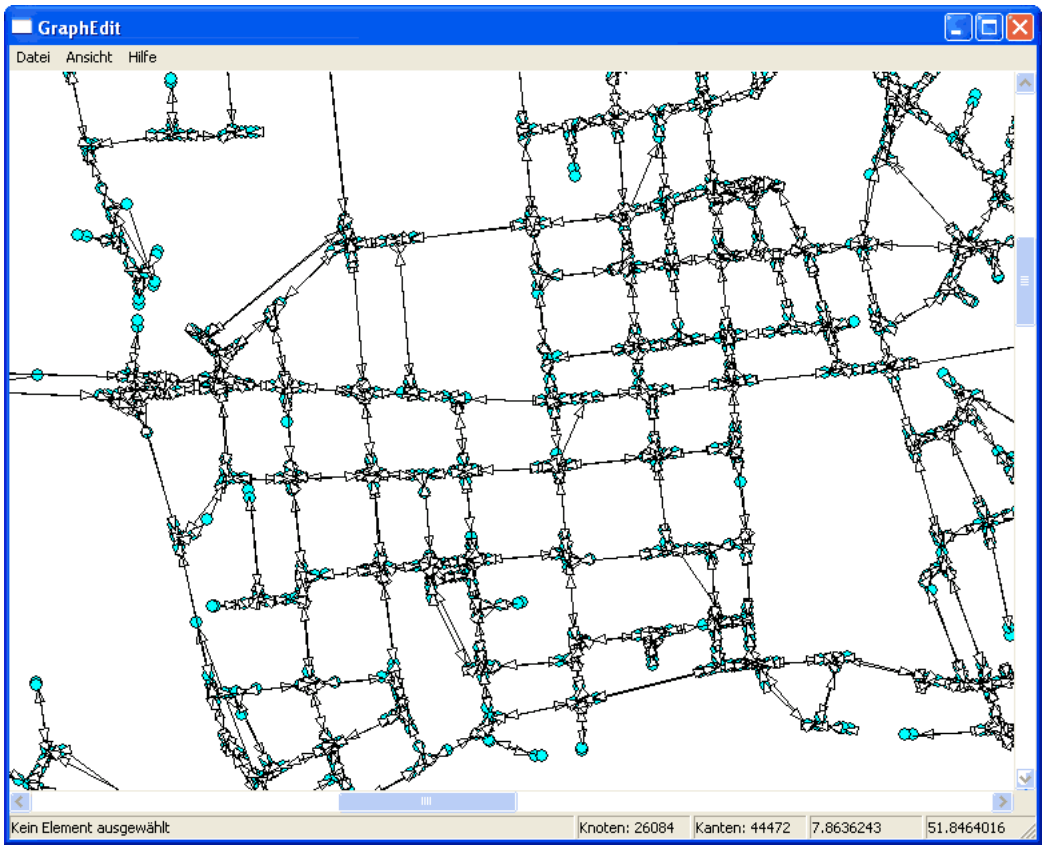


Abbildung 5.1: Visualisierung des Routinggraphens

5.3 Bearbeitung einer Routinanfrage

In diesem Abschnitt beschreiben wir, welche Teilprobleme zur Beantwortung einer Routinganfrage gelöst werden müssen und welche Techniken wir dazu verwenden. Unsere Implementierung basiert auf den Ergebnissen der Diplomarbeit unseres Betreues Shahid Jabbars, die wir an unsere Zwecke angepasst haben. Die Ergebnisse dieser Arbeit sind in [7] zusammengefasst. Zu Beantwortung einer Routinganfrage müssen die folgenden Schritte in dieser Reihenfolge ausgeführt werden.

