Karlsruhe Institute of Technology

Fast and Exact Mobile Navigation with OpenStreetMap Data

Diploma Thesis of

Christian Vetter

At the faculty of Computer Science
Institute for Theoretical Computer Science, Algorithmics II

Advisor: Prof. Peter Sanders

I would like to thank my advisor, Prof. Peter Sanders, for his support and useful advice. Also, I would like to thank Dennis Luxen for many informative discussion and help in getting started with OpenStreetMap data.

Finally I would like to thank Edith Brunel and Jonathan Dees as well as Simon Weinberger for rigorously proofreading this thesis, and my parents and the Begabtenstiftung Informatik Karlsruhe for the financial help they provided.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 1.3.2010

Contents

1. German Summary

Das erste Navigationssystem wurde bereits 1995 in Autos verbaut. Damalige Modelle hatten allerdings noch mit Schwierigkeiten zu kämpfen. So war die verfügbare Hardware nicht sehr leistungsfähig und Routenplanungsalgorithmen zudem noch recht fordernd. Weiterhin war die Genauigkeit von GPS Signalen für zivile Geräte stark eingeschränkt.

Im Jahr 2000 beendete die USA die Signaldegradation für zivile Geräte. Durch das Aufkommen günstiger mobiler Hardware und heuristischer Routenplanungsalgorithmen sind Navigationsgeräte inzwischen weit verbreitet.

Da die heuristischen Algorithmen zur Berechnung kürzester Wege keine kürzesten Wege berechnen, sondern nur wahrscheinlich relativ kurze Wege, stieg das Interesse an exakten und trotzdem schnellen Algorithmen. Dies führte zur Entwicklung mehrerer Ansätze, die um Größenordnungen schneller waren als traditionelle Algorithmen und sogar heuristische Lösungen übertreffen. Obwohl einige dieser Ansätze durchaus für den Einsatz in mobilen Geräten geeignet sind, sind sie zur Zeit nicht in kommerziellen System vorzufinden.

Aktuelle kommerzielle Systeme haben daher zwei hervorstechende Nachteile: Meist muss man für aktuelle Karten regelmäßig bezahlen und außerdem berechnen sie mit den verwendeten Heuristiken nur annähernd kürzeste Wege, dies meist auch noch relativ langsam. Auf der anderen Seite gibt es mit OpenStreetMap eine Ansammlung freien Kartenmaterials, das einen Qualitätsstandard erreicht hat, der es interessant für die Routenplanung macht. Allerdings beschränkt sich die zur Verfügung stehende freie Software rund um OpenStreetMap meist auf veraltete Algorithmen. So wird größtenteils noch Dijkstras Algorithmus oder A* eingesetzt. Beide sind nicht sonderlich geeignet für einen Einsatz in mobilen Geräten.

Um diese Lücke zu füllen, präsentieren wir eine freie, exakte und schnelle Navigationssoftware für mobile Geräte, z.B. PDAs und Handys. Unsere Lösung basiert dabei auf OpenStreetMap Daten und verwendet moderne Routenplanungsalgorithmen.

Da es eine hohe Bandbreite an mobilen Endgeräten gibt, nutzen wir zur Abstraktion vom unterliegenden Betriebssystem das GUI Toolkit wxWidgets. Es unterstützt unter anderem Windows, OS X, Linux, Unix, Windows Mobile, iPhone und Maemo.

Die Funktionalität ist in einer auf Plugins basierenden Modulstruktur gekapselt, um die nachträgliche Integration neuer Algorithmen zu vereinfachen. Dadurch können einzelne

Bestandteile des Programms leicht ausgetauscht werden. Dabei unterteilen wir die wichtigste Funktionalität in folgende Module: Importer, GPS, Rendering, Routing, GPS Lookup und Address Lookup. Da viele Algorithmen auf dem Prinzip der Vorberechnung beruhen, hat jedes Plugin die Möglichkeit seine Daten im Voraus zu verarbeiten. Die resultierenden Daten stehen den Modulen dann auf dem mobilen Endgerät zur Verfügung.

Das Importer Modul ist für die Aufbereitung der OpenStreetMap Daten zuständig. Da diese nicht sehr gut maschinenlesbar sind, ist damit nicht wenig Aufwand verbunden. So ist etwa der Einsatz eines kd-Baumes und effizienter Punkt-in-Polygon Algorithmen notwendig, um Straßen den entsprechenden Städten zuzuordnen. Des Weiteren sind die Daten teilweise fehlerbehaftet oder mehrdeutig, so dass das Modul recht fehlerrobust konstruiert ist.

Das GPS Modul liest die verfügbare GPS Hardware aus. Weiterhin bietet es die Möglichkeit, den Abstand zwischen zwei Punkten auf der Erdoberfläche zu berechnen. Dafür stehen zwei Approximationen zur Verfügung. Die akkuratere, aber langsamere, beruht auf der Vincenty-Formel und wird während der Vorverarbeitung der Daten verwendet. Die andere approximiert die Erdoberfläche mit einer Kugel und ist daher zwar schneller, aber auch ungenauer. Deswegen kommt sie vor allem auf dem mobilen Gerät zum Einsatz.

Das Rendering Modul ist für die Kartendarstellung zuständig. Um Konsistenz mit der OpenStreetMap Internetpräsenz zu erreichen, setzen wir den offiziellen Renderer Mapnik ein. Die gesamte Fläche, über die sich die Daten erstrecken, wird in der Vorberechnung einmal gezeichnet und in kleine Kacheln aufgeteilt. Diese Kacheln können dann zur Laufzeit benutzt werden um eine Kartenansicht zusammenzustellen. Dadurch erreichen wir einen hohen Detailgrad.

Das Routing Modul berechnet die kürzeste Route zum Ziel. Dafür haben wir eine parallele und vereinfachte Variante von Contraction Hierarchies entwickelt. Dabei werden iterativ unwichtige Knoten im Straßennetzwerk identifiziert und entfernt. Um die kürzesten Wege zu erhalten, werden künstliche Umgehungskanten eingeführt, die diese Knoten ersetzen. Nachdem hierdurch alle Knoten entfernt wurden, bilden die Originalstraßen zusammen mit den Umgehungskanten eine Hierarchie. In dieser findet man mit einer bidirektionalen Suche kürzeste Wege und kann die meisten Knoten dank der Umgehungskanten überspringen. Durch geschicktes Umordnen der Knoten lässt sich die Hierarchie kompakt speichern und ist auch in mobilen Szenarien effizient.

Das Address Lookup Modul berechnet für eine partielle Benutzereingabe Adressvorschläge. Hier kommt eine Kombination aus Prefix-Baum und Tournament-Baum zum Einsatz. Dadurch kann die Untermenge der Addressen, die dasselbe Prefix haben wie die Eingabe, schnell identifiziert werden. Daraus werden dann effizient die populärsten Addressen extrahiert. Die eingesetzte Datenstruktur ist kompatibel mit Unicode Zeichenketten, was vor allem für ausländische Adressen von Interesse ist.

Das GPS Lookup Modul benutzt eine Gitterstruktur, um zu der aktuellen GPS Position die nächstgelegenen Straßen zu finden. Auf Grund des fehlerbehafteten GPS Signals und eventuell

ungenauen Kartenmaterials müssen mehrere Kanten in einem Toleranzradius evaluiert und der beste Kandidat ausgewählt werden. Die Gitterstruktur kann platzsparend gespeichert werden und erlaubt schnelle Anfragen, hat allerdings ein paar Schwächen wenn die Daten eine sehr große Fläche abdecken.

Mit dieser Kombination an Modulen können wir die Vorberechnung für die OpenStreetMap Daten von Deutschland auf einem Desktop PC in unter 3 Stunden durchführen. Dabei ist das vom Rendering Modul eingesetzte Mapnik der limitierende Faktor. Dies macht sich noch stärker bemerkbar für die Datenmenge von Europa. Hier dominieren die ungefähr 2 Tage Renderzeit klar die restliche Bearbeitungszeit von knapp einer Stunde.

Unsere mobile Anwendung testen wir auf dem Smartphone HTC Diamond Touch 2. Trotz stark verlangsamter Flashspeicher Zugriffe zeigt sich, dass alle Komponenten performant genug sind für den praktischen Einsatz. So dauert es 15ms bis 170ms um eine Kartenansicht zu rendern, 20ms um eine lange Route zu berechnen und 3ms bis 9ms um GPS Koordinaten nachzuschlagen. Bis zu 16 Addressvorschlage können selbst im schlimmsten Fall in unter 200ms berechnet werden.

Damit gelingt es uns eine Softwarelösung vorzustellen, die es ermöglicht mobile Navigation mit modernen Algorithmen auf handelsüblichen Smartphones und PDAs zu betreiben. Besonders hervorzuheben ist, dass die eigentliche Routenplanung, trotz exakter kürzester Wege, hierbei den geringsten Aufwand erzeugt.

2. Introduction

The first automotive navigation system built into a car was sold in 1995. Early models had some challenges to face. Mobile hardware was not very powerful and routing algorithms still quite demanding. Furthermore, the limited accuracy of GPS signals for civilian use made position determination difficult.

In 2000, the USA ended the signal degradation, increasing the precision to 20m. With the advance of cheap mobile hardware and new heuristic routing algorithms satnavs almost superseded traditional routing by road atlas.

Since heuristic routing algorithms often do not compute the shortest route, but only a probably quite short route, interest in exact and fast algorithms increased. In recent years, much research went into developing such algorithms. As a result, several emerged that were orders of magnitude faster than traditional routing methods and even outperformed the heuristic algorithms. While some of these algorithms are quite suitable for mobile route planning, they have not made their way into commercial products yet.

At the same time, freely available data sets of road networks emerged and reached a stage that made them suitable for routing applications. While this lowered the bar for developing routing applications, most available software for such data sets uses outdated algorithms. This means that they either require powerful hardware or else are limited to small road networks.

In this thesis we present a mobile routing application using the freely available OpenStreetMap data set [osm10a]. We take a look at the feasibility of exact mobile routing algorithms under practical conditions.

While there exists a multitude of supplementing free software for OpenStreetMap data, as of yet none incorporate modern routing algorithms. Navit [nav10] uses Dijkstra's algorithm, Aosm [aos09] and TrackMyJourney [tra10] rely on online services for routing. We attempt to close this gap by supplying an open-source modular software suite for mobile navigation, while employing modern algorithms. This makes mobile routing applications with OpenStreetMap data practicable and might in turn result in an increase of error corrections in OpenStreetMap, which makes it even more attractive for routing purposes.

2.1. Related Work

Due to routing being the core of the application, we give a short history of related work in this category. Related work for the other integral parts is mentioned in the corresponding sections.

Because of its wide range of applications, routing is a much researched field in computer science. Early algorithms emerged in the 1950s and 60s. Dijkstra's algorithm [Dij59], Bellman Ford [Bel58] and A* [HNR07] are all optimal for arbitrary graphs with regard to asymptotic complexity.

However, most graphs encountered in practical applications have some inherent property that allows for faster routing. Especially road networks received a great deal of attention due to the large networks involved. In recent years, several *speed-up* techniques have been developed that are specifically engineered towards road networks. [DSSW09] gives an overview over successful approaches and combinations. They often trade additional pre-computation time and space consumption for faster query times.

A method common to many speed-up techniques is the *bi-directional search*. A forward search starts from the source while a backward search runs concurrently starting from the target, which essentially halves the Dijkstra search time. Several techniques do not only use this as an optional component, but it is an integral part for computing correct results.

Among the *goal directed* techniques is an approach based upon A* by [GH05]. The distance to several landmarks is precomputed and then used via the triangle inequality to enhance the potential function of A*. The resulting algorithm is called ALT.

Another goal directed approach is known as Arc-Flags [Lau04]. The graph is partitioned into several cells. Each edge is labelled with an arc vector that indicates for each cell whether this edge is on a shortest path which ends in that cell. Depending on the quality of the arc-flags, the search space is limited to relevant edges. This approach combines well with a bi-directional search to prevent fanning-out in the target cell. Also worth mentioning is the notion of multi-level arc-flags, where each cell is divided into smaller cells. This is a basic ingredient for SHARC routing [BD09].

Road networks have an inherent hierarchical property. When travelling longer distances it almost always pays off to favour better road types. Furthermore, certain junctions are used very often when leaving an area. Highway Hierarchies [SS05] make use of this by iteratively identifying important edges and bypassing unimportant nodes. The result is a hierarchy of so-called highway edges and shortcut edges. Each level of the hierarchy contains fewer edges and nodes. The further away the forward and backward search are from the source and target, the higher they can go in the hierarchy without missing the shortest path.

Contraction Hierarchies [GSSD08] [Gei08] rely entirely on bypassing, that is contraction of, nodes. Contrary to similar speed-up techniques, such as Highway Hierarchies, which only bypass a subset of the nodes, Contraction Hierarchies contract the whole graph. The resulting hierarchy contains many shortcut edges bypassing nodes. A bi-directional search then makes use of those shortcut edges, skipping most nodes in the graph in the process. This results in one of the most efficient speed up techniques. The hierarchical composition of the shortcuts can be exploited when handling a dynamic scenario. [GSSV09] shows how to deal with traffic jams, new roads and the like during the query.

A mobile variant [SSV08] capitalised on the hierarchical property and small search space of Contraction Hierarchies. This made exact and fast routing possible on mobile devices.

Contraction Hierarchies were extended to Time-Depedendent Contraction Hierarchies in [BDSV09] to cope with time-dependent edge weights. The main difficulty was how to bypass nodes efficiently, as it involves computing all-pair distances for all neighbours and all points in time. Consequently, the preprocessing was parallelised and in the process even simplified by [Vet09]. It was suggested that Contraction Hierarchies might benefit from similar alterations.

2.2. Our Contributions

In this thesis we present a routing application that uses free data sets and a modern routing algorithm. This includes many diverse tasks, ranging from importing the data and evaluating GPS signals to address lookup and the actual routing itself. Hence, a very modular design is chosen: Functionality is grouped and encapsulated into modules. This facilitates the incorporation of new algorithms and even gives the user the choice over which one to employ.

In Section 3 the overall application design is introduced. The modular structure is presented and the specific requirements for each type of module are listed. The most important modules are the importer, routing, rendering, GPS lookup and address lookup module.

Section 4 describes the reference implementation of the modules. Algorithms and data structures used are explained in detail and our method for importing the OpenStreetMap data is illustrated. Contraction Hierarchies have already proven useful in a mobile scenario and were therefore chosen as a routing algorithm. Alterations similar to the parallelisation of Time-Dependent Contraction Hierarchies are used to parallelise Contraction Hierarchies and simplify the preprocessing. The GPS lookup module uses a simple grid to locate nearby edges. A unicode trie and tournament tree are utilised by the address lookup module to present suitable suggestions to user input. The rendering module makes use of Mapnik, one of the official OpenStreetMap renderers, to raster the covered area into tiles. The Anti-Grain Geometry library is used to draw route overlays, points of interest and the like. This results in a look consistent with the official OpenStreetMap online presence.

In Section 5 experimental results are presented. The rendering module makes up the bulk of the preprocessing time, followed by the importer. While having a great compression ratio, the rendering module also takes up large amounts of storage space. That makes it unusable for continental sized graphs. A comparison between our variant of Contraction Hierarchies and the original version from [GSSD08] shows that the parallelisation and simplification save preprocessing time and memory while achieving similar query times. It turns out that the routing is very fast and suffers only slight penalties for larger data sets. The address lookup performs fast enough to keep up with the user input. The GPS lookup needs to adapt the grid size to the data set. Using the right grid sizes fast query times are achieved.

3. Design

One of the shortcomings of available routing software is that they do not use state of the art algorithms. Hence, a major design goal is to make it easy to switch between algorithms. The functionality is condensed and encapsulated into different modules. Each module communicates only through a fixed API which hides internal implementation details.

To simplify the exchange of modules a plugin-based structure is employed. For each type of module there is a factory, i.e., an object which plugins can register with in order for the main program to be able to request the creation of a module even without knowing its exact class. Additionally, a descriptive list of all plugins for a certain module is made available. The main program or the user can then choose a specific plugin at will.

Developing a plugin does not require recompilation of the main program or other plugins. Furthermore, many data structures common to related algorithms are made available, e.g., dynamic graphs, addressable priority queues and kd-trees. This helps to reduce development time for new plugins.

The second most important design goal is to ensure that porting the project to other devices is not overly complicated. To achieve this we try to keep the amount of system specific code in most modules to a minimum. On account of this system specific functionality is relegated and encapsulated in the system module. The only exception to this is the GPS module, as it needs to read out the GPS hardware.

To avoid duplication of functionality, and because the user interface usually needs to access OS-specific system calls, we opted for a cross-platform GUI library. wxWidgets [wxw10] is an open-source C++ library for 32-bit and 64-bit systems. There are ports available for Windows, OS X, Linux, Unix, Windows Mobile, iPhone and Maemo, among others. It strives for a native look, often using native GUI elements, which sets it apart from toolkits like GTK+. Furthermore a vast variety of system functionality is provided: Threads, file access, UTF-8 string support and events are the most important with respect to this thesis.

As a result, modules do not need to call OS-specific functions directly, with the exception of the GPS and system module. In spite of this, the system module is quite small. This is largely due to wxWidgets, as barely any additional functionality has to be provided.

Many modern algorithms make use of preprocessing the data to achieve faster query times. To accommodate this we give each plugin the opportunity to preprocess its data. The resulting

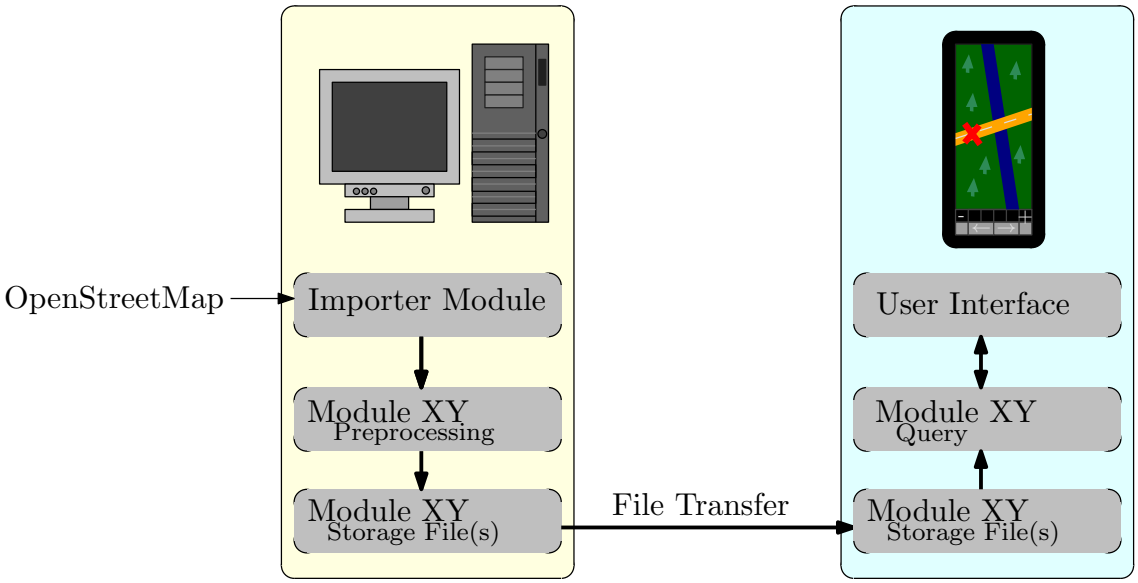


Figure 3.1.: Preprocessing / query architecture

files are then copied to the mobile device, where the query part of the plugin can make use of the precomputed data (cf. Figure 3.1).

This results in the modular design outlined in Figure 3.2. The program is divided into three layers. Each layer can only use modules in lower layers.

1. *User Interface* including the mobile variant.
2. *Main Modules*: rendering, routing, GPS lookup, and address lookup.
3. *System Abstraction*: GPS, system, and wxWidgets.

In the following, the requirements for each type of module are listed.

3.1. Importer API

The importer takes a specific data format and converts it for the use of the other plugins. Data conversion is therefore centralised, relieving the modules from the task of parsing the data themselves. To avoid duplicate parsing of complicated or fuzzy file formats the importer module is allowed to preprocess the input files and store the data in an intermediate format. Before the preprocessing of a plugin starts, the importer module is queried to deliver the appropriated data subset.

The reference implementation can import the OpenStreetMap data set.

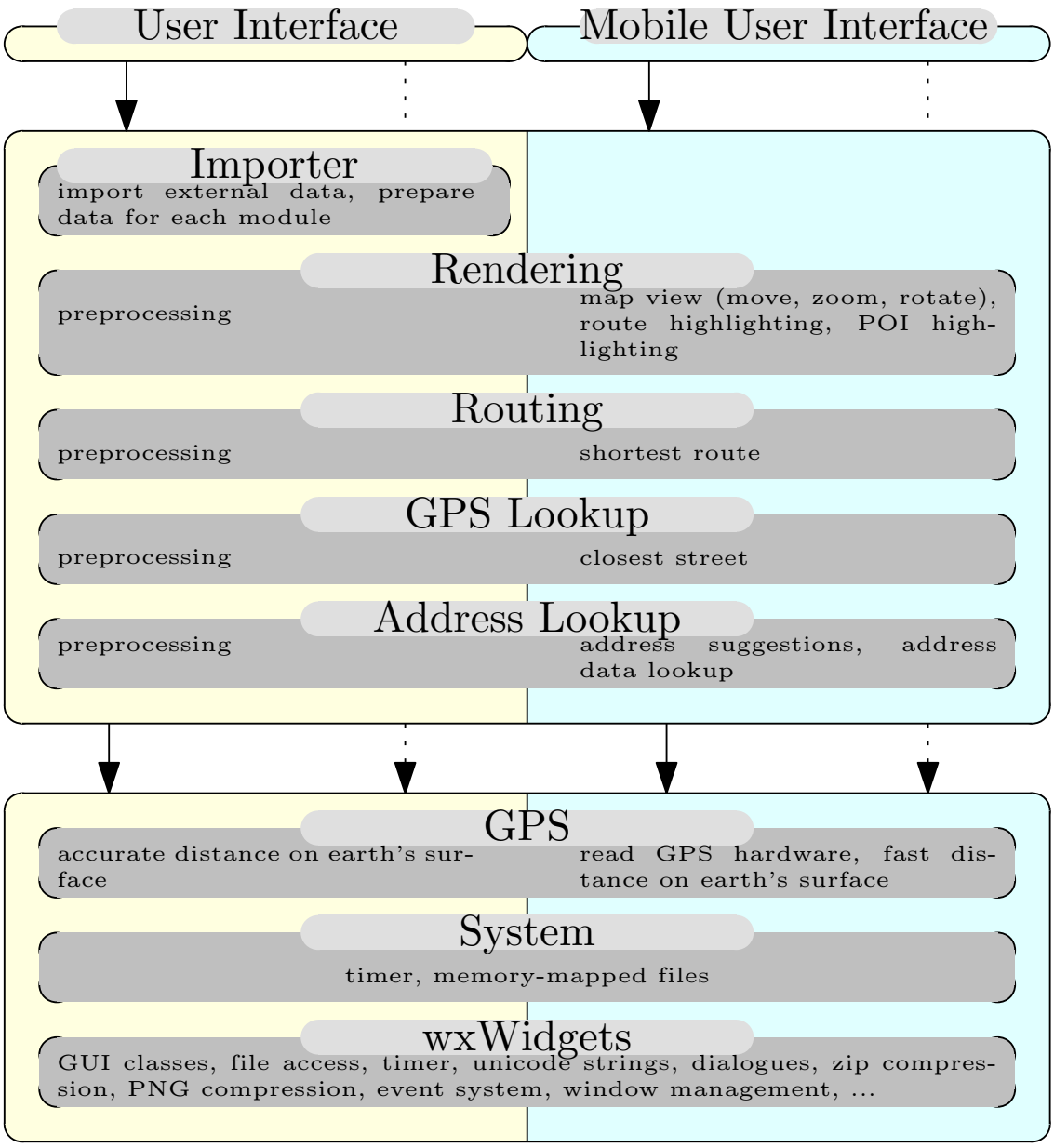
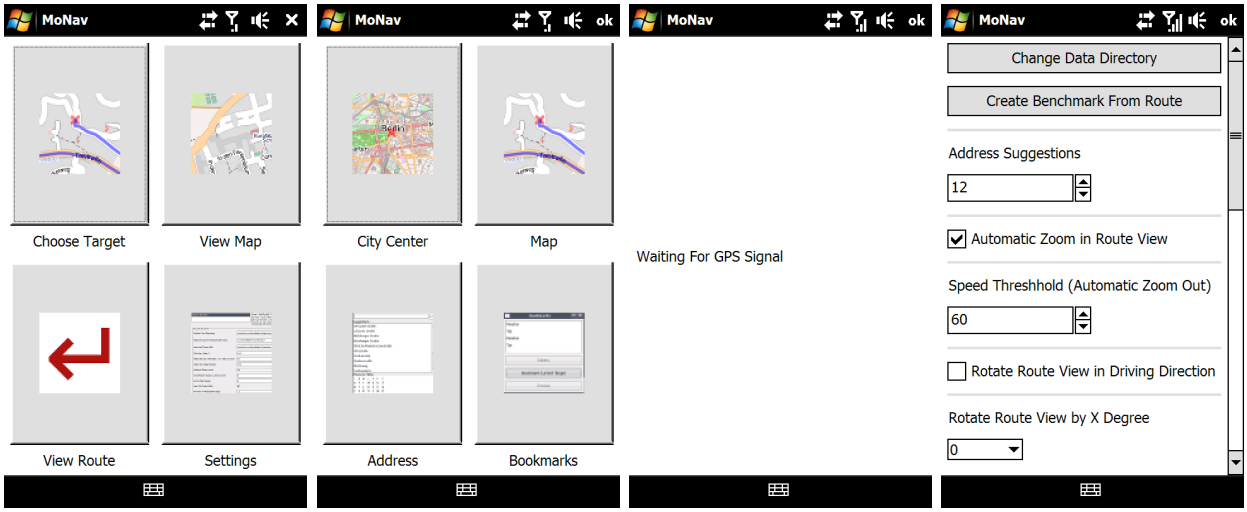


Figure 3.2.: Modular design in three layers



(a) Main menu (b) Choose target menu (c) Missing GPS Signal (d) Settings

Figure 3.3.: Various screenshots of the mobile user interface

3.2. GPS API

The GPS Module runs on a separate thread and polls the GPS hardware. Whenever new data is available, it is parsed and a wxWidgets event is sent to the specified event handler.

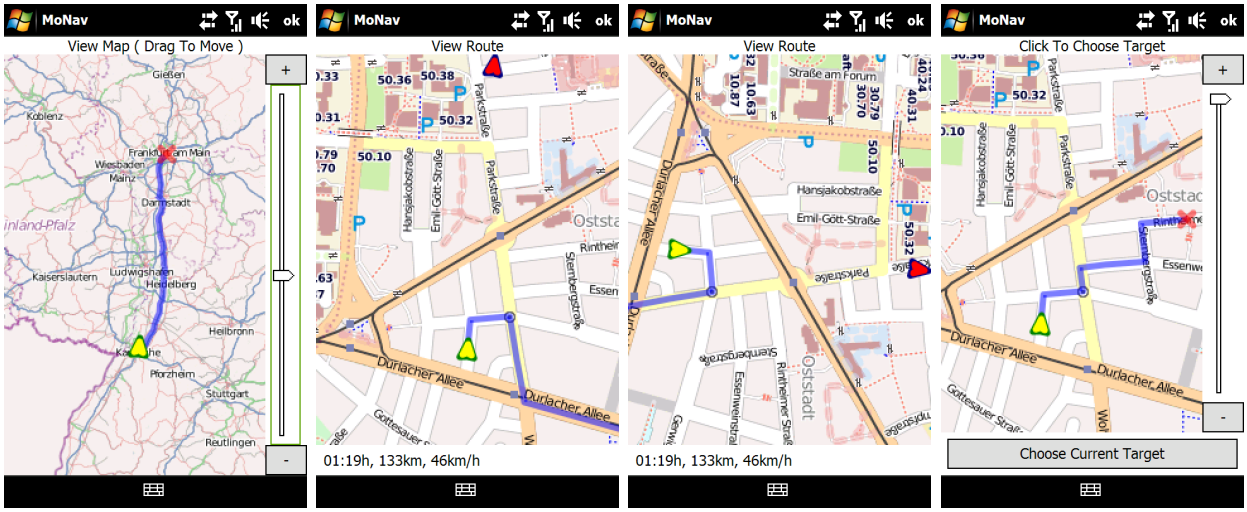
The interface itself provides methods to compute distances between two GPS coordinates. As this is a non-trivial problem, several approximations are available.

The reference implementation relies heavily on the Windows Mobile GPS drivers and is little more than a wrapper.

3.3. Rendering API

Figures 3.4(a) to 3.4(d) show various areas of application for a rendered map: The user can zoom and scroll the map, the route is displayed for driving instructions and possibly rotated, or a target is chosen directly via the map. These define our requirements for the rendering module.

The module provides a map view for several zoom levels. The main application requests an image centred on a specific GPS position and the plugin notifies the main application when the image is ready, enabling asynchronous rendering. The rendering module is required to



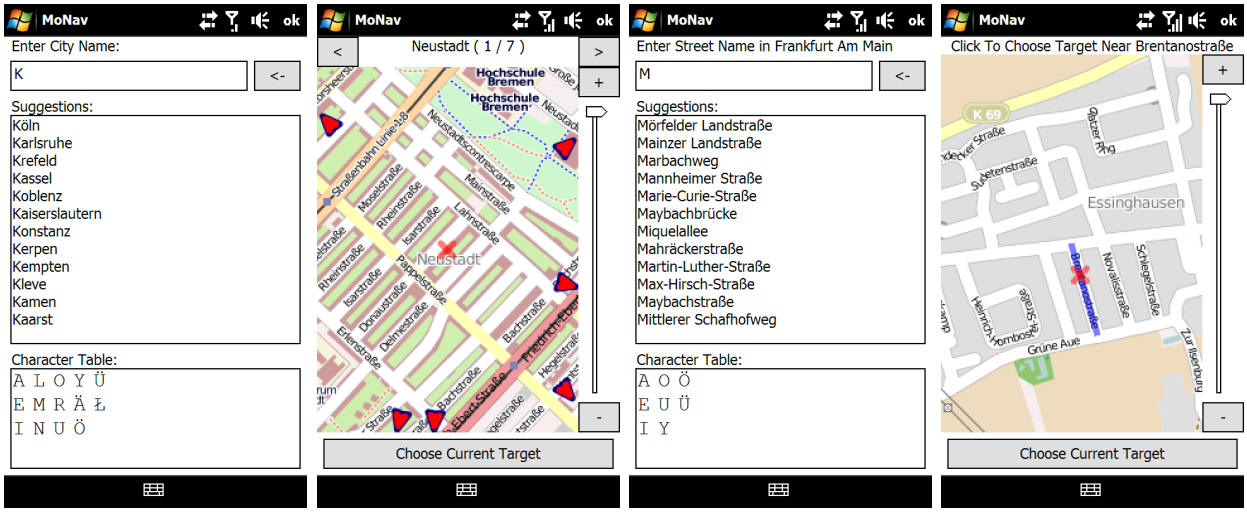
(a) Zoomed out map view (b) Route view (c) Rotated route view (d) Choose target via the map view

Figure 3.4.: Screenshots of applications of the rendering output

highlight given positions or streets, or rotate the view by a specified amount. In addition to this, the plugin needs to convert a position in the map view to a GPS coordinate to enable user input.

Most operating system use their own native bitmap format. In many cases only bitmaps in those formats can be copied directly onto the display. For the rendering module to be device-independent, it has to return a device-independent bitmap. The user interface, in this case wxWidgets, has the responsibility to convert this bitmap into a device specific format for displaying. This adds additional overhead for the conversion, but greatly increases portability of the rendering module: The rendering libraries do not have to worry about the device specific format.

Since we aimed for a high level of detail in the displayed maps, the reference implementation employs Mapnik, one of the official OpenStreetMap renderers. This also has its drawbacks: Mapnik is too slow and resource hungry to run on a mobile device. Therefore, we opted to pre-render most imagery in a manner similar to GoogleMaps' and OpenStreetMap's websites. This makes the reference implementation a raster map renderer.



(a) City selection

(b) Choosing between several cities with the same name

(c) Street selection

(d) Choosing target location on the street

Figure 3.5.: Screenshots of choosing the target via address input

3.4. Routing API

Given two positions on specific edges the routing module has to compute the shortest path between them. Paths should be output as a sequence of coordinates. The implementation can take advantage of similar queries as the target rarely changes during driving.

Because Contraction Hierarchies delivered good results in mobile scenarios, it was chosen for shortest path computation in the reference implementation. The algorithm is modified to use a simplified and parallelised preprocessing step.

3.5. Address Lookup API

We chose a somewhat inflexible model for parsing address input. Figures 3.5(a) to 3.5(d) depict a typical scenario: First, the user is required to input a city name. For each partial input, that is, after each character entered or deleted, a list of suggestions is generated. Once the desired destination city is included in this list, it can be selected. If several cities use the chosen name a map view is displayed, which allows the user to identify the correct one. Next, a street has to be chosen. Again, a list of suggestions is generated for the user input. At the time of writing, the use of street numbers is not widely spread in OpenStreetMap

data. Because we used OpenStreetMap as our sole data source, we opted to leave out street numbers for now. Therefore, each address is associated with the whole street, instead of its exact location. As a temporary workaround the user is presented with a map view and the option to pick a position on the street.

We can now specify the requirements for the address lookup module. Given a partially entered city or street name, the module has to assemble a list of suggestions. When a city name suggestion is chosen, the coordinates and identifiers of all cities with said name have to be returned. Given a city ID and a partial input of a street name, a suggestions list has to be generated. After choosing a street name suggestion, a set of coordinates describing the course of the street have to be returned.

The reference implementation uses a combination of unicode tries and tournament trees to compute suitable suggestions.

3.6. GPS Lookup API

The GPS lookup module has to find the closest position on routing edges in the vicinity of a given GPS coordinate. All edges within a certain distance have to be returned, as well as their nearest point. The main application may choose to disregard the actual closest edge and pick one depending on the orientation or the current route instead.

The reference implementation simply divides the area into a grid, distributing the edges accordingly. This has disadvantages for huge areas, but the raster renderer is the limiting factor in this regard.

3.7. System API

The system module provides various missing system specific functionality, such as memory mapped files and timers. While timers are included in wxWidgets, they have a low resolution on some devices, e.g., 1000ms on the Windows Mobile port.

4. Modules

In this section we describe each module in detail: First, we give a short reminder of the requirements. Next, important data structures and algorithms employed are introduced. This is followed by a description of the preprocessing of the data and data structures used to store the result. Finally, the processing of queries from the main program is presented.

4.1. Importer

The importer module processes OpenStreetMap data, converting it for use by other plugins. A kd-tree and a point-in-polygon algorithm are used to determine each city's administrative area.

4.1.1. OpenStreetMap

OpenStreetMap is a collaborative project to create a free editable map of the world. Over the last years accuracy and coverage have greatly improved, making it one of the best freely available GIS[1] data sets. The entire database, as well as country specific subsets, are available as XML files.

OpenStreetMap stores its data with only three data primitives:

- *Nodes* (cf. Figure 4.1): Nodes consist of an ID, a latitude and a longitude. They are used to represent points of interest or as part of the *way* primitive, and as such can be considered basic building blocks. Consequently every other data primitive depends on them.
- *Ways* (cf. Figure 4.2): A way consists of an ID and a sequence of nodes. When the first and last node are identical it is called a *closed way* and represents an area.
- *Relations*: A relation consists of an ID and several members with role attributes. A member can be any data primitive. Relations are used to group polygons into multi-polygons, or to define turning restrictions, among other things.

[1] Geographic information system

```
<node id="506456462" version="1" timestamp="2009-09-24T06:53:58Z"
    uid="16330" user="aluka" changeset="2602035" lat="49.0325173"
    lon="8.3603005">
  <tag k="boundary" v="city_limit"/>
  <tag k="is_in" v="de"/>
  <tag k="name" v="Karlsruhe"/>
  <tag k="traffic_sign" v="city_limit"/>
</node>
```

Figure 4.1.: OpenStreetMap node

```
<way id="3142524" version="6" timestamp="2009-01-29T07:53:36Z" uid="3112"
    user="HDietze" changeset="850890">
  <nd ref="10919224"/>
  <nd ref="269570193"/>
  <nd ref="85899428"/>
  <nd ref="25760258"/>
  <nd ref="85896230"/>
  <nd ref="10919223"/>
  <tag k="created_by" v="Potlatch 0.10f"/>
  <tag k="highway" v="unclassified"/>
  <tag k="maxspeed" v="50"/>
  <tag k="name" v="Erich-Kaestner-Strasse"/>
</way>
```

Figure 4.2.: OpenStreetMap way

Each data primitive can also be associated with a list of tags. A tag is a (key,value) pair. A list of community accepted tags regulates which tagging standards should be used to represent real world data. However, many tagging schemes in use are only proposed schemes, and as such currently still under discussion. Therefore, it is possible to have several competing or fluctuating systems in use to represent the same set of data. Also, machine readability is not the main focus behind the design of many tags. This makes parsing certain tags quite difficult.

As correct tags are not enforced, they have to be evaluated with care. A user may input arbitrary data for each tag. In Table 4.1 a list of examples for illegal or difficult to interpret tags can be seen. These are actual tags currently in use by the German subset of the OpenStreetMap data.

A good introduction to working with or contributing to OpenStreetMap is given in [RT].

Key	Value
maxspeed	yes
maxspeed	0,80
maxspeed	70; 50; 100
maxspeed	.5
oneway	yes:0h-4h;yes:12h-24h;-1:4h-12h
oneway	soon;signs already standing
oneway	no;yes
highway	footway, stairs, bycicle ramp, not passable by car.
highway	Zufahrt zum Anglergrundstück
population	1655 in 8 Ortsteilen!

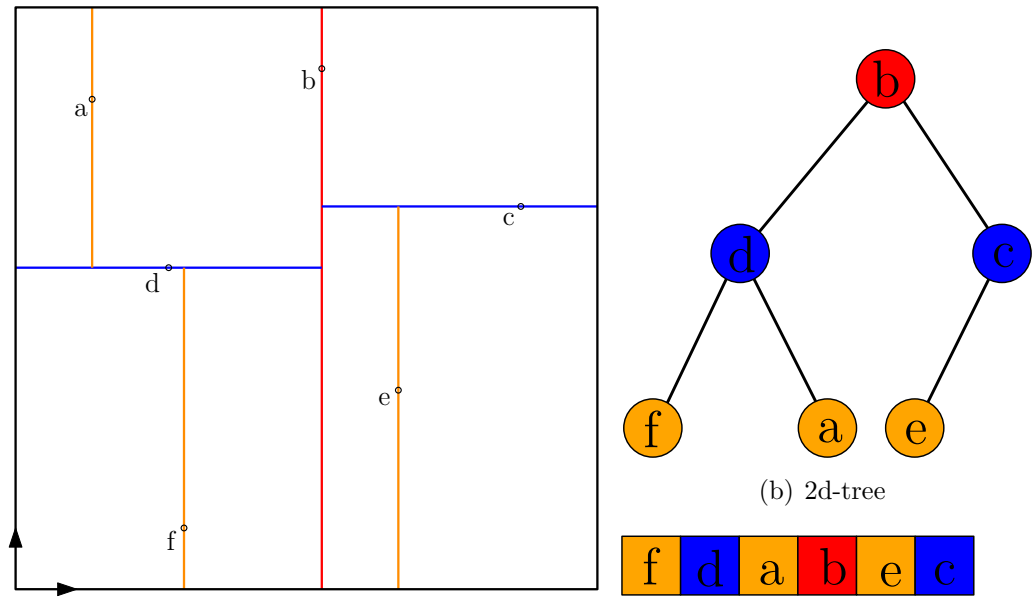
Table 4.1.: Illegal or difficult to interpret tags found in OpenStreetMap data

4.1.2. kd-tree

A kd-tree is a space partitioning data structure for organizing n points in a k-dimensional R space [Ben75] [LW77] [Ben90]. It is a binary tree that can be used to answer range or nearest-neighbour queries. Each non-leaf node splits the space of the sub-tree along one of the k dimensions. The left sub-tree contains the points to left of the hyperplane, the right sub-tree the ones on the right side. Performance and space consumption depend on the choice of the hyperplane for each node.