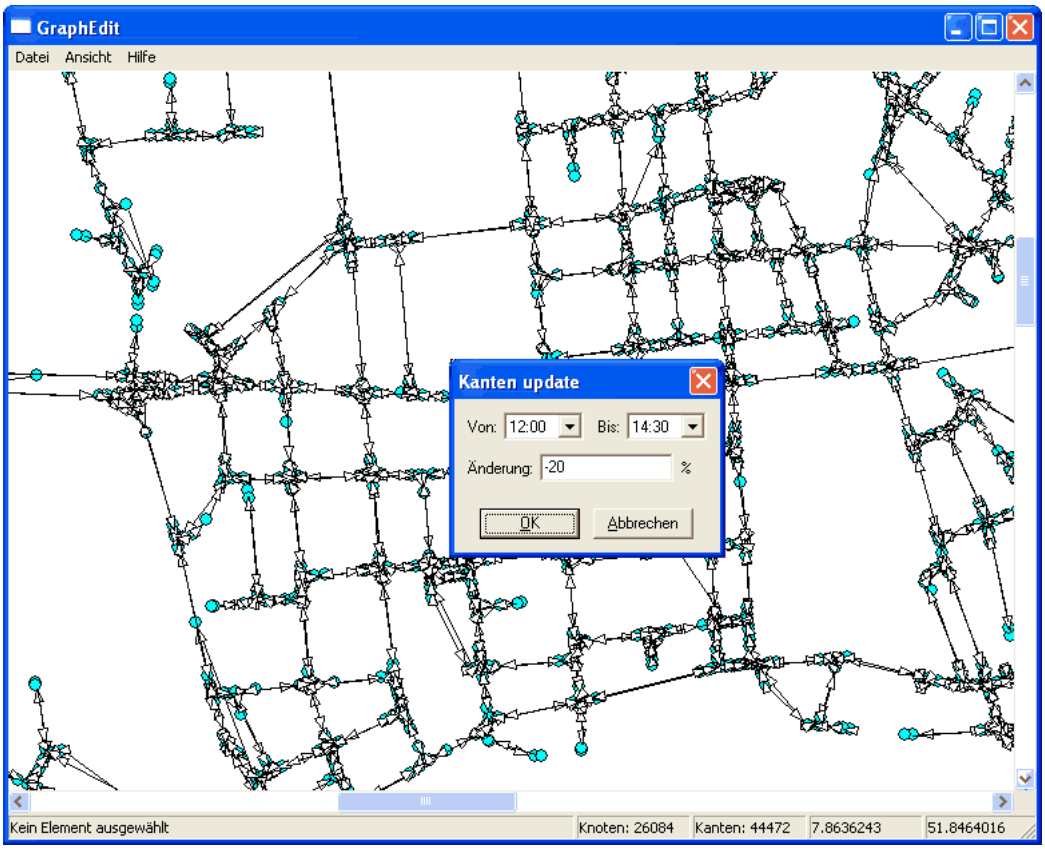


Abbildung 5.2: Aktualisierung der Kanten

1. Einordnung der Start- und Zielkoordinaten in den Routinggraphen. Dazu verwenden wir Voronoidiagramme.

2. Berechnung kürzester bzw. schnellster Wege mit dem Algorithmus von Dijkstra. An dieser Stelle beschreiben wir auch die von uns betrachteten Speed-Up Techniken.

3. Nachverarbeitung und Annotation des berechneten Weges.

5.3.1 Berechnung eines nächsten Nachbarn

Alle Routinganfragetypen haben gemeinsam, dass zunächst zwei Knoten S' und T' im Routinggraphen bestimmt werden müssen, von denen aus ein kürzester bzw. schnellster Weg berechnet werden kann, da wir nicht davon ausgehen können, dass die Koordinaten S und T als Knoten im Routinggraphen vorhanden sind. Dieses Teilproblem ist ein Nächstes- Nachbarn Problem. Eine naheliegende Lösung für diese Poblem ist es, diejenigen Punkte S' und T' zu bestimmen, die minimalen Abstand zu S bzw. T haben.

Zur schnellen Lösung des Nächste-Nachbarn Problems berechnen wir, nachdem wir den Graphen aus der Textdatei ausgelesen haben, ein Voronoi-Diagramm. Mit Hilfe des Voronoidiagramms können wir den nächsten Nachbarn einer Start- bzw. Zielkoordiante in Zeit $O(\log n)$ berechnen. Die Berechnung des Voronoidiagramms benötigt Zeit $O(n \log n)$. Diese Funktionen werden von der Bibliothek LEDA zur Verfügung gestellt. Eine theoretische Einführung in das Themangebiet der Nächsten-Nachbarn Berechnung und Voronoidiagramme findet man zum Beispiel in [3].