Construction. There are several ways to construct a kd-tree, since we are free to choose the hyperplane that splits the space of a sub-tree. The common way to construct a kd-tree is as follows: The depth of a sub-tree determines the splitting dimension. When descending in the tree, we simply alternate the dimension in a predefined order. For each sub-tree the median of all points in the sub-tree along the splitting dimension is determined. The splitting hyperplane then goes through the median. Child nodes are inserted until each leaf node contains exactly one point. This results in a balanced binary tree. Figure 4.3(a) shows the chosen hyperplanes for the 2d-tree in Figure 4.3(b).

Picking the median allows us to establish an implicit kd-tree that does not need to store much extra information. The whole structure of the kd-tree can be encoded in the order of the points: Each node is represented by a sequence $(p_i, \ldots, p_{\lceil (j+i)/2 \rceil}, \ldots, p_j)$, where $p_{\lceil (j+i)/2 \rceil}$ denotes the median. $p_{\lceil (j+i)/2 \rceil}$ and the depth of the node determine the splitting plane. $(p_i, \ldots, p_{\lceil (j+i)/2 \rceil - 1})$ are left of the splitting plane, while $(p_{\lceil (j+i)/2 \rceil + 1}, \ldots, p_j)$ are right of the splitting plane. The source node is represented by $(p_1, \ldots, p_n)$. When starting from the source node we can traverse the whole tree. The only additional data we store is the bounding box of the entire data set. The implicit version of the 2d-tree in Figure 4.3(b) is depicted in Figure 4.3(c).



(a) Hyperplanes dividing the space in a 2d-tree

(b) 2d-tree

(c) Implicit 2d-tree

Figure 4.3.: 2d-tree

Selecting the median in linear time requires somewhat complex algorithms, but many programming languages include an efficient implementation in their libraries. C++ for example provides nth_element in the STL, which can select the median and also split the points in the required way. This makes constructing an implicit kd-tree quite simple, as seen in Listing 1.

Range Query. A range query RangeQuery(r, p) returns all points whose distance to p is less than r. We need a metric that performs two types of distance calculation in R^k:

1. The distance between a point q and p
2. The smallest distance any point in the hyperrectangle of a sub-tree can have to p

In most cases the Euclidean distance is used, but an arbitrary metric can be applied. The query itself is quite simple, as seen in Listing 2: We perform a depth-first search, only considering sub-trees if the distance of their hyperrectangle to p is less or equal than r. For each point we encounter, we check whether the distance to p is smaller than r. Obviously, all necessary points are inspected.

For uniformly or randomly distributed points and small values of k the range query performs quite well in praxis. Depending on r, the metric used, and the point distribution, it is possible that all points in the tree have to be considered in spite of the resulting point set being quite small.

Algorithm 1: kd-tree Construction

Data: points p = $(p_1, \ldots, p_n)$, dimension k
Result: implicit kd-tree

```
1  stack s;
2  Push(s,(1,n,0));
3  while NotEmpty(s) do
4  |  (left,right,depth) ← Pop(s);
5  |  if right − left ≤ 1 then
6  |  |  continue;
7  |  end
8  |  median ← ⌈(right + left)/2⌉;
9  |  NthElement(p,left,right,median,CompareByDimension(depth % k));
10 |  Push(s,(left,median − 1,depth + 1));
11 |  Push(s,(median + 1,right,depth + 1));
12 end
13 return p;
```

Algorithm 2: kd-tree Range Query

Data: node n = root, point p, range r
Result: $P = \{q \mid \texttt{Distance}(q,p) \leq r\}$

```
1  P ← ∅;
2  rect ← HyperRectangle(n);
3  if Distance(rect, p) ≤ r then
4  |  if Distance(Median(n),p) ≤ r then
5  |  |  P ← P ∪ {Median(n)};
6  |  end
7  |  if HasLeftChild(n) then
8  |  |  P ← P ∪ RangeQuery(LeftChild(n),p,r);
9  |  end
10 |  if HasRightChild(n) then
11 |  |  P ← P ∪ RangeQuery(RightChild(n),p,r);
12 |  end
13 end
14 return P;
```

Algorithm 3: kd-tree Nearest-Neighbour Query

Data: node n = source, near point q = source, point p, range r = ∞
Result: (nearest point q, distance r)

```
1  rect ← HyperRectangle(n);
2  if Distance(rect, p) < r then
3  |  if Distance(Median(n),p) < r then
4  |  |  r ← Distance(Median(n),p);
5  |  |  q ← Median(n);
6  |  end
7  |  if HasNearChild(n) then
8  |  |  (q,r) ← NearestNeighbourQuery(NearChild(n),q,p,r);
9  |  end
10 |  if HasFarChild(n) then
11 |  |  (q,r) ← NearestNeighbourQuery(FarChild(n),q,p,r);
12 |  end
13 end
14 return (q,r);
```

Nearest Neighbour Query. A nearest neighbour query NearestNeighbour(p) returns one of the points with minimal distance to p. It is similar to the range query, as can be seen from Listing 3: We start with an infinite range and update it whenever we find a closer point. While descending in the tree we choose the nearest sub-tree first. The nearest sub-tree usually contains points that are closer to p and are therefore helpful in reducing the range.

For n uniformly or randomly distributed points and small values of k the nearest neighbour query is answered in $\mathcal{O}(\log(n))$. As was the case with the range query, problem instances can be constructed which require $\mathcal{O}(n)$ points to be investigated.

4.1.3. Point In Polygon

There are several algorithms to determine whether a point p is in a given polygon r. The most common one is the ray casting algorithm given in Listing 4 [Shi62]. An infinite ray starting from p is cast and the amount of intersections with the polygon r is counted. If the number of intersections is even, p is located on the outside of r, otherwise it is on the inside. Because it simplifies the computation the ray is always cast upwards. A check for intersection is performed for every edge of the polygon. Two special cases have to be considered: First, the ray may pass directly through a vertex. In this case only one intersection must be counted, which can be solved by only allowing intersections with the the right vertex of each edge. Second, the ray may lie directly on an edge, in which case no intersection must be counted.

A good summary of point-in-polygon algorithms is presented in [Hai94].

Algorithm 4: Ray Casting Algorithm

Data: t coordinate of the query point, P polygon
Result: t within P? $\in$ {true,false}

```
1  inside ← false;
2  i ← size(P) −1;
3  for j ← 0 to size(P) do
4  |  if (P[j]_x > t_x) ≠ (P[i]_x > t_x) then                    // edge crosses line t_x
   |  |  if t_y < ((t_x − P[j]_x)/(P[i]_x − P[j]_x))(P[i]_y − P[j]_y) + P[j]_y then   // edge is above t
5  |  |  |  flip(inside);
6  |  |  end
7  |  end
8  |  i ← j;
9  end
10 return inside;
```

4.1.4. Preprocessing

All modules, except for the renderer, use only a small subset of the data: The GPS lookup and routing module operate solely on the routeable nodes and edges and the address module only requires specific places and corresponding routeable edges. To avoid processing unnecessary data the XML file is preprocessed in two phases. The libxml2 library [lib10] is used to access the XML file via the XmlTextReader interface to minimise the memory footprint while walking the XML tree.

Phase 1. All routeable ways are determined and extracted, as well as country and place boundary ways. Places which are of interest to the address module are also stored. Then node lists are created for both the routeable ways and the boundaries.

Phase 2. The ways and boundaries extracted in phase 1 are still missing the coordinates of the involved nodes. These are read in a second pass. The boundaries are then used to associate each node with a country. Furthermore, it is determined for each city which subset of nodes is located within its boundaries. The method used depends on whether a boundary polygon is provided for the city in question. First, all nodes are inserted into a kd-tree.

1. If a boundary polygon is given, a perimeter of the city node which contains the polygon is computed. This perimeter is used to query all nodes from the kd-tree that might fall within the city's boundary. To check whether a node actually lies within the boundary the ray casting algorithm is used.

2. If no boundary polygon is provided, it is checked whether a radius is given for the city node. Since most city nodes have no radius tagged, a default value is used depending on the node's type. This radius is used to request all nodes within the perimeter from the kd-tree and associate them with the city.

This information is required by two modules. The address module needs to know which city a street belongs to. The routing module needs an average speed associated with each edge. But most edges are not tagged with speed information and therefore default speed values according to the street type have to be used. Most countries employ different default speed values for streets within city boundaries, in which case it is necessary to know whether a street belongs to a city or not.

In addition to these two cases, OpenStreetMap defines another alternative of associating a way with a city: The city node is in an area that is tagged with specific *landuse* values and the way resides entirely in the union of that area and all intersected or nearby areas of similar type. This is difficult to check for and not a very accurate definition, prompting us to omit testing for it.

Most nodes are not needed for routing, as they are not adjacent to a routeable edge, e.g., when they are used to compose non-accessible roads or geographical features like woods. We remap the IDs of nodes used for routing into a contiguous range starting from 0, so the GPS and routing module only have to deal with a compact node ID space.

4.2. GPS

The GPS module provides the means to read out the GPS hardware and compute distances between two geodesic points.

4.2.1. GPS Distance Computation

As the surface of the Earth is only a 2-dimensional manifold, computing the distance between two points on the surface is not as straightforward as for a 2-dimensional plane. A point $p = (\alpha, \beta)$ on the surface of the Earth is given by its latitude α and longitude β. We use the following two approximations to compute the distance between $p = (\alpha, \beta)$ and $q = (\gamma, \delta)$.

Spherical Approximation. When approximating the Earth's surface with a sphere we can use the haversine formula to compute the distance. The haversine of x is defined as in Formula 4.1.

$$\text{haversine}(x) = \sin^2\left(\frac{x}{2}\right) \tag{4.1}$$

For every two points $p = (\alpha, \beta)$ and $q = (\gamma, \delta)$ on the surface of a sphere the haversine formula 4.2 holds true.

$$\text{haversine}\left(\frac{\text{distance}}{\text{radius}}\right) = \text{haversine}(\alpha - \gamma) + \text{haversine}(\beta - \delta)\cos(\alpha)\cos(\gamma) \tag{4.2}$$

By using the inverse sine we can also solve Formula 4.3 for the distance, resulting in Formula 4.4.

$$\text{haversine}\left(\frac{\text{distance}}{\text{radius}}\right) = \sin^2\left(\frac{\text{distance}}{2 * \text{radius}}\right) \tag{4.3}$$

$$\text{distance} = 2 * \text{radius} * \arcsin\sqrt{\text{haversine}\left(\frac{\text{distance}}{\text{radius}}\right)} \tag{4.4}$$

Combining this with the haversine formula 4.2, we can compute the distance with Formula 4.5.

$$\text{distance} = 2 * \text{radius} * \arcsin\sqrt{\sin^2(\frac{\alpha - \gamma}{2}) + \sin^2(\frac{\beta - \delta}{2})\cos(\alpha)\cos(\gamma)} \tag{4.5}$$

This approximation requires a total of 5 trigonometric functions. The result usually has an error of about 0.3% on the surface of the Earth. This makes it suitable to be employed on the mobile device. More precisely, the GPS lookup module uses it to compute the closest edges and the main application to approximate the length of the current route.

WGS84 Ellipsoid Approximation. The standard approximation manifold in cartography and navigation is the WGS84 ellipsoid. The earth geoid can be better approximated by an ellipsoid than a sphere. Computing distances on the surface of an ellipsoid, however, is not as trivial. The common method is to use Vincenty's Formulae [Vin75] to iteratively compute the distance until it converges. The result has an error of less than 5mm on the WGS84 ellipsoid. Distances between nearly antipodal points cannot be computed correctly, though.

As seen in Listing 5, only three trigonometric functions per iteration are necessary: $\cos\lambda$, $\sin\lambda$ and $\arctan(\frac{\sin\sigma}{\cos\sigma})$. Despite this, the WGS84 approximation is much slower than the spherical approximation. While this makes it unsuitable for use on the mobile device, it is the first choice for distance calculation during the preprocessing. The importer module makes extensive use of it since OpenStreetMap data only specifies the end points of an edge, not the actual metric length itself.

Algorithm 5: Vincenty's Inverse Formula

Data: a= major semiaxis, b = minor semiaxis, α, γ, L = $\beta - \delta$

Result: distance $|p, q|$

$f \leftarrow \frac{a-b}{a}$;

$U_\alpha \leftarrow \arctan((1-f)\tan\alpha)$;

$U_\gamma \leftarrow \arctan((1-f)\tan\gamma)$;

$\lambda \leftarrow L$;

while λ *not converged* **do**

  $\sin\sigma \leftarrow \sqrt{(\cos U_\gamma \sin\lambda)^2 + (\cos U_\alpha \sin U_\gamma - \sin U_\alpha \cos U_\gamma \cos\lambda)^2}$;

  $\cos\sigma \leftarrow \sin U_\alpha \sin U_\gamma + \cos U_\alpha \cos U_\gamma \cos\lambda$;

  $\sigma \leftarrow \arctan(\frac{\sin\sigma}{\cos\sigma})$;

  $\sin\phi \leftarrow \frac{\cos U_\alpha \cos U_\gamma \sin\lambda}{\sin\sigma}$;

  $\cos^2\phi \leftarrow 1 - \sin^2\phi$;

  $\cos(2\sigma_m) \leftarrow \cos\sigma - 2\frac{\sin U_\alpha \sin U_\gamma}{\cos^2\phi}$;

  $C \leftarrow \frac{f}{16}\cos^2\phi(4 + f(4 - 3\cos^2\phi))$;

  $\lambda \leftarrow L + (1-C)f\sin\phi(\sigma + C\sin\sigma(\cos(2\sigma_m) + C\cos\sigma(2\cos^2(2\sigma_m) - 1)))$;

end

$u^2 \leftarrow \frac{\cos^2\phi(a^2-b^2)}{b^2}$;

$A \leftarrow 1 + \frac{u^2}{16384}(4096 + u^2(-768 + u^2(320 - 175u^2)))$;

$B \leftarrow \frac{u^2}{1024}(256 + u^2(-128 + u^2(74 - 47u^2)))$;

$\Delta\sigma \leftarrow B\sin\sigma(\cos(2\sigma_m) + \frac{1}{4}B(\cos\sigma(2\cos^2\sigma_m - 1) - \frac{1}{6}B\cos(2\sigma_m)(4\sin^2\sigma - 3)(4\cos^2(2\sigma_m) - 3)))$;

return $bA(\sigma - \Delta\sigma)$;

4.2.2. Query

The GPS plugin is OS-specific and therefore not portable at all. Our implementation calls the Windows Mobile GPS Intermediate Driver [gps10] to extract the current position, speed and heading. This driver supplies already parsed GPS information, making the plugin little more than a wrapper.

4.3. Rendering

The rendering module provides a top-down map view. Different zoom levels are available, routes and locations are highlighted and the map can be zoomed. Mapnik is used to prerender the whole data set during preprocessing. The Anti-Grain Geometry library is employed to compose the map view and draw highlights.

4.3.1. Mapnik

Mapnik [map10] is an open-source C++ mapping toolkit, which is used as one of the official OpenStreetMap renderers. It has several input plugins. At the time of writing two render backends are available: CAIRO and Anti-Grain Geometry. How the map is rendered can be defined freely via stylesheets. Features can be arranged in layers and restricted to certain zoom levels.

OpenStreetMap uses an ESRI Shapefile in combination with a PostGIS PostgreSQL database to make its data available to Mapnik.

The shapefile is used for the coastlines, which are represented by very large closed polygons. Parts relevant for a specific area cannot be extracted efficiently from the OpenStreetMap data. Using the shapefile allows for considerably faster extraction. For the lower zoom levels 0 to 9 a low resolution shapefile is used.

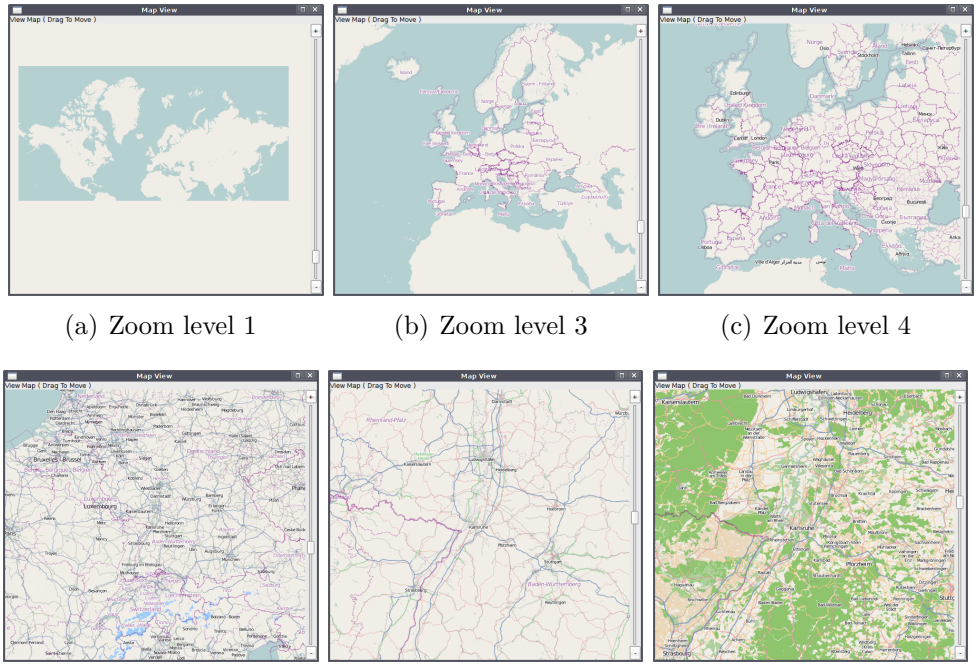
PostgreSQL [pos10b] is an open-source object-relational database management system. PostGIS [pos10a] adds support for geographic objects to PostgreSQL, using R-tree-over-GiST spatial indexes to enable high speed spatial querying. This makes it possible for Mapnik to query only the OpenStreetMap data subset relevant to the rendered image.

Examples of features displayed at various zoom levels can be seen in Figures 4.4 and 4.5.

4.3.2. Preprocessing

The OpenStreetMap XML file is imported into a PostGIS PostgreSQL database using the OpenStreetMap utility program osm2pgsql [osm10b]. Furthermore, a shoreline shapefile, symbols, fonts and the stylesheet used by OpenStreetMap have to be provided. The Mercator projection is used to transform the coordinates.

For each requested zoom level the entire area of the data set is rendered. Larger areas cannot be rendered all at once, though. Instead, the area is divided into large meta-tiles which are rendered one by one. These meta-tiles overlap by a margin to avoid some discrepancies: Mapnik moves labels in a way that prevents them from crossing the image borders. Consequently, the final image would look awkward as some labels might be displayed twice, once on each side of the meta-tile border. The meta tiles are then cut into smaller tiles and the margin is discarded in the process.



(a) Zoom level 1 (b) Zoom level 3 (c) Zoom level 4

(d) Zoom level 6 (e) Zoom level 8 (f) Zoom level 9

Figure 4.4.: Zoom levels 1 to 9

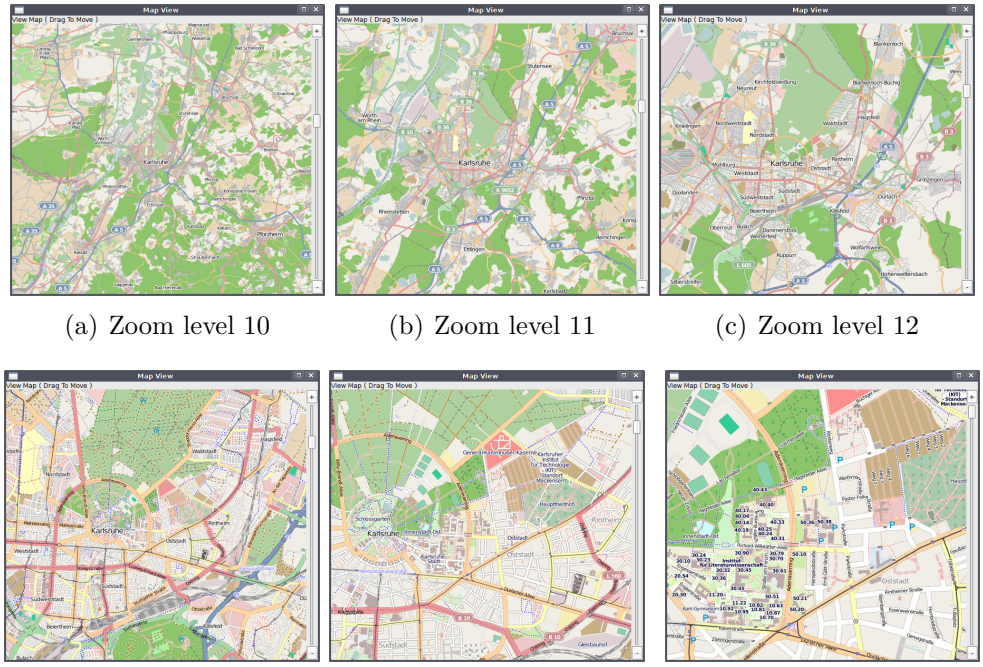
4.3.3. Data Structures

Each tile is then compressed into PNG image format. Optionally, each tile can be reduced to 256 different colours beforehand. The reduced colour space is usually sufficient, even with anti-aliased edges. Tiles are stored according to their zoom level in a large tile file with a separate index file.

Tiles without any routeable streets in the vicinity can be skipped: For each zoom level, the tiles, which will have edges rendered on them, are computed, storing a bit for each tile to indicate whether it should be skipped. Nevertheless, skipped tiles still require space in the index file. While this filters out the majority of tiles for higher zoom levels, it actually saves very little storage space: The skipped tiles usually feature little variety and are compressed very well by the PNG compression scheme.

4.3.4. Query

The mobile device needs to compose the image from tiles, display the route and highlight some points on the map. The image can be rotated and, in order to be consistent with the Mapnik



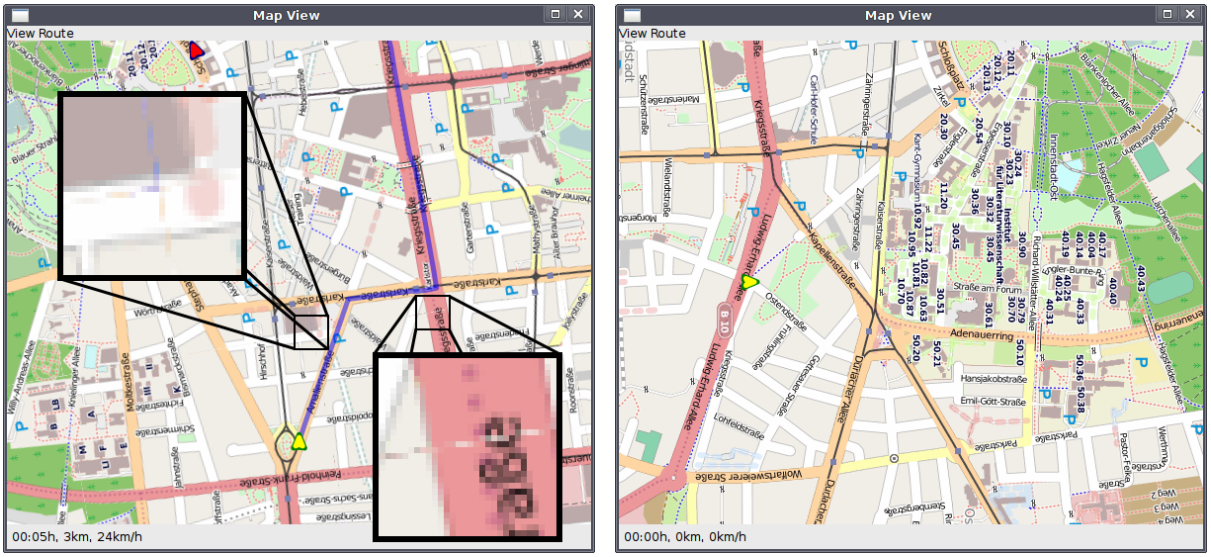
(a) Zoom level 10 (b) Zoom level 11 (c) Zoom level 12

(d) Zoom level 13 (e) Zoom level 14 (f) Zoom level 15

Figure 4.5.: Zoom levels 10 to 15

imagery, the route should be anti-aliased. As it was not the focus of this thesis to duplicate such functionality, we chose to rely on an existing rendering library. Portability and speed were the main concerns, since most libraries are either limited to one or very few operating systems, or quite slow. We employed the Anti-Grain Geometry library [ant10], which is a fast C++ template library and completely device-independent. Because it is also available as a output module for Mapnik we can achieve a consistent look.

A simple LRU replacement strategy is used for the tile cache. When an image is requested, the module first checks whether the necessary tiles are present in the cache, and loads them if necessary. Each tile is rendered directly on its rotated position in the final image. This represents an affine transformation of the image and can be handled very efficiently with little memory overhead. However, artefact glitches on the tile borders cannot be prevented without memory and performance penalties, since the image would have to be composed before rotating it. An example for such artefacts is given in Figure 4.6(a). It also becomes apparent that the rotation algorithm used, while being quite fast, slightly blurs the image. The artefacts vanish when rotating by multiples of 90°, as shown in Figure 4.6(b). This makes it easy to adjust the image to the orientation of the mobile device without significant loss of quality.



(a) Rotation artefacts

(b) 90° rotation

Figure 4.6.: Rotation

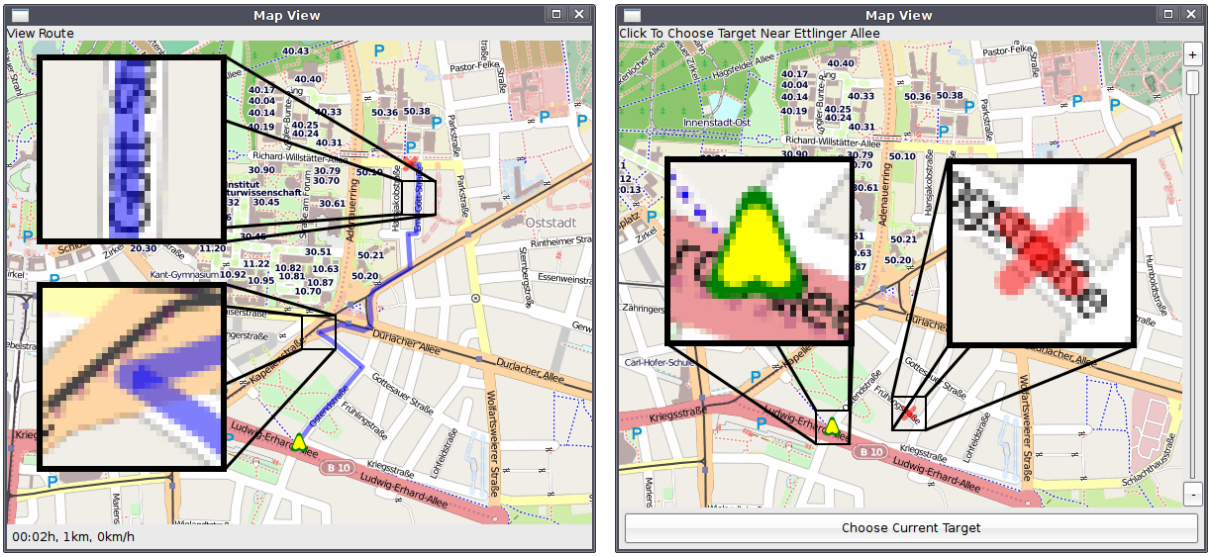
For drawing the route overlay, the route is first clipped to the image boundaries. Details are omitted whenever too many points fall within the requested image, and the lines composing the route are drawn half-transparent and anti-aliased. The route is not checked for self-intersections, as this would be quite computation-intensive for longer routes. This produces artefacts whenever the route intersects itself or sharp corners are drawn. An example is given in Figure 4.7(a). Finally, both the current position and highlighted targets are drawn with anti-aliased polygons, as can be seen in Figure 4.7(b).

4.4. Routing

The routing module computes the shortest path in the road network. We developed a new variant of Contraction Hierarchies for use in this module. Contraction Hierarchies itself make use of Dijkstra's algorithm and bidirectional search.

4.4.1. Dijkstra's Algorithm

Given a directed graph $G = (V, E)$ with static edge weights $w : E \mapsto \mathbb{R}^+$, Dijkstra's algorithm (cf. Listing 6) computes the shortest path between a source node s and target node t. The



(a) Route

(b) Position and target

Figure 4.7.: Highlighting

break condition in line 6 can easily be modified to compute shortest paths and shortest path distances from s to several target nodes. While the path itself can be reconstructed from the *parent* function by walking backwards from the target, this function need not be computed at all if only the shortest path distance itself is of interest.

We distinguish between 3 states for each node:

- *Undiscovered*: The node has not been inserted into the priority queue yet.
- *Reached*: The node was inserted into the priority queue and has not been deleted yet. Its distance is only tentative and may change.
- *Settled*: The node has already been removed from the priority queue. Its distance is the correct shortest path distance and will not change any more.

Depending on the kind of priority queue used, Dijkstra's algorithm is asymptotically optimal. If the queue has a decreaseKey operation that runs in $\mathcal{O}(\log n)$, the runtime of Dijkstra's algorithm is bounded by $\mathcal{O}((|V| + |E|) \log |V|)$. While Fibonacci Heaps improve this to $\mathcal{O}(|E| + |V| \log |V|)$, they are quite slow in practice.

4.4.2. Bidirectional Search

The main weakness of Dijkstra's algorithm is that it has to visit every node with a shortest path distance shorter than or equal to t's. This can be mitigated by using bidirectional search.

Algorithm 6: Dijkstra's Algorithm

Data: graph $G = (V, E)$, weight function $w :\mapsto \mathbb{R}$, $s \in V$, $t \in V$
Result: shortest path from s to t, distance from s to t

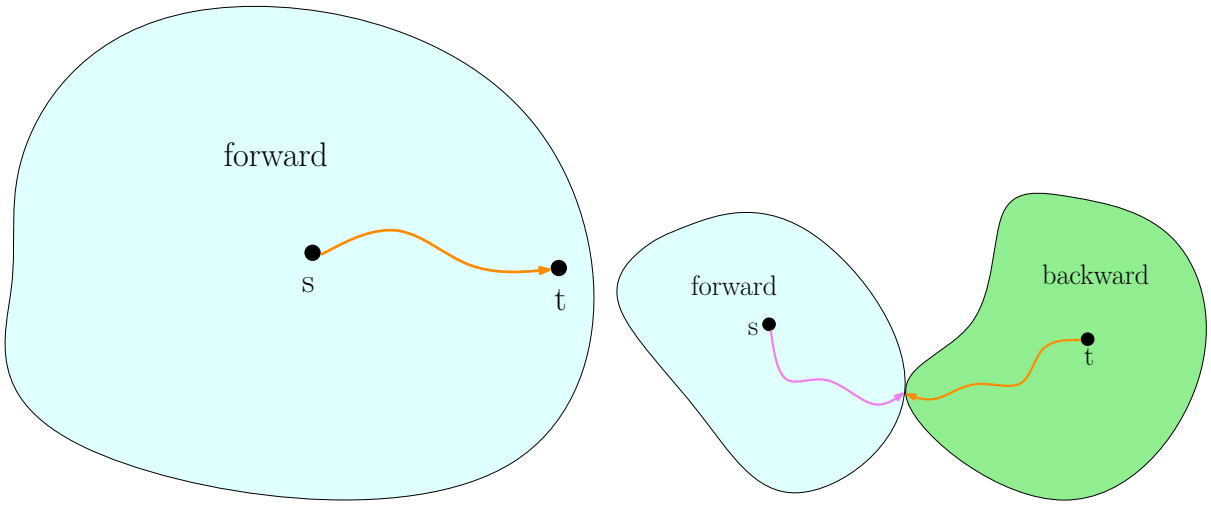
```
1  q ← addressable priority queue;
2  Insert(q,s,0);
3  distance(s) ← 0;
4  while NotEmpty(q) do
5  |  n ← DeleteMin(q);
6  |  if n = t then
7  |  |  return parent, distance
8  |  end
9  |  forall edges (n, u) ∈ E do
10 |  |  if u new node then
11 |  |  |  distance(u) ← distance(n) + w(n,u);
12 |  |  |  Insert(q,u,distance(u));
13 |  |  |  parent(u) ← n;
14 |  |  end
15 |  |  else if distance(u) > distance(n) + w(n,u) then
16 |  |  |  distance(u) ← distance(n) + w(n,u);
17 |  |  |  DecreaseKey(q,u,distance(u));
18 |  |  |  parent(u) ← n;
19 |  |  end
20 |  end
21 end
22 return parent, distance;
```

One Dijkstra search is performed from s and a backward search runs concurrently from t. After a node has been settled by both searches, the shortest path can be determined using both parent and distance functions (cf. Figure 4.8).

This technique usually visits only half of the nodes a normal Dijkstra's algorithm would. While this might not seem like a huge improvement, the bidirectional search enables or speeds up other shortest path algorithms and is therefore integral for many speed-up techniques.

4.4.3. Contraction Hierarchies

Given a directed graph $G = (V, E)$ with static edge weights $w : E \mapsto \mathbb{R}^+$, Contraction Hierarchies solve the shortest path problem. While they do not hold any asymptotic advantage over Dijkstra's algorithm, they are several magnitudes faster in practice, at least for road networks.



(a) Dijkstra's algorithm (b) Bidirectional search

Figure 4.8.: Dijkstra's algorithm and bidirectional search

Overview. The idea is to skip unimportant nodes during the query. Nodes are removed one by one from G and shortcut edges are inserted to preserve all shortest path distances between the remaining nodes. This is called *contracting* a node. All removed nodes and edges form a contraction hierarchy. A node u is called *higher up* in the hierarchy than a node v if it was removed from G after v was.

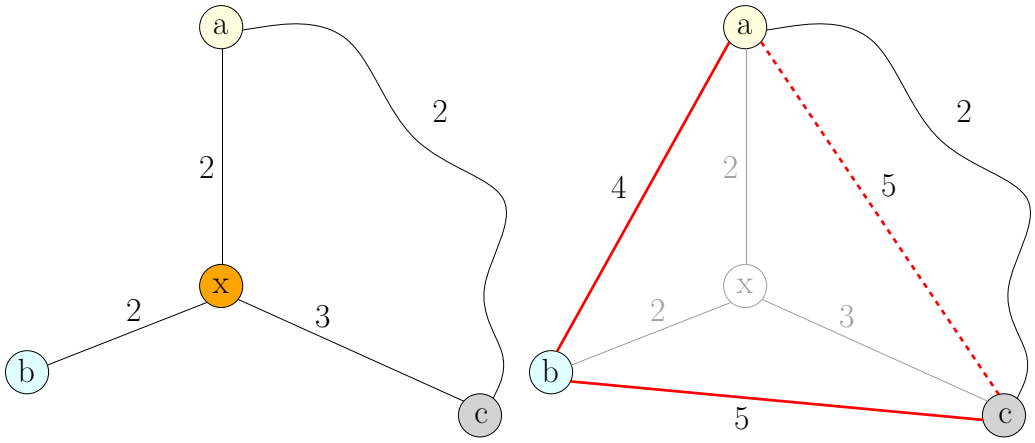
The query is similar to a bidirectional Dijkstra's algorithm. Given a source node s and target node t, a forward and a backward search start from s and t, respectively, relaxing only forward edges which lead to a node higher up. It is guaranteed that a shortest path[2] will be found due to the inserted shortcuts.

The order in which nodes are contracted is crucial to the speed of the query. A set of heuristics is employed to minimize the amount of inserted shortcuts and ensure a uniform distribution of contracted nodes. Nodes are evaluated based on a simulated contraction, nodes, edges, or already contracted nodes in the vicinity. Updating the heuristic and repeating the simulations after each contraction constitutes the main amount of work.

Construction. A simplified version of the construction procedure can be seen in Listing 7. The nodes are contracted one by one, according to the priority they were assigned by the heuristics. A detailed description of possible heuristics can be found in [GSSD08].

The contraction of a node x requires several steps. First of all, a set of shortcut edges is determined that maintains the correct shortest path distances: For each pair of edges (u, x)

[2]It is not guaranteed that all shortest path between s and t will be found



(a) Before (b) After

Figure 4.9.: Contraction of a node

and (x, v), a shortcut edge (u, v) with $w(u, v) = w(u, x) + w(x, v)$ is considered. If the path (u, x, v) is a shortest path the shortcut is inserted. This can easily be checked for by calculating the distances between each pair of neighbours u and v of x using Dijkstra's algorithm. The distance calculations are very local and therefore not too costly. The shortcut (u, v) bypasses x. In this case we call x the *middle node* of the shortcut. The middle node is stored for each shortcut to enable reconstruction of the original shortest path.

After the shortcuts are inserted, x and its adjacent edges are removed from the graph and added to the contraction hierarchy. These changes to the graph require the heuristic to be updated, which is done by repeating the simulation of the contraction for several nodes, depending on the heuristics used. Updating is generally several times slower than the contraction itself.

An example of a contraction can be seen in Figure 4.9: The node x is contracted and for each pair of neighbours a shortcut is considered. Because a and c are already connected by a shorter path, no shortcut is necessary. Afterwards, all shortest path distances remain the same.

Shortest Path Query. The query finds a shortest path from a source node s to a target node t. A modified version of a bidirectional search is used. Both forward and backward search only relax edges that lead higher up in the hierarchy. A search stops if the current node is further away than the tentative s-t distance. The result is a shortest path which may contain shortcuts. The shortcuts are replaced by the corresponding original edges recursively: Each shortcut (u, v) with middle node x is replaced by the two edges (u, x) and (x, v). Since these edges might be shortcuts themselves, this is repeated recursively until a shortest path in the original graph G is obtained.

Algorithm 7: Contraction Hierarchy Construction

Data: graph $G = (V, E)$
Result: contraction hierarchy for G

```
1  CH ← (V_CH ← V, E_CH ← ∅);
2  p ← InitPriorities(G);
3  while V ≠ ∅ do
4  |   n ← u|p(u) = min(p(V));
5  |   E ← E ∪ FindNecessaryShortcuts(G,n);
6  |   E_CH ← E_CH ∪ {(u, v) ∈ E|u = n ∨ v = n};
7  |   E ← E \ {(u, v) ∈ E|u = n ∨ v = n};
8  |   V ← V \ u;
9  |   p ← UpdatePriorities(G,n,p);
10 end
11 return CH;
```

Long shortcuts may require a great amount of unpacking steps. It is therefore prudent to store a pre-unpacked version of these shortcuts. A simple lookup will provide the unpacked version in this case. This requires duplication of data items stored with the original edges and path nodes, which may lead to higher space consumption.

A query example is given in Figure 4.10. Shortcuts are represented by dotted edges. Every shortest path is transformed in a way similar to the path between s and t. Forward and backward search both can reach the highest node on the path, relaxing only edges leading higher up. Whenever a search would have to go downwards to reach a node higher up, the relevant part of the path is represented by a shortcut.

The query only inspects a small part of the graph, skipping all edges leading to lower nodes. The drawback is that these edges cannot be used any more to identify paths wrongly assumed to be shortest paths: Forward or backward search may mark nodes as settled even though their tentative distance is not yet the shortest path distance. Searching for a shortest path via such nodes is unnecessary, as we do not discover new shortest paths this way. A technique called *stall-on-demand* is used to identify some of those nodes and prune the search. The procedure is only described for the forward search, but the backward search proceeds analogously. When the forward search encounters a backward edge (u, v) while settling a node v and the node u has already been visited, it checks whether v would have been reached on a shorter path via u. If this is the case v has been settled on a non-shortest path and can be stalled. A stalled node is simply ignored when it is deleted from the priority queue. This stalled state is now propagated through the already discovered nodes with a breadth-first search. We have to take care when stalling reached nodes, as they may yet be reached via a shorter path. When this happens they have to be un-stalled.