5.3.2 Kürzeste Wege und der Algorithmus von Dijkstra

Die drei von uns unterstützten Routingvarianten erfordern alle die Lösung eines Kürzesten-Wege Problemes mit verschiedenen Kostenfunktionen. Varainte 1 entspricht dem allgemein bekannten Single-Source-Single-Target-Shortest-Path Problem und kann mit dem Algorithmus von Dijkstra gelöst werden.

Die Varianten 2a und 2b sind ebenfalls Single-Source-Single-Target-Shortest-Path Probleme. Dabei ist allerdings zu berücksichtigen, dass die Bewertungsfunktion zeitabhängig ist. Mit einer entsprechenden Anpassung können auch diese Varianten mit dem Algorithmus von Dijksta gelöst werden. Unmittelbar einsichtig ist diese Feststellung nicht, aber der Versuch ein Beispiel zu konstruieren, bei dem der Algorithmus von Dijksta nicht den schnellesten Weg berechnet, liefert eine Beweisidee, um die Korrektheit dieses Verfahrens zu zeigen. Um kürzeste Wege schnell berechnen zu können, haben wir verschiedene Preprocessing und Speed-Up Techniken betrachtet.

1. Lösung des All-Pairs-Shortest-Path Problem

2. Zielgerichtete Suche

3. Das Containerkonzept von Wagner, Willhalm und Zaroliagis [28, 29]

Diese stellen wir im Folgendem kurz vor und beschreiben, wie wir diese eingesetzt haben.

Lösung des All-Pairs-Shorstest-Path Problem Die Lösung des All-Pair-Shorstest-Path Problem erlaubt es, Routinganfragen sehr schnell zu beantworten. Ein Nachteil dieses Ansatzes ist allerdings, dass er $O(n^2)$ Speicherplatz benötigt. Wir werden ihn deshalb nicht weiter verfolgen.

Zielgerichteten Suche Zielgerichtete Suche kann immer dann eingesetzt werden, wenn der betrachtete Graph in die Ebene eingebettet ist. Prinziell versucht man dabei, zunächst solche Kanten zu bevorzugen, die in Richtung des Ziels zeigen. Dazu modifiziert man die Kantengewichte der Art, dass das Gewicht von Kanten, die in Richtung des Ziels zeigen, kleiner und für andere Kanten größer wird. Sei s der Startknoten,

t der Zielknoten und $g(u, v)$ das Gewicht der Kante von u nach v. Das modifizierte Kantengewicht berechnet sich dann durch

$$g'(u, v) = g(u, v) + \mathit{Abstand}(v, t) - \mathit{Abstand}(s, t).$$

Man kann von dieser modifizierten Gewichtsfunktion zeigen, dass der Algorithmus von Dijkstra angewendet auf den Graphen mit den modifizieten Gewichten, stets einen bezüglich der Gewichtsfunktion g optimalen Weg berechnet.

Das Containerkonzept Das Containerkonzept wurde von Wagner und Willhalm [28] vorgeschlagen. Die zentrale Idee dieses Ansatz ist es, zu jeder Kante (s, v) eine Datenstruktur zur Verfügung zu stellen, in der alle Knoten t gespeichert werden, für die gilt: ein kürzester Weg von s nach t führt über die Kante (s, v). Das explizite Abspeichern aller Knoten t benötigt ebenfalls zu viel Speicherplatz, so dass angenommen wird, dass die Knoten in die Ebene eingebettet sind. Zu jeder Kante (s, v) wird dann ein geometrisches Objekt (z.B. ein achsenparalleles Rechteck) bzw. Container gespeichert, so dass alle Knoten t, unter Umständen auch noch weitere, innerhalb dieses geometrischen Objektes liegen. Dieser Ansatz ist in einem gewissen Sinne nur noch heuristisch, da es unter Umständen vorkommen kann, dass ein Knoten t in einem geometrischen Objekt einer Kante (s, t) enthalten ist, obwohl ein kürzester Weg von s nach t nicht über diese Kante führt. Im Algorithmus von Dijkstra berücksichtigen wir bei einer kürzesten Wegberechnung von s nach t dann nur noch solche Kanten, in deren Container der Zielknoten t enthalten ist. Die Berücksichtigung der Container im Algorithmus von Dijkstra garantiert immer noch die korrekte Berechnung eines kürzesten Weges. Die bisher beschriebenen Container nennen wir *Target-Container*. Wagner und Willhalm haben in Experimenten gezeigt, dass mit Hilfe dieser Container die Laufzeit des Algorithmus von Dijkstra um bis zu 90% gesenkt werden kann.