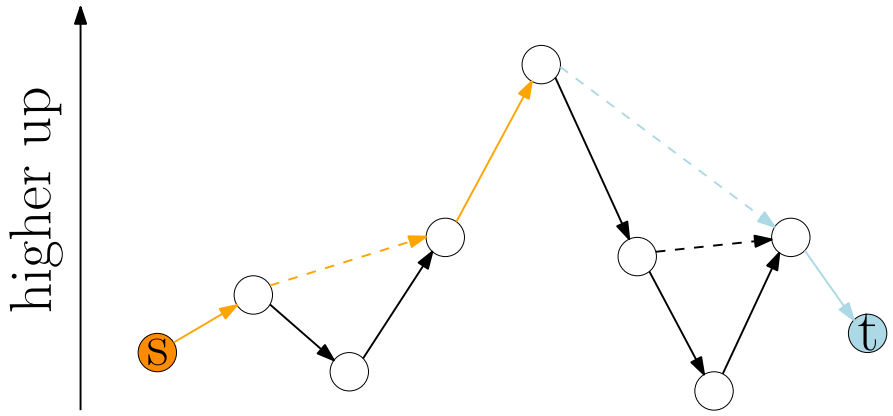


Figure 4.10.: Contraction Hierarchies Query

Data Structures. Since the query only relaxes edges leading to nodes higher up, it is unnecessary to store incoming or outgoing edges leading to lower nodes. This cuts the amount of edges stored in half. Even though we are using a bidirectional query each edge is only stored once.

Mobile Data Structures. Although the query is limited to only a few nodes, these are increasingly scattered throughout the graph as the search ascends the hierarchy. Cache misses are expensive on mobile devices and therefore it is desirable to improve the data locality. The nodes are first ordered topologically by a depth-first search. This groups nodes together that are likely to be visited in tandem during the query. Then nodes of similar height are sorted into bins, conserving their order within each bin. This is very beneficial, especially for the highest nodes, which are almost always visited in every mid and long range query. It is also notable that sorting the nodes into bins does not destroy the topological property of our order, as edges only lead to nodes higher up.

To exploit the nature of the flash memory used in most mobile devices the graph is then compressed into fixed-size blocks. The blocks are filled greedily one after another according to the new node order. Each block is compressed in a way that allows for random access. The topological property of the order is exploited during the compression. Each edge only leads to already compressed nodes, and therefore we know in which block they have been placed. We make the distinction between internal and external edges, i.e., edges leading to other blocks. We add another indirection for external edges by storing the ID of all adjacent blocks in an array. External edges now refer to the adjacent blocks array instead of the block ID of their target. Furthermore, we distinguish between internal and external shortcuts, i.e., shortcuts whose middle node is in another block.

All the data in a block is compressed by using as few bits as possible. The first-edge array, adjacent blocks array, node information, external edges target and flags are straightforward.

Internal edges of a node can only lead to nodes added to the block beforehand and therefore require even less bits. The edge weight is partitioned into two classes that use a small and a larger number of bits, respectively, since shortcuts tend to have larger edge weights. We compute an optimal partition for this. Storing the middle node of an external shortcut requires the maximal amount of bits, though, as it is not yet known in which block the node will reside.

Unpacking of shortcuts slows down the query by a great amount. Whenever an external shortcut is unpacked, it may require the loading of the block the middle node resides in. We therefore pre-unpack all external shortcuts. We decrease the amount of additional data necessary by exploiting the following observation: Whenever a shortcut e_1 is removed during a contraction and a new shortcut e_2 is inserted that uses it, the unpacked path of e_1 is now included in the unpacked path of e_2. Therefore, we only need to store unpacked versions for shortcuts which are not part of other shortcuts, and point the remaining shortcuts to the right sub-sequences. This saves about half the storage space for pre-unpacked shortcuts.

During the query a LRU cache is used. Whenever the edges of a node are requested, the block containing the node is loaded into the cache.

4.4.4. Preprocessing

First of all, the OpenStreetMap graph has to be converted to a simple graph: parallel edges and loops are removed. Then a Contraction Hierarchies variant is used to preprocess the graph. [Vet09] showes that Time-dependent Contraction Hierarchies can be parallelised and simultaneously simplified. We adopt this approach for Contraction Hierarchies (cf. Listing 8): Heuristics determine a set of nodes to be contracted in parallel and are updated after a set of nodes is contracted.

The set of heuristics we adopted is simpler than the original set. For each node x it is determined whether it should be contracted in the current iteration by:

1. A contraction of x is simulated.
2. A priority $p(x)$ is computed using a weighted sum of:
    - *Edge Quotient*: The amount of shortcut edges added during simulation divided by the amount of edges removed. This penalises adding a lot of shortcuts, keeping the graph sparse.
    - *Hierarchy Depths*: Initially, each node u has a depth $d(u) = 1$. After a neighbour v of u is contracted, the depth is updated to $d(u) = \max(d(u), d(v) + 1)$. This emphasises uniform distribution of contracted nodes and keeps the height of the hierarchy small.

Algorithm 8: Parallel Contraction Hierarchy Construction

Data: graph $G = (V, E)$
Result: contraction hierarchy for G

```
1  CH ← (V_CH ← V, E_CH ← ∅);
2  p ← ParallelInitPriorities(G);
3  while V ≠ ∅ do
4  |   N ← ParallelGetContractionNodes(G,p);
5  |   E ← E ∪ ParallelFindNecessaryShortcuts(G,N);
6  |   E_CH ← E_CH ∪ {(u,v) ∈ E|u ∈ N ∨ v ∈ N};
7  |   E ← E \ {(u,v) ∈ E|u ∈ N ∨ v ∈ N};
8  |   V ← V \ N;
9  |   p ← ParallelUpdatePriorities(G,N,p);
10 end
11 return CH;
```

- *Original Edges Quotient*: Initially each edge (u, x) represents one original edge and $o(u, x) = 1$. For a shortcut (u, v) with middle node x we set $o(u, v) = (u, x) + (x, v)$. The original edges quotient is $\sum_{\substack{(u,v)\\ \text{added}}} o(u, v)$ divided by $\sum_{\substack{(u,v)\\ \text{removed}}} o(u, v)$.

3. If for each neighbouring node u in the k-neighbourhood of x $p(x) < p(v)$, x is added to the set of nodes to be contracted. The k-neighbourhood of x are all nodes that can be reached from x by a depth-first search which is pruned at depth k. For $k \geq 2$ the nodes can be contracted in parallel without any data dependencies.

A value of 2 is used for k as it gives the best performance for preprocessing and query time. Step 1 and 2 have to be repeated for a node x only after a neighbour has been contracted. Step 3 has to be repeated in each iteration. This heuristic can be updated in parallel.

These changes do not only speed up the preprocessing, they also greatly simplify the heuristics compared to the original Contraction Hierarchies, which e.g. compute Voronoi decompositions of the graph. We will see in Section 5.7 that we do not sacrifice query time, and even save a great deal of memory during preprocessing.

While we also incorporated the dynamic version of Contraction Hierarchies from [GSSV09], no information about traffic jams and road works is obtained at the moment. For further details we refer to this paper, since the technique described therein can be easily implemented.

4.4.5. Data Structures

The same data structure introduced by Mobile Contraction Hierarchies can be used. We only need to integrate the data we want to store with each edge and node. For the moment we

store the coordinates of each node in the same way the ID was stored, dropping the ID. Also, no data is stored for the edges. The coordinates need more storage space than the ID. In particular, the size of the pre-unpacked shortcuts' path increases, a problem which intensifies when data is stored for the edges.

4.4.6. Query

The query computes the distance between two points s and t located on two edges. s is located on the edge (u, v) with distance d_{u-s} from u and t is located on the edge (x, y) with distance d_{x-t} from x. First of all, it is checked whether both edges are identical, in which case the shortest path can easily be returned. Otherwise, it is guaranteed that one of the nodes $\{u, v\}$ and one of the nodes $\{x, y\}$ lies on the shortest path. Then, we run a modified Contraction Hierarchies query: The forward search starts from u and v with the appropriate distances, the backward search from x and y. This results in a path from u or v to x or y. We check which path is the shortest. Now the coordinates s and t are added to the start and end, respectively, thereby obtaining a shortest path from s to t.

4.5. Address Lookup

The address lookup module has to suggest address entries for a partially inputted string. It makes use of trie and tournament tree data structures to compute suggestions.

4.5.1. Trie

A trie, or prefix tree, is used to store and retrieve data associated with strings [DLB59] [Knu98]. It is a tree where each edge is labelled with a character. The key associated with each node is the sequence of characters on the unique path from the source of the tree to said node. A trie with several words inserted can be seen in Figure 4.11. The orange filled nodes have data associated with them. The respective keys are listed on the side.

Construction. The construction of a trie is straightforward. Given a set of pairs $(key, data)$, we begin with an empty trie, containing only the source node. For each pair $(key, data)$ we descend the trie from the source to the node associated with the key. At each node we search for the edge labelled with the appropriate character. If such an edge does not exist we insert it along with a new node. Therefore, each new key may require up to length(key) new nodes and edges. The data is then stored with the associated node.

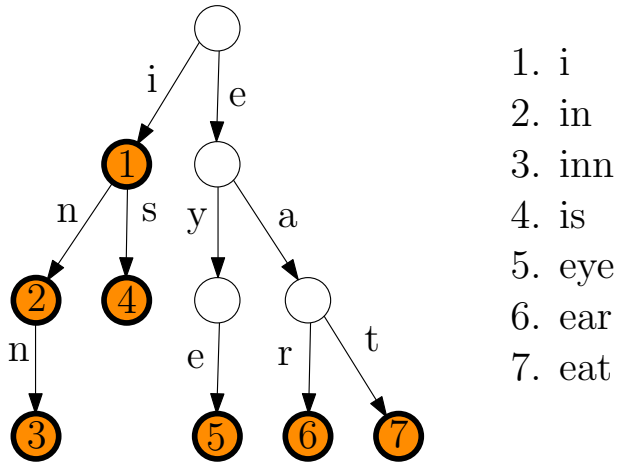


Figure 4.11.: Trie

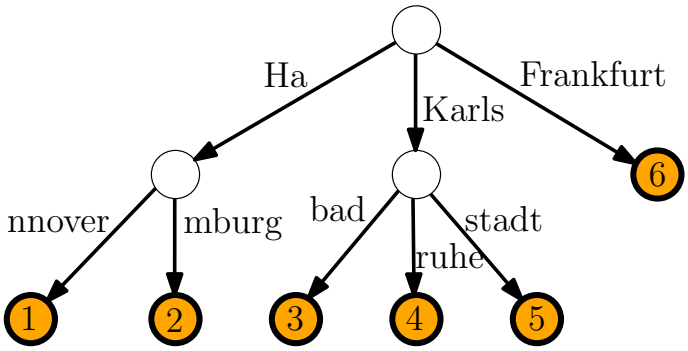
Key Lookup. Given a key, we want to find out whether or not it was inserted into the trie and return the associated data. We proceed as if we were to insert the key. When an edge would have to be added, we can be sure that the key was never inserted. Otherwise we find the associated node and can return the data.

Key Suggestions. Given a key prefix, we can return a set of keys starting with said prefix by simply finding the node associated with the prefix. The path to each stored item of data in the sub-tree of this node represents a key with the required prefix.

Path Compression. Usually a trie contains many nodes with only one child. Such nodes can be removed from the trie. Edges are labelled with strings instead of characters. We traverse the trie in a breadth-first manner. For each node n we find paths $(n, n_1, ..., n_k)$ with $n_i, 1 \leq i < k$, having only one child, n_k having less or more than one child, and each $n_i, 1 < i < k$, having no data stored with it. We replace each path with a single edge labelled with the string associated with the path.

The query can easily be adapted to cope with compressed tries. When looking for the node associated with a key we now search for a matching substring instead of a character. Because there can be at most one such substring, the required changes are minimal.

A trie with path compression has at most two nodes per key inserted and therefore very little overhead. This can be seen in Figure 4.12. To store this trie without path compression we would need an additional 31 nodes.



1. Hannover
2. Hamburg
3. Karlsbad
4. Karlsruhe
5. Karlsstadt
6. Frankfurt

Figure 4.12.: Trie with path compression

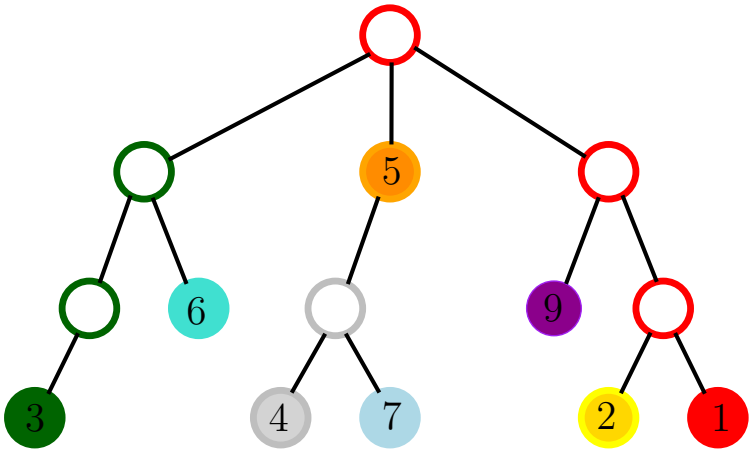


Figure 4.13.: Unbalanced Tournament Tree

4.5.2. Tournament Tree

A tournament tree is a data structure used to extract the k smallest values from a set of n values [Knu98]. It is a tree where nodes can be associated with distinct values. Each node is labelled with the node associated with the smallest value in its sub-tree. Figure 4.13 shows an unbalanced tournament tree. The nodes containing numbers have been assigned values. Each node is labelled with the colour of the node with the smallest value in its sub-tree as the border colour. The smallest valued node can be read directly from the source, in this case the red node. Also note, that each sub-tree is a tournament tree itself. We will make use of this fact later on.

Construction. Given a tree with n nodes we simply compute the correct labels for all nodes with a depth-first search. If only a set of values is given we can construct a balanced tree from it and label the nodes correctly. In both cases the complexity is $\mathcal{O}(n)$.

Select Query. The set s with the k smallest values has to be computed. We start with the source node. It is labelled with the node containing the smallest value, which is subsequently inserted into S. Whenever a value is inserted into S, we remove the associated node n and perform an update to maintain a valid tournament tree. This is done by traversing the path from the node n up to the root. Each node u on the path is labelled with n. We now search among all labels of all children of u for the smallest label and label u with this label.

In this manner we can find the smallest k values. In a balanced tree with n nodes the asymptotic complexity is in $\mathcal{O}(k \log n)$.

If the tournament tree data structure is read-only we cannot update the tree. But, if k is small enough for k nodes to fit into writeable memory we can still perform a select query: For each node that we alter the modified version is stored in main memory. These nodes are sorted by the value of their currently labelled nodes. Since we only want to extract k values, it suffices to store the k smallest modified nodes in writeable memory. All other nodes could not possibly have one of the smallest k values left in their sub-tree. Because we do not remember all modifications, nodes labelled with already extracted nodes have to be ignored.

4.5.3. Preprocessing

For each place[3] a trie is built by inserting all associated street names as keys and their coordinates as values. Streets that belong to no place are ignored. The streets are now ordered by the aggregate length of their pieces. Each trie is then used to build a tournament tree. We combine both in a tree that is a trie as well as a tournament tree: Edges are labelled with strings and characters, nodes are labelled with the rank of the most important street name stored in their sub-tree.

Places are ordered by their population count. Then each place name, along with the index of its respective trie, is inserted into the place name trie. A tournament tree is built from and combined with it in the same way as for the street names.

4.5.4. Data structures

Tries with path compression are used. The string labels of the edges are stored in UTF-8 [TA06]. UTF-8 is a variable length character encoding for unicode strings. It is backwards

[3]City, town, hamlet or village

compatible with the ASCII encoding scheme, which saves storage space, especially for the 95 included printable characters.

All tries are stored sequentially in one file. Each edge stores the target node's position in the file directly, instead of its ID. This makes storing an index for the node data unnecessary: The only node we need to address randomly is the source node of a trie. Its position is simply fixed at the start of the trie. Each node is then stored in a contiguous area together with their outgoing edges.

4.5.5. Query

The query has to suggest address entries for a partial input string. First, the trie data structure is used to look up its associated node x. The tournament tree data structure of the sub-tree of x can then be used to extract the best k address entries starting with the input string. This is done by keeping the most promising k nodes encountered so far in a priority queue. Iteratively the node labelled with the smallest rank is extracted from the priority queue and all child nodes are inserted. Whenever a node with an associated name is extracted, we add it to our tentative suggestion list. Once the node labelled with the smallest rank in the priority queue has a larger rank than any of our k smallest tentative suggestions, we have successfully extracted the k best ranked suggestions in the sub-tree.

An example is depicted in Figure 4.14. A ranked list of place names is used to build a trie. Subsequently, a tournament tree is build upon and eventually combined with it. Next, a query for places starting with "Karl" is processed: The prefix is looked up in the trie and suggestions are extracted.

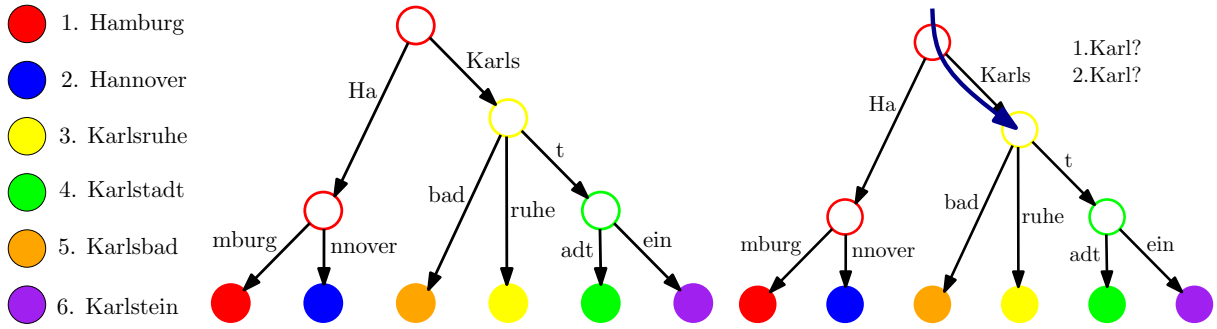
The query is not very time critical, as the average street name trie is quite small, e.g., about 0.7KB for Germany. Furthermore, for each character the user inputs or deletes only one query is issued. Therefore, no cache is used and each node is read directly from the file.

4.6. GPS Lookup

The GPS lookup module finds nearby edges using a grid structure.

4.6.1. Preprocessing

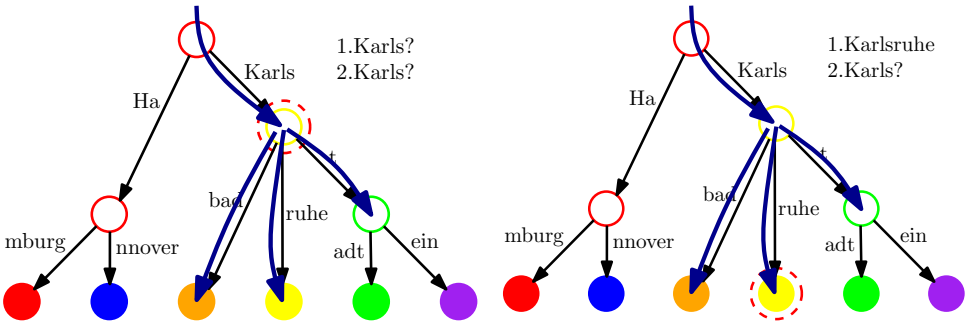
All edges are inserted into a grid: Their coordinates are transformed into the Mercator projection. Then the area is divided into equally sized cells. Each edge is inserted into every cell it intersects, which can easily be achieved by employing Bresenham's line drawing algorithm [Bre98].



(a) Ranked list of place names

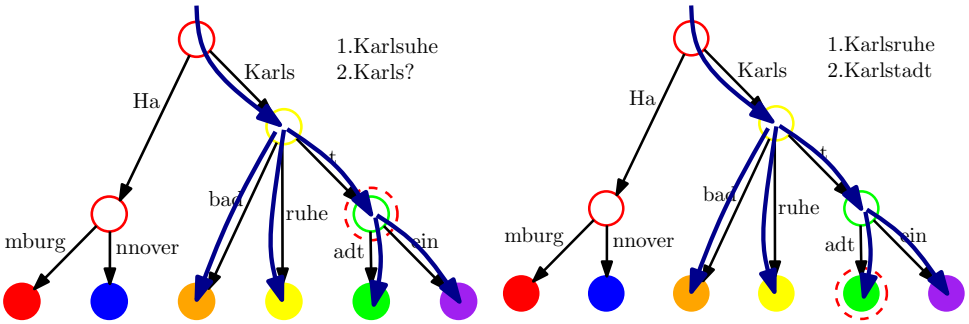
(b) Combined trie and tournament tree

(c) Lookup of "Karl"



(d) Extract suggestion, Iteration 1

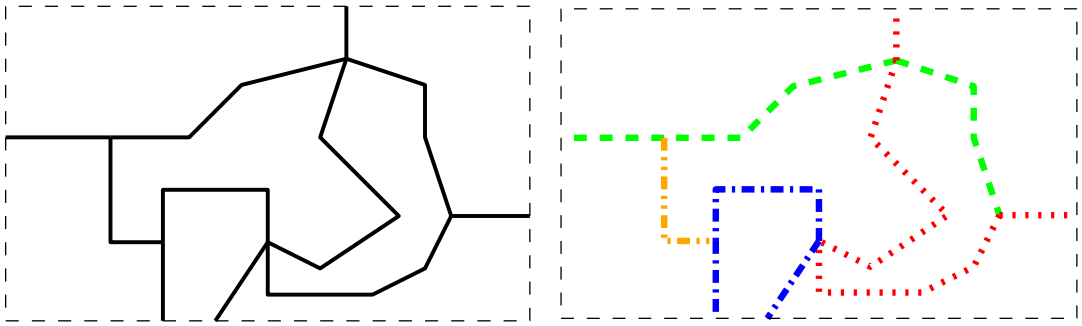
(e) Extract suggestion, Iteration 2



(f) Extract suggestion, Iteration 3

(g) Extract suggestion, Iteration 4

Figure 4.14.: Address lookup query



(a) Routeable edges within a cell (b) Decomposition into paths

Figure 4.15.: A cell is divided into paths

4.6.2. Data Structures

Each cell is compressed separately. First of all, the edges are greedily chained into paths: We randomly choose a starting edge and then iteratively add suitable edges until no further successor can be found. This process is repeated until all edges belong to a path (cf. Figure 4.15). Then, the node data, i.e., coordinates and IDs, is sorted by the order it is referenced in the paths. This enables us to index the nodes in a path efficiently: For each path node we store a flag indicating whether the node has been referenced before. If the flag is set to 0 we do not need to store the index of the node explicitly, as it is the next unreferenced node in the node data. Otherwise, we store the node data array index of the referenced node. This requires a relatively small amount of bits, as only previously referenced nodes are addressed. Because many nodes within OpenStreetMap data have only two adjacent edges, a large amount of storage space is saved: Only intersections have to be explicitly referenced.

Node coordinates are clipped to the cell using the Liang-Barsky line clipping algorithm [Bar84]. As cells are all fixed-size, the range of coordinates is well known and sufficiently few bits are used to store them.

This compression scheme does not enable us to randomly access an edge within a cell. Instead the whole cell has to be unpacked. This is no disadvantage, though, since the query computes the distance to all edges in relevant cells anyway.

4.6.3. Query

The query returns a set of edges within a given radius of a position. For each edge the nearest location on the edge is marked. First of all, the grid cells that fall within the given radius are determined. For each cell, the distance to all edges is computed and, at the same time,

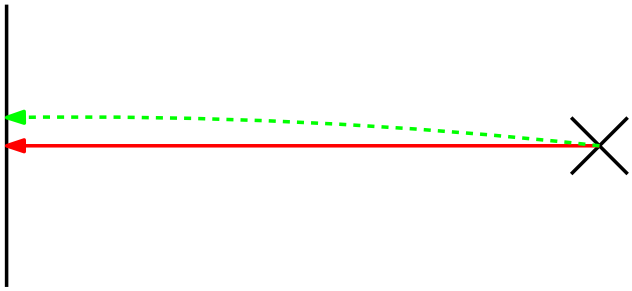


Figure 4.16.: Approximating the nearest point with Euclidean geometry

the nearest point on the edge. This is done with euclidean geometry. Shortest paths on the surface of a sphere are orthodromes, i.e., great circle arcs. They usually are not projected onto straight lines in the Mercator projection, but curves. Within a local environment a straight line is a good approximation, though. In Figure 4.16 an example is given. The green dotted path is the shortest path from the given position to the edge. The approximation used assumes the red path is shorter and computes a slightly different nearest point. This is a well known fact in navigation, where these approximate shortest paths are known as rhumb lines or loxodromes.

5. Experiments

5.1. Test Setup

The preprocessing was tested on a machine with a Core2 Duo E8400 at 3.0GHz. 4 memory modules with 2GB RAM each were used in dual channel mode. Both cores share a 6MB level 2 cache. The program was compiled with GCC 4.4.1, using optimisation level 3. The machine was running Xubuntu 9.10.

The mobile application ran on a HTC Diamond Touch 2. It features a Qualcomm MSM7200A processor at 528MHz, and is equipped with 288MB RAM, a 480x800 TFT-LCD touchscreen and an internal GPS antenna. A SanDisk 16GB microSDHC card with a speed performance rating of class 2 was used to store the data. The device ran Window Mobile 6.1 with a HTC modified user interface. Both the mobile program and the libraries were compiled with Visual Studio 2008, using whole program optimisation and the highest optimisation level.

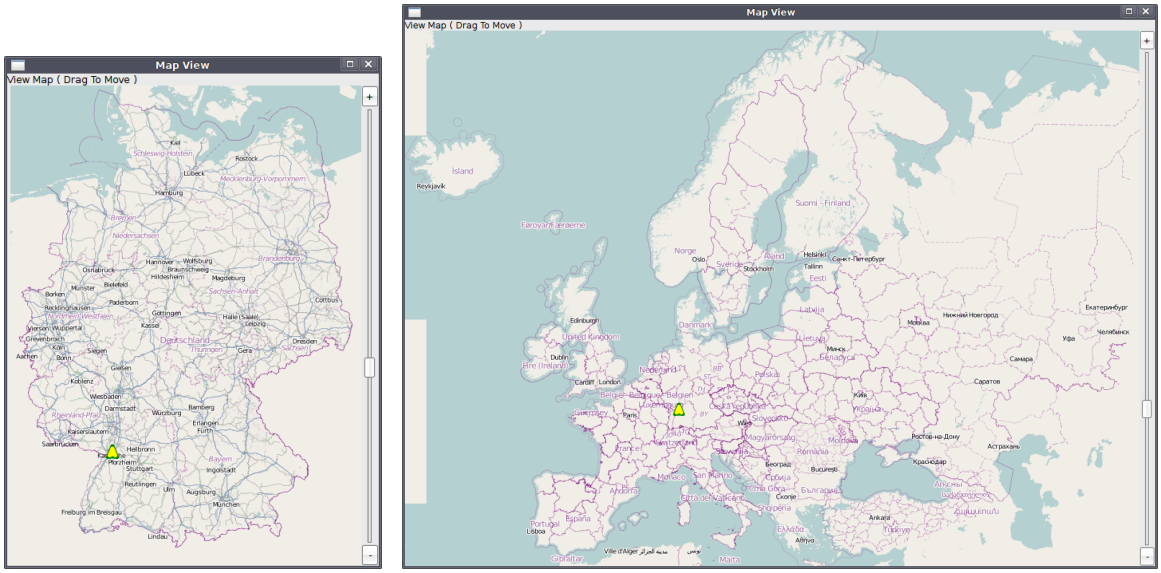
5.2. Input

Our main input set is the OpenStreetMap subset of Germany. Most experiments are conducted on the European subset as well, but the raster renderer requires too much storage space for its preprocessed data to fit on the SD card. Therefore, the rendering module is excluded from tests on the European data set.

The German subset consists of 10 357 560 routing nodes and 22 169 092 routeable edges on 1 828 176 ways. It covers an area of about 357 000km^2 and is stored in an 6.8GB XML file.

The European subset consists of 48 181 278 routing nodes and 102 034 476 routeable edges on 7 065 314 ways. It covers an area of about 20 000 000km^2 and is stored in a 34.2GB XML file.

Figure 5.1 shows map views for both input sets.



(a) Germany (b) Europe

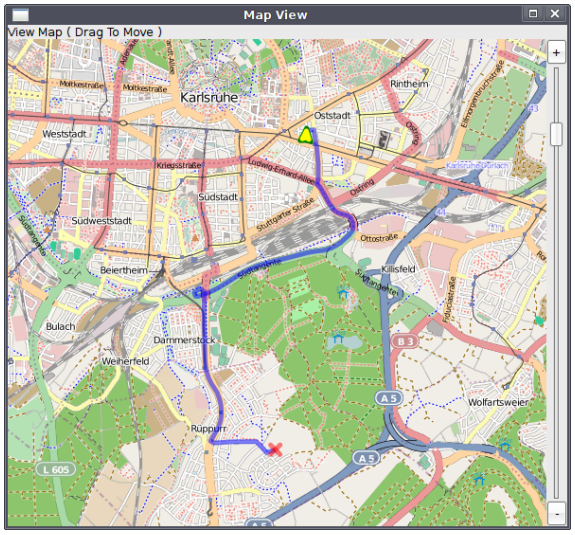
Figure 5.1.: Input subsets

Name	Target	Distance [km]	Time [m]	Average Speed [km/h]
Short	Karlsruhe-Rüppurr	7	9	46
Mid	Wiesbaden-Dotzheim	152	98	101
Long	Hohen Neuendorf	654	375	109
Very Long	Minsk	1 678	1 314	81

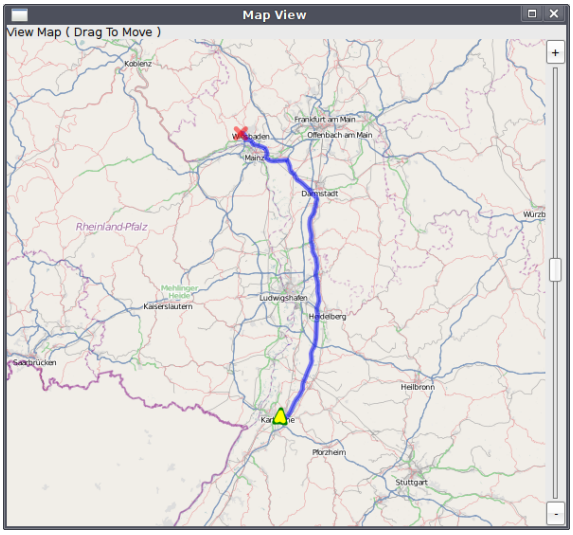
Table 5.1.: Test drives

5.3. Test Drives

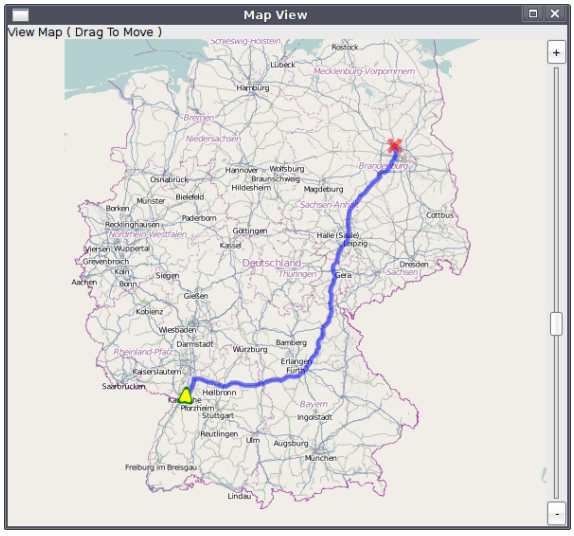
The query speed of the GPS lookup, rendering and routing module is measured for four different test drives. All drives start out in Karlsruhe. Their target, travel distance, travel time and average speed are listed in Table 5.1. Figure 5.2 shows a map view for each of them. These tests are not recordings of actual drives, but rather synthetic benchmarks, which are constructed from the shortest path between Karlsruhe and the destination. A new GPS position is generated twice per second. Each position update triggers a query for the rendering, routing and GPS lookup module. To better simulate the behaviour of real GPS hardware a random error of up to 10m is introduced. For each of the three involved modules we present their query time averaged over the whole drive, as well as a more detailed scatter plot containing all query times.



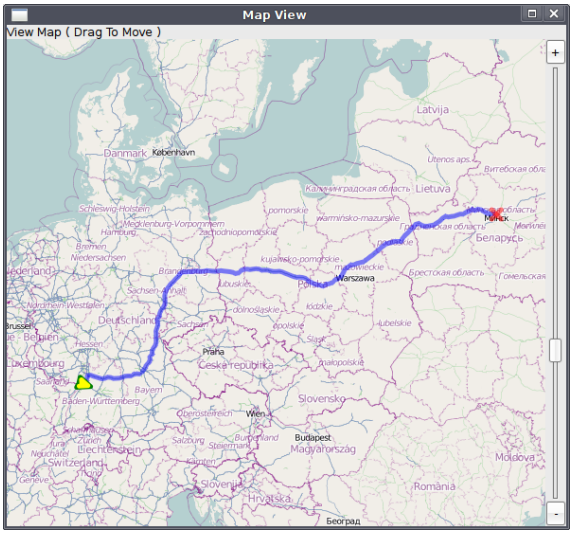
(a) Karlsruhe-Rüppurr



(b) Wiesbaden-Dotzheim



(c) Hohen Neuendorf



(d) Minsk

Figure 5.2.: Test drives

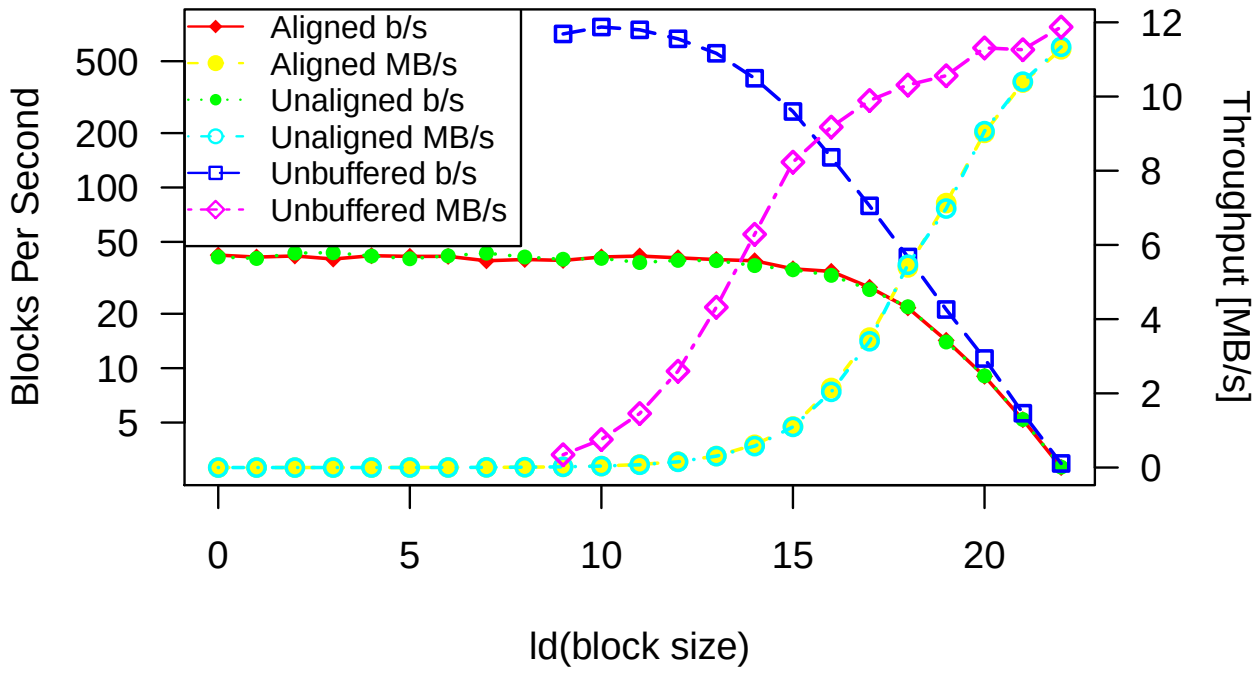


Figure 5.3.: Reading from flash memory

5.4. Flash Memory

We test the flash memory's reading performance for varying block sizes. 1000 random blocks out of a 1GB file are requested for each block size. Figure 5.3 depicts the results for aligned and unaligned access. Up to a block size of 16KB, the amount of blocks read per second remains essentially the same. Increasing the block size beyond this point results in a larger throughput, but diminishes the amount of blocks read per second. Most modules have little use for the high sequential reading speed achievable by choosing larger block sizes, and therefore read the data in small chunks. Consequently, we are limited to about 40 block reads per second across all modules. This is rather slow when compared to the performance described in [SSV08], where about 1000 to 500 block reads with 1KB to 4KB sized blocks are reported. On the other hand, switching to unaligned access does not incur the expected speed penalties.

The comparatively poor reading speeds are most likely due to overzealous prefetching strategies of the underlying OS. We bypass wxWidgets and access the file directly, which results in enormous improvements (cf. Figure 5.3). Unbuffered access has to be aligned to the file system block size, though. The results are more in line with [SSV08]. Since most modules require unaligned access this approach is unsuitable for our purpose.

	Preprocessing [s]	Memory [MB]	Temporary Storage [MB]
Germany	533	960	810
Europe	2 533	5 308	3 713

Table 5.2.: Importer preprocessing

	Preprocessing [s]	Memory [MB]	Storage [MB]
Germany	7 265	938	4 223
Europe	165 525	1 081	35 483

Table 5.3.: Rendering preprocessing

5.5. Importer

The main workload of the importer is parsing the OpenStreetMap XML file. This is reflected in the preprocessing time in Table 5.2: Importing the European data set takes proportionally longer. The importer uses temporary files to store intermediate data in between its two passes and before relaying it to the respective plugins. These files are compressed with wxWidgets zlib streams, resulting in only 810MB temporary storage used for the 6.8GB Germany XML file. In spite of this, memory usage is still high, which is due to additional requirements for associating each node with a city: The *k*d-tree, node information, coordinates and boundaries have to be kept in memory.

5.6. Rendering

A tile size of 256x256 pixels is used. The PNG data is reduced to 256 colours and tiles without adjacent edges are removed. Each meta-tile is 31x31 tiles large, with an additional margin of 1 tile. Zoom levels 0 through 15 are rendered, which amounts to a total of 1 284 564 tiles, or 78.4 Giga Pixel, for Germany, and 132 287 640 tiles, or 8.1 Tera Pixel, for Europe.

As a detailed raster renderer, it is expected for the rendering module to take up a considerable amount of preprocessing time and storage space. This is confirmed in Table 5.3. It takes about 2 hours to raster Germany and about 2 days to raster Europe. While Europe requires about 100 times more tiles, it is only about 20 times slower in preprocessing, since large areas are devoid of OpenStreetMap data, such as the Mediterranean Sea. The PNG compression works quite well for this kind of data. The 78.4 Giga Pixel for Germany would require about 940GB storage space uncompressed, instead of 4.2GB in PNG format. The compression ratio is even better for Europe, only 35.5GB instead of 97TB are needed.