Unter Umständen sind die Kantengewichte aber nicht statisch, sondern dynamisch. Die Änderung eines Kantengewichtes kann zu Änderungen der Container führen. Auch für dieses Problem haben Wagner, Willhalm und Zaroliagis [29] eine Lösung vorgestellt. Zunächst definieren sie zu jeder Kante einen *Source-Container*. Ein Source-Container speichert zu einer Kante (u, t) alle diejenigen Knoten s, so dass ein kürzester Weg von s nach t mit der Kante (u, t) endet. Mit Hilfe der Source- und Targetcontainer können Kantenupdates relativ schnell durchgeführt werden, ohne dass alle Container neu berechnet werden müssen.

Container und kürzeste Wege Die Länge einer Kante ist eine statisiche Information. Unter Umständen kann es aber vorkommen, dass eine Straße gesperrt wird, so dass die zugehörige Kante gelöscht werden müsste. Dies kann simuliert werden, indem die Länge der Kante auf unendlich gesetzt wird. Dies führt dazu, dass sowohl Target- als auch Sourcecontainer aktualisert werden müssen.

Prinzipiell wird dieser Fall komplett durch den Ansatz von Wagner, Willhalm und Zaroliagis erfasst, so dass wir keine Änderungen an diesem vornehmen müssen.

Container und schnellste Wege Schnellste Wege können abhängig von der gewählten Abfahrtszeit verschieden sein. Grundsätzlich erwarten wir aber keine großen Unterschiede bei einem schnellsten Weg von s nach t zu unterschiedlichen Abfahrtszeiten.

Ein naheliegender Ansatz zur Lösung dieses Problemes ist der Folgende: Wir lösen zu jedem möglichen Zeitfenster das All-Pair-Shortest-Path Problem und erzeugen daraus zu jeder Kante e zu jedem Zeitfenster einen Container $C_e(t)$. Dieser Ansatz erscheint uns zu speicherplatzintensiv, so dass wir vorschlagen, alle Container $C_e(t)$ zu einer Kante e zu vereinigen:

$$C_e = \bigcup_{t_i} C_e(t_i)$$

Dieser Ansatz erscheint uns wegen der oben beschriebenen Annahme, dass sich schnellste Wege zu verschiedenen Zeitpunkten nur wenig unterscheiden, als sehr sinnvoll.

Implementierung An der Universität Konstanz wurden im Rahmen einer Semesterarbeit [20] eine Templateklassensammlung entwickelt, die die zielgerichtete Suche und den Containeransatz zur Verfügung stellt. Die Templateklassensammlung wurden mit der MixInProgrammierung entwickelt. Wir haben dieser Klassensammlung eine weitere hinzugefügt, die die von uns verwendeten Multikantengewichte der 96 Zeitfenster unterstützt. Die durch die Templateklassensammlung zur Verfügung gestellten Container benutzen wir bei der Berechnung kürzester Wege. Den Containeransatz zur Berechnung schnellster Wege haben wir aber nicht realisiert. Experimente für das Kürzeste-Wege Problem haben nämlich gezeigt, dass das Preprocessing zur Berechnung der Container auf sehr großen Graphen extrem zeitaufwendig ist. Auf einem Graphen mit ca. 50.000 Knoten benötigt das Preprocessing ca. 20 Minuten. Dies würde zu einem geschätzten Zeitaufwand von $96 \cdot 20\,\text{min} = 32$ Stunden führen, der sicherlich nicht vertretbar ist. Der große Zeitaufwand liegt zum einen in der Tatsache, dass das Preprocessing Zeit $O(n^2)$ benötigt, zum anderen vermuten wir, dass die versteckten Konstanen sehr gross sind, da die Template-Klassen aus sehr vielen Subklassen aufgebaut sind. Bei der Berechnung schnellster Wege setzen wir stattdessen nur die zielgerichtete Suche ein.

5.3.3 Postprocessing

Die im vorhergehenden Abschnitt beschriebene Templateklassen berechnen einen kürzesten bzw. schnellsten Weg als Folge von Knoten des zugrunde liegenden Routinggraphen. Zunächst bestimmen wir deshalb die Kanten zwischen diesen Knoten und extrahieren mit dieser Information die Datailinformationen der Kanten aus der Datenbank. Die Detailinformationen beschreiben uns exakt den zu fahrenden Weg, den wir nun, wie in Abschnitt 5.2.1 beschrieben, annotieren. Dazu berechnen wir zu jedem

Knoten die Winkel aller ausgehenden Kanten relativ zu der Kante, über die wir den Knoten erreichen. Außerdem entscheiden wir mit Hilfe des Spatproduktes (siehe [3] Kap. 1), ob die Kante, über die wir den Knoten erreichen, und die Kante, über die wir den Knoten wieder verlassen, ein Rechts- oder Linkssystem bilden. Mit den berechneten Winkeln und dieser zusätzlichen Information können wir nun bestimmen, in welche Richtung der Weg fortgesetzt wird. Das Spatprodukt ist wie folgt definiert. Seien v_1, v_2 und v_3 drei Knoten, die in die Ebene eingebettet sind. Offensichtlich spannen sie ein Dreieck auf, dessen Flächeninhalt A wir mit der folgenden Determinate berechnen können.

$$A = \begin{vmatrix} v_1(x) & v_1(y) & 1 \\ v_2(x) & v_2(y) & 1 \\ v_3(x) & v_3(y) & 1 \end{vmatrix}$$

Falls $A = 0$ ist, dann lieben die Punkte auf einer Geraden. Falls $A > 0$, dann liegt ein Rechtssystem vor, anderenfalls ein Linkssystem.

Da die PDA-Gruppe mit Hilfe dieser Informationen Sound-Dateien ausgeben möchte, die einem Fahrer ansagen, in welche Richtung er seine Fahrt fortsetzen soll, fügen wir den Abstand zur nächsten Kreuzung und die Richtung an dieser hinter jeder Kreuzung wie in Abschnitt 5.2.1 beschrieben ein.

5.4 Experimente

Wir haben unser System auf verschiedenen großen Graphen getestet und die Laufzeiten zum Einlesen des Graphens und zur Beantwortung einer Routing-Anfrage gemessen.

	Server-Setup		Weglänge	Routing-Anfrage	
	Mit Container	Ohne Container		Mit Container	Ohne Container
314 Knoten 428 Kanten	680 ms	320 ms	19 Knoten	< 1 ms	10 ms
26084 Knoten 44472 Kanten	29 min	16 s	270 Knoten	35 ms	75 ms

5.5 Zusammenfassung

Basierend auf einen GPS-annotierten Graphen haben wir ein Client-Server-Routing-Modul entwickelt, das kürzeste bzw. schnellste Wege zwischen einem Start-Ziel-Koordinatenpaar berechnet. Unser System unterscheidet sich von bisher existierenden Systemen insofern, dass wir unterschiedliche Fahrzeiten zu unterschiedlichen Zeitpunkten berücksichtigen. Außerdem erlauben wir die Aktualisierung dieser Fahrzeitinformationen über eine graphischen Oberfläche.

Literaturverzeichnis

[1] Benjamin Biedermann. Global Positioning System, Geodätische Anwendung: Vermessung des Maindreiecks. `http://home.vr-web.de/benji/gps.htm`.