Table 5.4 lists query times for the rendering module. A cache size of 20 tiles was used. The European data set is omitted, as it took up more storage space than was available. Rendering

	Rotate	Short	Mid	Long	Very Long
Germany	no	15.64	37.11	105.34	-
Germany	yes	66.28	87.99	166.67	-

Table 5.4.: Rendering query times in milliseconds

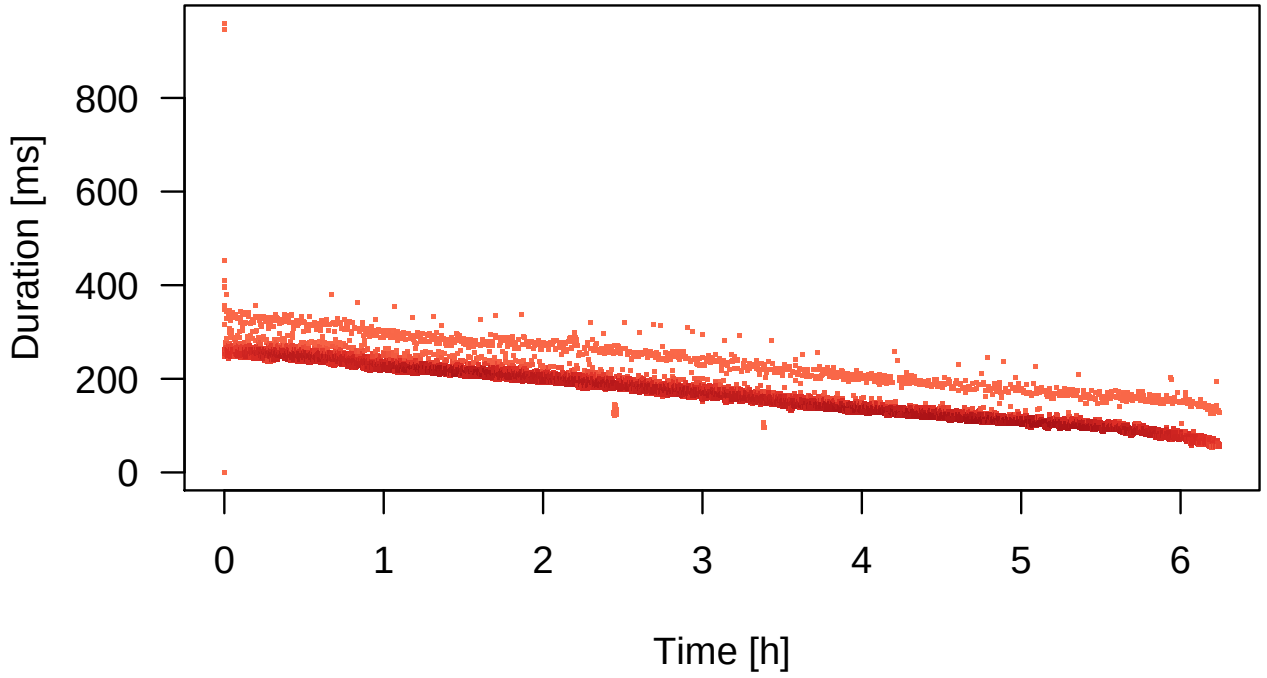


Figure 5.4.: Rendering time for long range drive

the tiles is actually quite fast, even if rotated. Clipping and rendering the anti-aliased and transparent route overlay takes more time, especially for longer routes. If the image has to be rotated, additional time is spent to draw the tiles. On the other hand, rotating the route itself does not have a large impact on query speed. The overhead for rotating the image is only 61ms for the very long test drive, while even the short test drive requires an additional 51ms.

Detailed measurements for the long range test with rotation can be seen in Figure 5.4. The rendering speeds up as the distance to the target decreases, since the routes which have to be rendered become shorter. Moreover, it is apparent whenever the query has to load tiles from file and decompress them, instead of reading them from the cache.

It was a design choice that the rendering module should render device-independent images. This introduces additional overhead (cf. Table 5.5). The conversion to a native bitmap is quite fast. Drawing such a bitmap, however, takes longer than even most rendering queries. This is due to a limited bitmap implementation in the Windows Mobile port of wxWidgets.

	Convert	Draw
Time [ms]	7.73	112.48

Table 5.5.: Image conversion and drawing times

	Preprocessing [s]	Memory [MB]	Storage [MB]
Germany	204	1 199	296
Europe	1 279	5 289	1 504

Table 5.6.: Routing preprocessing time and space consumption

Device-dependent bitmaps are not supported at the time of writing. Instead, the image is first converted to a device-independent Windows Bitmap type. The OS' drawing routine then has to convert the bitmap to the device-dependent Windows Bitmap format on-the-fly.

5.7. Routing

During the preprocessing of our Contraction Hiearchies variant, the priority of a node is determined by a heuristic using a weighted sum. We use the following factors: edge quotient 2, hierarchy depth 2, original edges quotient 1. For the compressed graph data structure a block size of 4KB is chosen, as this is the standard block size in most file systems.

Table 5.6 shows the preprocessing results. Processing the German data set only takes 204 seconds and needs about 1.2GB RAM. Also, the preprocessing scales quite well: The European data set is about 5 times larger and only takes about 6 times longer, while requiring less than 5 times the memory. The required storage space scales just as well. Germany needs 296MB and Europe about 1.5GB.

Table 5.7 shows the storage requirements in detail. It is obvious that the user data, i.e., node coordinates and pre-unpacked shortcut paths, makes up the bulk. The pre-unpacked shortcuts alone account for more than half of the storage space. This will worsen if additional edge or node data has to be returned by the routing module. Pre-unpacking the shortcuts cannot be omitted without a large impact on query speed, however, as shown by [SSV08]. Overall, the high compression ratio redeems the large overhead and keeps file sizes relatively small.

Table 5.8 shows the average query times for all 4 test drives. A cache size of 4 MB is used. The very long test drive is omitted for Germany, since it leaves the subset early on. On the German subset the query takes only 2.48ms on average for the short range drive and about 21ms for the long range drive. Query times scale quite well for the European subset, taking only slightly longer for both the short and long range drive, while the mid range drive actually

	Germany		Europe	
	[MB]	Comp. [MB]	[MB]	Comp. [MB]
Edge Index	39	14	183	68
Edge Target	74	26	364	129
Edge Weight	76	31	365	146
Middle Node	34	14	169	70
Coordinate	158	50	734	238
Pre-Unpacked	311	151	1 581	818

Table 5.7.: Routing storage space and compression

	Short	Mid	Long	Very Long	Random
Germany	2.48	9.68	20.59	-	76.98
Europe	3.03	9.08	21.76	32.05	272.99

Table 5.8.: Routing query times in milliseconds

is 0.6ms faster. The very long range drive needs 32.05ms, though. The increased graph size seems to have little impact on the query time. Of course longer queries also take more time.

To allow for a better comparison with routing papers we also perform tests with random queries. First, 1000 random queries are executed to warm up the cache. Then, query times for an additional 1000 random queries are measured. The results are displayed in Table 5.8 along with the test drive measurements. Since random queries tend to span half of the covered area on average their execution times are longer: 76ms and 273ms for Germany and Europe respectively. These values reflect the time required to compute a route for the first time. The only relevant blocks in the cache represent the top part of the hierarchy, as it is needed in almost every long distance query. During a normal test drive, subsequent route calculations benefit from the fact that the queries are very similar, e.g., most necessary blocks have already been loaded into the cache.

Figure 5.5 shows query times for the long range drive and the German subset in detail. The influence of cache misses is obvious. The number of cache misses decreases towards the end of the drive, since the overall search space grows smaller. It can be seen that the routing time increases during the first hour. This is caused by the fact that the source, Karlsruhe, is situated near the border of the German subset, which initially limits the search space. Also, the routing time erratically drops or rises for short time intervals. Whenever the current location is higher in the hierarchy, the forward search space is smaller, leading to decreased query times. The backward search space remains almost identical, only decreasing as the drive nears its end.

Table 5.9 compares the Parallel Contraction Hierarchies (PCH) variant introduced in this thesis with the Contraction Hierarchies (CH) implementation from [GSSD08]. Query times

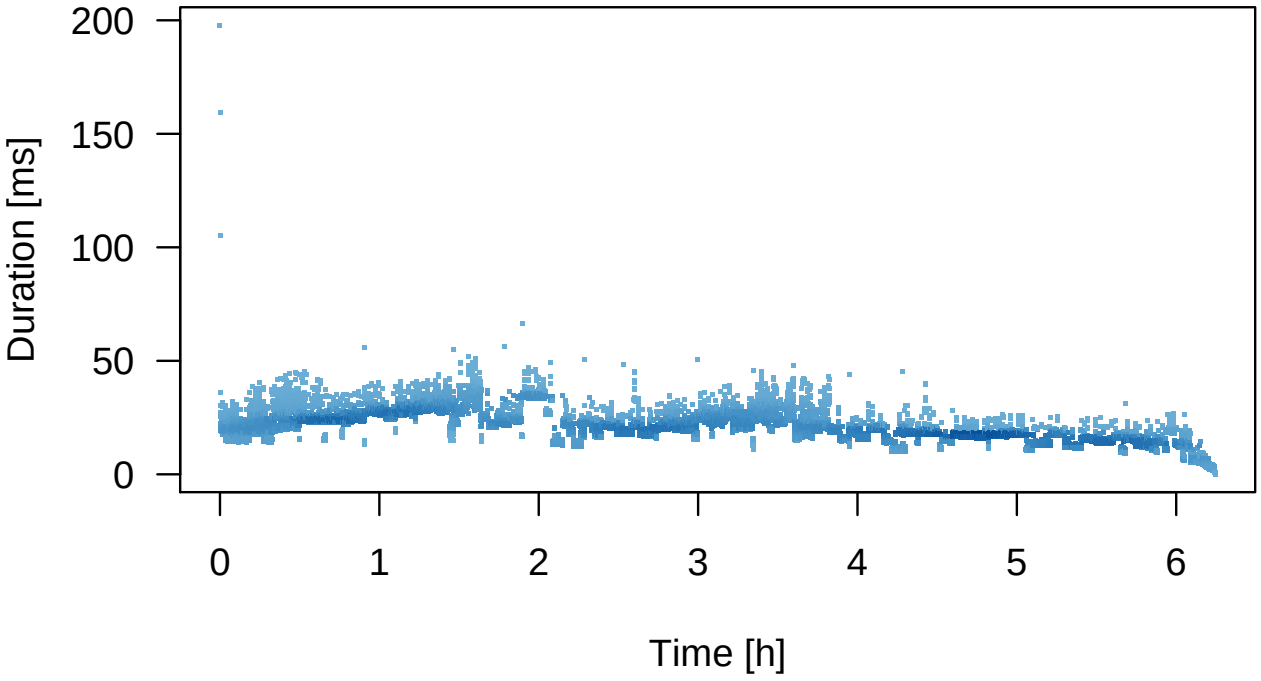


Figure 5.5.: Routing time for long range drive

	Preprocessing [s]	Memory [MB]	Storage [MB]	Query [ms]
Germany (CH)	275	2 181	389	0.3086
Germany (PCH)	204	1 199	382	0.3089
Europe (CH)	-	>RAM	1 841	0.6582
Europe (PCH)	1 006	5 289	1 815	0.6610

Table 5.9.: Routing module comparison between the original contraction hierarchies (CH) and parallel contraction hierarchies (PCH)

and storage space are measured over 100 000 random queries using the CH query code, as well as the respective file format. For CH the *aggressive* profile from [GSSD08] is used. Query times and storage space requirements do not differ much for the two variants. When restricted to 2 cores, PCH are only marginally faster during preprocessing. No preprocessing time is given for Europe (CH), as the program allocates huge amounts of memory and is forced to use swap excessively. With regard to memory consumption during preprocessing, PCH seem to have an edge over CH, especially since the additional memory used to store node coordinates is included in the figures.

	Preprocessing [s]	Memory [MB]	Storage [MB]
Germany	57	319	93
Europe	201	3 158	333

Table 5.10.: Address lookup preprocessing time and space consumption

		City Name		Street Name	
	Suggestions	Average	Max	Average	Max
Germany	16	15.56	182.86	1.67	64.49
	8	12.38	142.67	1.44	54.98
	4	11.58	104.42	1.31	54.86
	2	5.28	55.75	0.76	65.26
	1	4.93	54.13	0.76	53.95
Europe	16	23.97	208.78	1.51	53.97
	8	20.72	207.93	1.46	126.22
	4	20.49	200.98	1.23	27.38
	2	17.93	149.84	1.64	47.92
	1	15.69	143.43	1.34	68.91

Table 5.11.: Address lookup query times in milliseconds

5.8. Address Lookup

Table 5.10 shows the preprocessing time and storage space required by the address lookup module. The preprocessing is finished after 57s and 201s for Germany and Europe, respectively. Memory consumption is moderate, with 319MB for Germany and 3.1GB for Europe. Because path compression is only applied to the tries right before writing to the storage files, they occupy a comparatively large amount of main memory.

No cache is used, the storage files are directly accessed via memory mapped files. Table 5.11 shows the average and worst case query times for a set of random queries. We perform a sequence of queries for 1000 random city names, adding one letter at a time. For each chosen city name an analogous sequence of queries is performed for 10 random street names. The street name queries are faster, since the amount of unique street names in most cities is smaller than the amount of unique city names. The worst case occurs when no character is typed in yet. The query time is below the usual typing speed for mobile devices, even for the computation of as much as 16 suggestions. This means we are able to compute as many suggestions as can fit onto the screen, enabling the user to pick his target with less character inputs.

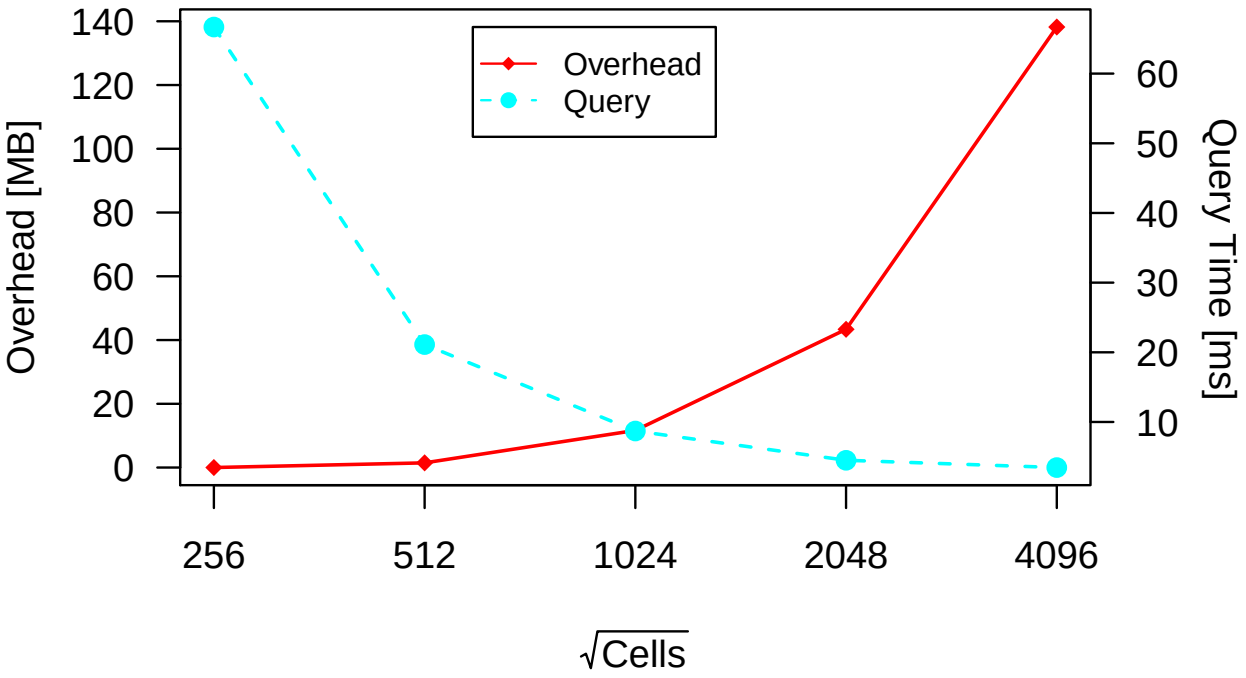


Figure 5.6.: Germany GPS lookup query time for different amount of cells

	Preprocessing [s]	Memory [MB]	Storage [MB]
Germany	63	617	107
Europe	84	3 058	465

Table 5.12.: GPS lookup preprocessing time and space consumption

5.9. GPS Lookup

Figures 5.6 and 5.7 display the query time for varying amounts of cells. Only the short drive is tested, since it lies completely within city boundaries, which is expected to be the most problematic case for the query. The space consumption of the index rises proportionately to the amount of cells, while the compression ratio decreases slightly for smaller cells. A grid of 1024x1024 cells is chosen for Germany and a grid of 4096x4096 cells for Europe. These sizes provide adequate query times and keep the amount of storage space required low.

The preprocessing itself only takes 63s and 84s for Germany and Europe, respectively, as seen in Table 5.12. For Germany, the memory usage is 617MB and the storage space 107MB. This scales well for the approximately 5 times larger European subset, requiring only 3GB memory and 465MB storage space.

Table 5.13 lists the query times in detail. The longer the drive, the faster the GPS lookup on average: On the German subset 8.71ms are required within city limits, while the long

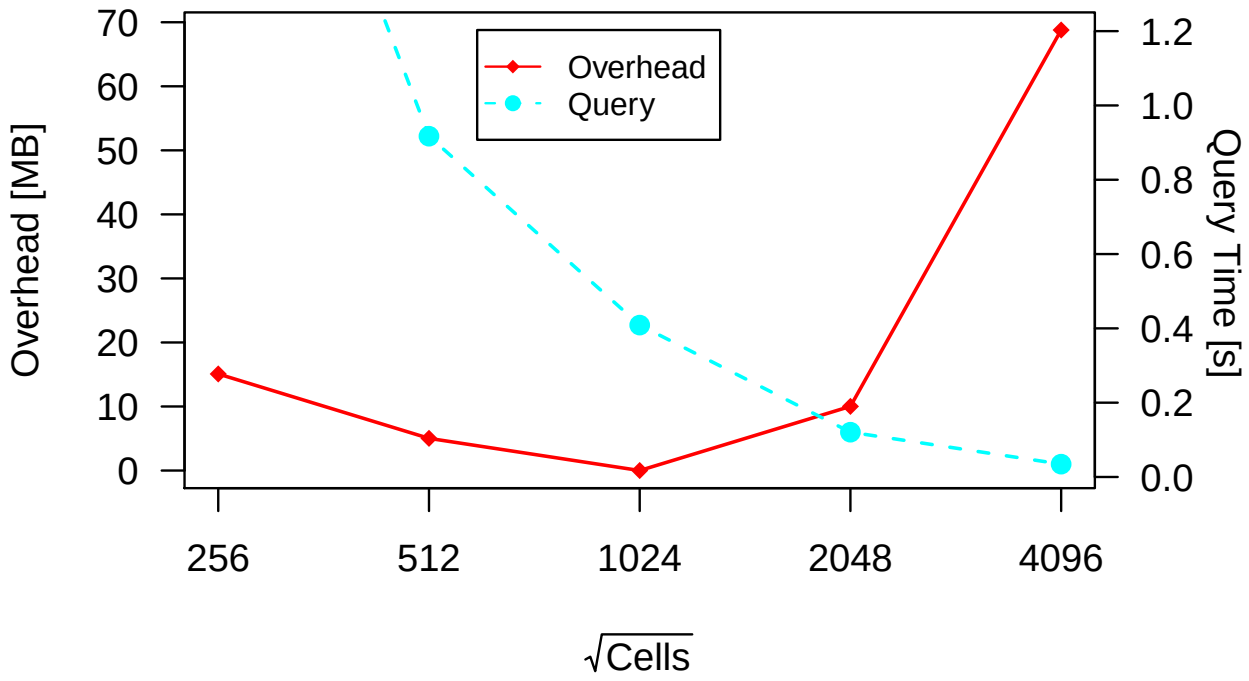


Figure 5.7.: Europe GPS lookup query time for different amount of cells

	Short	Mid	Long	Very Long
Germany	8.71	3.82	3.06	-
Europe	34.05	14.52	8.71	4.86

Table 5.13.: GPS lookup query times in milliseconds

range drive takes only 3.06ms. This is due to the low density of edges in the vicinity of motorways. The very long range drive on the European subset benefits even more from this, as the OpenStreetMap data is more sparse in Eastern Europe. Overall, queries on the European subset are slower than on the German one. It uses 16 times as many cells as the German subset, but covers an area roughly 103 times the size. Combined with the fact that the European subset is about 5 times larger, this means that each cell is bigger and filled with more edges. Additionally, the amount of empty cells increases, since the data set becomes more sparse.

Figure 5.8 reveals some disadvantages of fixed-size cells. All query times for the long range drive on the German subset are shown. Whenever the drive draws close to a city, or is located within city boundaries, query times spike. This is also the reason for the short range drive being slower than the long range drive on average.