[2] Arik Dasen. AIS: Ein System für die automatische Interpretation von Strassenkarten. Master's thesis, Philosophische und Naturwissenschaftliche Universität zu Bern, 2001.

[3] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[4] Department of Army. Universal Transverse Mercator Grid. Technical Report U. S. Army Technical Manual TM 5-241-8, Department of Army, 1973.

[5] Diverse. Wikipedia, The Free Encyclopedia. `http://www.wikipedia.org`.

[6] Douglas E. Comer. *Internetworking with TCP/IP. Principles, Protocols, and Architectures.* Prentice Hall, 1995.

[7] S. Edelkamp, S. Jabbar, and T. Willhalm. Geometric travel planning. In *International Conference on Intelligent Transportation Systems (ITSC)*, 2003.

[8] Institut für Geoinformatik der Universität Münster. `http://ifgivor.uni-muenster.de`.

[9] G. M. Voronoi. Novelle application des parametres continus a la theorie des formes quadratiques. *Reine und Angewandte Mathematik*, 133:97–178, 1907.

[10] Garmin International Inc. Garmin International. `http://www.garmin.com`.

[11] Rafael C. Gonzales. *Digital Image Processing*, chapter 11.1.5. Prentice Hall, 2001.

[12] GPS Gesellschaft für professionelle Satellitennavigation mbH. FUGAWI.de – Die deutschen Seiten zur FUGAWI Software (Moving Map und FTracker). `http://www.fugawi.de/index-karten.html`.

[13] Serra J. *Image Analysis and Mathematical Morphology*. Academic Press, New York, 1982.

[14] Ottmar Labonde. `http://www.ottmarlabonde.de`.

[15] Landesvermessungsamt Nordrhein-Westfalen. Cd-Rom Allgemein – Landesvermessungsamt NRW. `http://www.lverma.nrw.de/produkte/topographische_karten/cd_rom/allgemein/Cd_Rom.htm`.

[16] Landesvermessungsamt Nordrhein-Westfalen. Landesvermessungsamt NRW. `http://www.lverma.nrw.de`.

[17] MagicMaps GmbH. 3D Geovisualiserung: Videoproduktion, Webdienste, Landkarten auf CD-ROM | MagicMaps GmbH. `http://www.magicmaps.de`.

[18] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[19] Michael Wößner. Wie funktioniert GPS? Alles Wissenswerte. `http://www.kowoma.de/gps/index.htm`.

[20] Jasper Möller. Geometrische Optimierungen für Dijkstras Algorithmus. Technical report, University of Konstanz, www.inf.uni-konstanz.de/ moellerj/praktika/Bericht.pdf, 2002.

[21] Technische Unviverstät München. Semantic objects and context for road extraction. `http://www9.informatik.tu-muenchen.de/projects/roads.html`.

[22] National Marine Electronics Association. Publications and Standards from the National Marine Electronics Association (NMEA)/NMEA 0183. `http://www.nmea.org/pub/0183`.

[23] National Marine Electronics Association. The National Marine Electronics Association. `http://www.nmea.org`.

[24] Peter H. Dana. The Global Position System. `http://www.colorado.edu/geography/gcraft/notes/gps/gps_f.html`.

[25] Roger E. Sanders. *ODBC 3.5 Developer's Guide*. McGraw-Hill Osborne Media, 1998.

[26] S. Schrödl and S. Rogers and C. Wilson. Map refinement from GPS traces. Technical Report RTC 6/2000, DaimlerChrysler Research and Technology North America, Palo Alto, CA, 2000.

[27] Thales Navigation Inc. Magellan. http://www.magellangps.com.

[28] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. Technical report, Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 2003.

[29] D. Wagner, T. Willhalm, and C. Zaroliagis. Dynamic shortest path containers. Technical report, Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, Computer Technology Institute, and Department of Computer Engineering & Informatics University of Patras, 2003.

[30] Fachhochschule Wedel. http://www.fh-wedel.de/~bek/c/data/bmp.txt.