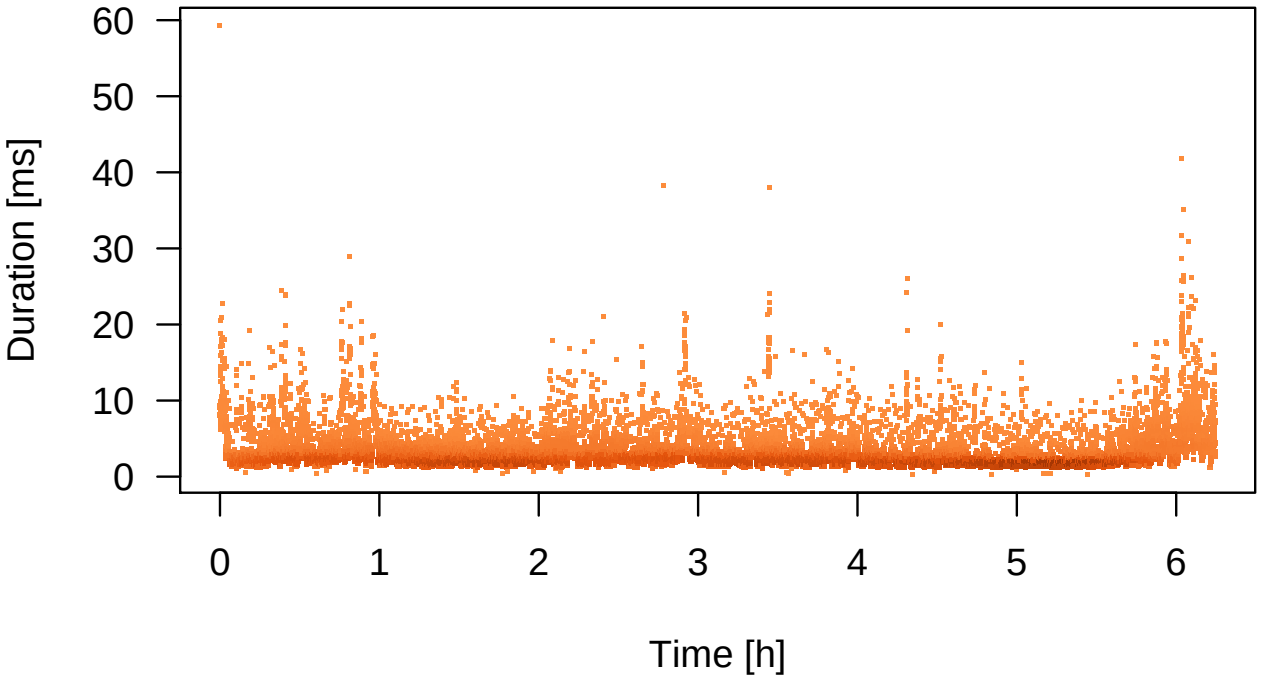


Figure 5.8.: GPS Lookup time for long range drive

6. Discussion And Future Work

The GPS lookup module has some scaling problems with large, sparse data sets that need to be addressed. An external kd-tree has some disadvantages: As the edges are not 2-dimensional objects, this would either require a 4d-tree or result in an increased amount of duplicate edges, both adding complexity. Another idea is to employ R-Trees [Gut88], more specifically variants which are optimised for external memory scenarios. The most promising seems to be Sort-Tile-Recursive [LEL97] due to its simplicity and speed. A further possibility is to keep the grid structure, but replace the index with a perfect hash function like HDC [BBD09]. This would allow us to keep the cells small enough for fast performances without running into trouble with index sizes.

It is also worth mentioning that the GPS lookup module uses a simple method to determine the best suitable edge: Whenever the GPS signal deviates more than a fixed distance from the current route, a new position on a routing edge is determined based on the GPS position and heading. This can be improved by using the history of GPS coordinates to determine the route the driver most likely has followed up to now. [NK09] and [DBJF07] show that a hidden Markov model can be employed to greatly increase accuracy.

The rendering module does quite well with regard to rendering speed, but is hindered mostly by long preprocessing times and enormous storage space needs. A vector based renderer would improve on the situation, but most likely sacrifice either detail or rendering speed and quality.

The routing module has no obvious disadvantages. It has fast preprocessing, moderate storage space needs, fast query times, scales well and is exact. In fact, it is so fast that it almost does not pay off to compute whether the driver strays from the route, instead of just recomputing the whole route. The variant introduced in this thesis even has very moderate memory consumption, allowing preprocessing of larger data sets on desktop computers. Moreover, support for traffic jams, roadblocks and the like can easily be added later on, sacrificing some storage space and query time in the process. There is room for improvements, though. Turning restriction and penalties could be included, or points-of-interest near the current route could be searched for. Furthermore, it might be beneficial to request only the first few kilometres, instead of the entire route. This would speed up the rendering, as less edges would have to be clipped. Additionally it could make the routing query faster. We would not need to unpack shortcuts which lie too far ahead.

The address lookup module is fast and space efficient, but lacks any kind of error robustness for invalid queries. Furthermore, the lack of street numbers in the module itself, as well as in the OpenStreetMap data, is in need of improvement.

One kind of module is missing for now: A driving instructions module. Spoken driving instructions are much more useful for the driver than a simple map view. As these require some voice acting and, in many cases, manual editing of the data set, this is beyond the scope of this work.

The OpenStreetMap data seems to be mature enough to be used for routing applications and not only pretty maps. Machine readability should definitely be improved. Some tagging schemes in use are simply not sensible in this regard.

Portability of the solution is great for any platform wxWidgets already supports. Only the GPS module and some system functions have to be adjusted. Porting to a platform not supported by wxWidgets is more difficult. The majority of the modules rely on wxWidgets classes and functionality. It might be useful to look into other cross-platform user interface kits and determine whether a larger variety of mobile devices is supported.

The file access should definitely be improved to close the gap to the unbuffered access. This would require changing the toolkit or introducing a custom file class into the system module, decreasing portability somewhat in the process. However, a custom class would have the advantage that it could use unbuffered file access internally, increasing random access speed and making it possible to simulate an empty file cache during a benchmark.

Appendix A.

Additional Data Structures

A.1. Addressable Priority Queue

Addressable priority queues are used by the routing module. During the preprocessing they are needed for the limited local Dijkstra searches. Since the query is just a modified bidirectional search with the addition of stall-on-demand, it also requires a priority queue.

Our priority queue is realised as a binary heap. To make this heap addressable, an additional index vector is kept, indicating the position of already inserted nodes in the heap. Furthermore, a lookup structure is used to find the index of a node.

Two different lookup structures can be chosen via template parameters. During the preprocessing the lookup structure should provide very fast lookup and resetting the queue should not be too costly. The query, on the other hand, should require as little memory as possible. In both cases we need to store temporary information for each node, e.g., distance and parent function. The priority queue is used to store this data only for the nodes already discovered during a search, which saves a lot of memory for the local searches as well as the query.

The first lookup structure for up to n nodes is a doubly-linked array. A dynamically sized array stores all i elements inserted up to now in the order they were added. A second array of size n stores each element's position in the first array. If an element has not been inserted so far, the information in the second array need not be correct, as we can infer it quite easily: First, we determine whether the stored position is valid, i.e., smaller than i and larger than -1. Then, we simply look up the element stored at that position. If it is the element in question we have found it, otherwise it has never been inserted. This data structure allows for element access in constant time. Inserting a new element can be done in amortized constant time. In addition to this, it can be reset very quickly by simply setting the size of the first array to 0. An example of a doubly-linked array is given in Figure A.1. Red arrows indicate invalid entries.

The second lookup structure is a simple self-balanced binary tree, more precisely the *map* data structure provided by the C++ STL. It should definitely be replaced by a hashmap in

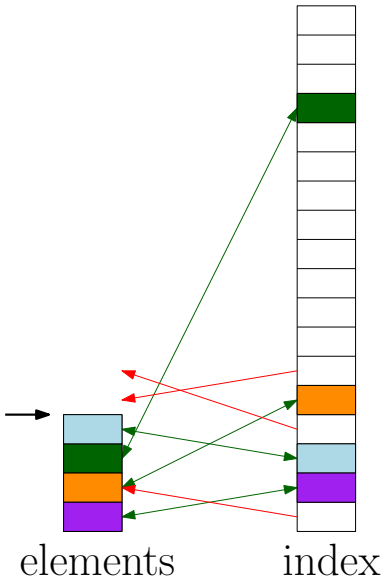


Figure A.1.: Doubly-linked array

the future. While most compilers already include one for the proposed ISO/IEC TR 19768 C++ Library Extensions, they do so in a compiler specific way. Hence, to improve portability, we postpone employing a hashmap until the standard has been implemented.

A.2. Dynamic Graph

Most types of routing plugins will require a dynamic graph. We used a simple dynamic graph, which provides fast operations, while wasting some memory when many edges are inserted and deleted. Our implementation is based on incidence arrays. One array of edges stores all outgoing and incoming edges for each node in a contiguous block. The second array stores for each node the starting position of its edge block, its amount of edges, and the capacity of its block. Deleting edges is done in constant time by swapping the edge to be removed with the last one in the block. When an edge is to be inserted into an already full buffer, the whole buffer is moved to the end of the edge array and buffer's capacity is doubled. A disadvantage of this approach is that space allocated for edges is never freed.

Parallel Contraction Hierarchies benefit from this kind of dynamic graph. After a bunch of nodes have been contracted in parallel most remaining nodes have several new adjacent shortcut edges. Therefore, after a few iterations most of the remaining nodes' data is grouped together at the end of the edge array. This increases cache efficiency of the local searches during a contraction.

Appendix B.

Additional Implementation Details

B.1. Floating Point Operations

Many mobile devices have a limited instruction set. Some are missing specific floating point operations, which consequently have to be emulated by the compiler. Furthermore, available floating point calculations tend to be slow when using a high resolution. Therefore, one of our implementation goals was to minimise the use of floating point operations.

GPS Coordinates. GPS coordinates are prominently featured in many modules. The GPS lookup needs to work with them, the routing module needs to return them and the rendering module needs to display them. In addition, the GPS lookup and the rendering module use GPS coordinates in their Mercator projected version. Therefore, an integer GPS coordinate class that stores coordinates in the Mercator projection is used throughout the project. Latitude and longitude represented by a 32bit unsigned integer each. This means that the resolution is worst at the equator, as the Mercator Projection enlarges areas further away from the equator. Thus, the worst case resolution on the Earth's surface is $\frac{40\,041.47}{2^{32}}\text{km} \approx 9.32\text{mm}$. This is precise enough for our purposes, considering that GPS receivers are inherently inaccurate.

GPS Lookup Module. Unless a very crude approximation or a custom fixed point class is used, the GPS lookup module can most likely not manage without floating point operations. The current implementation avoids floating point operations for as long as possible. Because the device we tested in this thesis displayed adequate floating point performance, we recalculate the distance to each approximated nearest point on an edge using the spherical approximation explained in Section 4.2. It might be beneficial to deactivate this when running the application on devices with limited floating point capabilities, sacrificing some accuracy in the process.

Rendering Module. The rendering module uses the fixed point anti-aliased rendering provided by the Anti-Grain Geometry library. The only remaining source of floating point operations is the rotation of the route's coordinates. This could be prevented if the image were to be rendered prior to rotating it. As mentioned in Section 4.3, this would also remove the artefacts at the tile boundaries. However, it would increase memory consumption and add overhead for rendering to another buffer. On systems that heavily penalise floating point operations it might still be beneficial.

B.2. Settings

Most modules provide the user with a set of settings. Because each module should be oblivious to the user interface, a centralised settings class is introduced. Each module can easily add various predefined types of settings to an instance of the class. The settings class then generates a settings window for the user interface to display. Changes to this window are automatically synchronized with the class. Furthermore, the chosen settings are stored according to the operating system's standard method: e.g., in the registry on Microsoft Windows variants or in an INI file on Linux based systems.

B.3. Unaligned Memory Access

Many modules compress their preprocessed data into streams of bits or bytes. Special care has to be taken when decompressing this data. In some cases it is tempting to use a pointer to a byte stream, cast it into a pointer to an integer stream and then extract the appropriate amount of bits or bytes all at once. However, many mobile devices do not allow unaligned data access. It is usually required that each data access is aligned to addresses divisible by 4. While compilers normally generate instructions to work around this when working with unaligned data types, they will not in this case. A pointer to an integer may be required to point to an aligned integer and therefore a pointer to a byte cannot be cast into a valid integer pointer. Any subsequent access to the pointer's target results in an exception, often crashing the application.

The alternative is to assemble the necessary data byte by byte. This is of course slower and therefore not desirable on devices that allow unaligned memory access. To avoid duplicating functionality in each plugin, a set of functions is provided which extract a specified amount of bits or bytes from a byte stream. These can easily be switched to a version respecting alignment. In fact, our Windows Mobile version has to work around the unalignment issue, as Windows Mobile has disabled unaligned memory access by default.

B.4. wxWidgets Bugs

wxWidgets features some troublesome bugs. First of all, the wxString class used by all wxWidgets classes is not thread-safe. A problem occurs when the wxWidgets logging feature is not turned off. If a wxWidgets class logs an item, e.g., an internal error, from another thread the program might crash. The only work-around is to disable logging completely.

The Windows Mobile port seems somewhat neglected and has many code artefacts from Windows CE. Some sizing events are lost, resulting in windows not being correctly updated, and recalculating the layout seems to fail in certain cases. Furthermore, the internal file function uses only a 32bit signed integer for the offset, making it impossible to access more than the first 2 GB of a file. This bug can be easily fixed in the wxWidgets source, though. Moreover, the wxWidgets time measuring functions have only a resolution of 1 second for some devices.

List of Tables

List of Figures

List of Algorithms

Bibliography

[ant10] (2010, Feb.) Anti-grain geometry. [Online]. Available: http://www.antigrain.com/ 27

[aos09] (2009, Jun.) Aosm. [Online]. Available: http://www.arktosmobile.de/aosm/ 4

[Bar84] B. A. Barsky, "A new concept and method for line clipping," *ACM Trans. Graph.*, vol. 3, no. 1, pp. 1–22, 1984. 43

[BBD09] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger, "Hash, displace, and compress," in *ESA*, ser. Lecture Notes in Computer Science, A. Fiat and P. Sanders, Eds., vol. 5757. Springer, 2009, pp. 682–693. 58

[BD09] R. Bauer and D. Delling, "SHARC: Fast and Robust Unidirectional Routing," *ACM Journal of Experimental Algorithmics*, vol. 14, p. 2.4, May 2009, special Section on Selected Papers from ALENEX 2008. [Online]. Available: http://doi.acm.org/10.1145/1498698.1537599 5

[BDSV09] G. V. Batz, D. Delling, P. Sanders, and C. Vetter, "Time-Dependent Contraction Hierarchies," in *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*. SIAM, April 2009, pp. 97–105. 6

[Bel58] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958. 5

[Ben75] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975. 17

[Ben90] ——, "K-d trees for semidynamic point sets," in *SCG '90: Proceedings of the sixth annual symposium on Computational geometry*. New York, NY, USA: ACM, 1990, pp. 187–197. 17

[Bre98] J. E. Bresenham, "Algorithm for computer control of a digital plotter," pp. 1–6, 1998. 41

[DBJF07] J. Drummond, R. Billen, E. Joao, and D. Forrest, Eds., *Dynamic and Mobile GIS: Investigating Changes in Space and Time*, ser. Innovations in GIS. CRC Press, 2007, chapter: Map Matching for Vehicle Guidance. [Online]. Available: http://www.crcpress.com/product/isbn/9780849390920 58

[Dij59] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959. 5

[DLB59] R. De La Briandais, "File searching using variable length keys," in *IRE-AIEE-ACM '59 (Western): Papers presented at the the March 3-5, 1959, western joint computer conference*. New York, NY, USA: ACM, 1959, pp. 295–298. 37

[DSSW09] D. Delling, P. Sanders, D. Schultes, and D. Wagner, "Engineering Route Planning Algorithms," in *Algorithmics of Large and Complex Networks*, ser. Lecture Notes in Computer Science, J. Lerner, D. Wagner, and K. A. Zweig, Eds. Springer, 2009, vol. 5515, pp. 117–139. 5

[Gei08] R. Geisberger, "Contraction Hierarchies," Master's thesis, 2008, http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/geisberger_dipl.pdf. 6

[GH05] A. V. Goldberg and C. Harrelson, "Computing the Shortest Path: A* Search Meets Graph Theory," in *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, 2005, pp. 156–165. 5

[gps10] (2010, Feb.) Gps intermediate driver. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb202086.aspx 24

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks," in *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, ser. Lecture Notes in Computer Science, C. C. McGeoch, Ed., vol. 5038. Springer, June 2008, pp. 319–333. 6, 7, 31, 52, 53

[GSSV09] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," 2009. [Online]. Available: mailto://geisberger@kit.edu 6, 36

[Gut88] A. Guttman, "R-trees: a dynamic index structure for spatial searching," pp. 599–609, 1988. 58

[Hai94] E. Haines, "Point in polygon strategies," pp. 24–46, 1994. 20

[HNR07] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, February 2007. [Online]. Available: http://dx.doi.org/10.1109/TSSC.1968.300136 5

[Knu98] D. E. Knuth, *The Art of Computer Programming 3. Sorting and Searching: The Classic Work Newly Updated and Revised*, 2nd ed. Addison-Wesley Longman, Amsterdam, June 1998. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201896850 37, 39

[Lau04] U. Lauther, "An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background," in *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung.* IfGI prints, 2004, vol. 22, pp. 219–230. 5

[LEL97] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez, "Str: A simple and efficient algorithm for r-tree packing," Tech. Rep., 1997. 58

[lib10] (2010, Feb.) The xml c parser and toolkit of gnome - libxml. [Online]. Available: http://xmlsoft.org/ 21

[LW77] D. T. Lee and C. K. Wong, "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees," *Acta Informatica*, vol. 9, no. 1, pp. 23–29, March 1977. [Online]. Available: http://dx.doi.org/10.1007/BF00263763 17

[map10] (2010, Feb.) Mapnik c++/python gis toolkit. [Online]. Available: http://mapnik.org/ 25

[nav10] (2010, Feb.) Navit - car navigation system. [Online]. Available: http://www.navit-project.org/ 4

[NK09] P. Newson and J. Krumm, "Hidden markov map matching through noise and sparseness," in *GIS '09: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* New York, NY, USA: ACM, 2009, pp. 336–343. [Online]. Available: http://portal.acm.org/citation.cfm?id=1653771.1653818 58

[osm10a] (2010, Feb.) Openstreetmap. [Online]. Available: http://www.openstreetmap.org/ 4

[osm10b] (2010, Feb.) Osm2pgsql. [Online]. Available: http://wiki.openstreetmap.org/wiki/Osm2pgsql 25

[pos10a] (2010, Feb.) Postgis. [Online]. Available: http://postgis.refractions.net/ 25

[pos10b] (2010, Feb.) Postgresql: The world's most advanced open source database. [Online]. Available: http://www.postgresql.org/ 25

[RT] F. Ramm and J. Topf, *OpenStreetMap*, 2nd ed. Lehmanns Media-Lob.de. [Online]. Available: http://www.worldcat.org/isbn/386541320X 16

[Shi62] M. Shimrat, "Algorithm 112: Position of point relative to polygon," *Commun. ACM*, vol. 5, no. 8, p. 434, 1962. 20

[SS05] P. Sanders and D. Schultes, "Highway Hierarchies Hasten Exact Shortest Path Queries," in *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, ser. Lecture Notes in Computer Science, vol. 3669. Springer, 2005, pp. 568–579. 5

[SSV08] P. Sanders, D. Schultes, and C. Vetter, "Mobile Route Planning," in *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, ser. Lecture Notes in Computer Science, vol. 5193. Springer, September 2008, pp. 732–743. 6, 48, 51

[TA06] The Unicode Consortium and J. Allen, *The Unicode Standard, Version 5.0*, 5th ed. Addison-Wesley Professional, November 2006. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321480910 40

[tra10] (2010, Feb.) Trackmyjourney. [Online]. Available: http://www.trackmyjourney.co.uk/ 4

[Vet09] C. Vetter, "Parallel Time-Dependent Contraction Hierarchies," 2009, student Research Project. http://algo2.iti.kit.edu/documents/routeplanning/vetter_sa.pdf. 6, 35

[Vin75] T. Vincenty, "Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations," *Survey Review*, vol. 22, no. 176, pp. 88–93, 1975. 23

[wxw10] (2010, Feb.) wxwidgets. [Online]. Available: http://www.wxwidgets.org/ 8