Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Extending HikeDroid — Smart Features for Hikers

Bachelor's Thesis

Dominik Landtwing

dominila@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Tobias Langner, Samuel Welten
Prof. Dr. Roger Wattenhofer

January 27, 2012

Acknowledgements

I would like to thank my supervisors **Tobias Langner** and **Samuel Welten** for giving me an opportunity to work on this interesting topic, answering my questions about HikeDroid's initial architecture, providing feedback to my suggestions, coming up with ideas for solutions when I found myself in a dead-end, finding bugs in HikeDroid and supporting me in my work. Despite both of them supervising multiple theses at the same time, they always managed to help me with arising problems.

Furthermore I would like to thank **Damian Pfammatter** for developing and documenting HikeDroid 1.0, which is the foundation for this thesis' work, and the developers of the following libraries:

- igraph[1]
- OpenCV[2]
- mahotas[3]
- AndroidPlot[4]
- Simple[5]
- JSI[6]

Without their work, HikeDroid 2.0 would never had been at the point it is right now.

[1] http://igraph.sourceforge.net/
[2] http://opencv.willowgarage.com/
[3] http://luispedro.org/software/mahotas
[4] http://androidplot.com/
[5] http://simple.sourceforge.net/
[6] http://jsi.sourceforge.net/

Abstract

This document is a technical report on the development of HikeDroid 2.0, an advanced hiking application for the Android mobile platform developed as part of a bachelor's thesis at ETH Zürich. It describes the extension of a basic mapping application with multiple features such as *digital elevation models* and *routing*, both in terms of design and implementation.

The major part of this document covers the computation of a graph representation of hiking trails (in Switzerland) on the sole basis of raster image data. Said graph is then used to perform route computations for hikers between any two locations in Switzerland.

Keywords: Android, Routing, Shortest Path, Image Graph Reconstruction, Digital Elevation Model, Hiking

Contents

Chapter 1

Motivation and Goals

1.1 Motivation

In Switzerland, hiking is a very popular activity[1]. One reason might be that the geographic and infrastructural conditions for hiking are close to optimal due to the many hills and mountains and a large network of hiking trails.

When hiking, the following problems or issues often arise:

1. As hikers don't just follow arbitrary paths, hiking does also involve some sort of navigation, typically done using printed maps. While manual navigation usually works well, it is time-consuming and sometimes error-prone (especially for people with bad sense of orientation).
2. In addition to navigation, some hikers do also keep track of the routes they walked. Manual drawing of the followed routes is possible, but error-prone. Additionally, management of those routes is cumbersome.
3. Last but not least, hikers might be interested in the altitude differences on a particular route, either to estimate whether a certain route is feasible for them or just out of sportive ambitions.

Note that most of the above description does also apply to similar activities such as mountainbiking, which is also part of the target audience of HikeDroid.

1.2 Goals

The main focus of this thesis is the extension of HikeDroid with three features help solving the problems listed in the section above: Routing, Elevation Profiles and Trail Recording and Management.

[1] Official statistics are hard to find, but according to http://www.revueschweiz.ch/dokumente/upload/8b460_schweiz_media_2012_1_mail_d.pdf, the leading swiss hiking magazine has 124'000 regular readers.

1.2.1 Routing for Hikers

With the widespread use of mobile devices such as smartphones and tablets and the availability of GPS and internet access, the task of manual navigation using printed maps can be replaced. HikeDroid is an attempt to make hiker's lifes easier. While HikeDroid 1.0, the starting point for this thesis, is a basic mapping application replacing printed maps by a digital one, HikeDroid 2.0 shall further assist hikers with planning trips by introducing routing features.

1.2.2 Elevation Profiles

HikeDroid 2.0 shall provide means to display a route's elevation profile without the use of an internet connection.

1.2.3 Route Recording and Management

Instead of having to manually draw routes onto maps, HikeDroid 2.0 shall facilitate this task by allowing users to record their locations (and thus their routes) using the GPS receiver built into most recent mobile devices. Furthermore it shall allow simple management of recorded routes and import/export from/to other geographic information system software.

Chapter 2

HikeDroid 1.0

2.1 Introduction

HikeDroid is a mobile application based on the Android mobile platform. As its name suggests, its main target audience are hikers and groups with similar interests such as mountainbikers.

HikeDroid in its current state is the result of two bachelor's theses. Where it is necessary, the results of both theses will be distinguished as follows: The application resulting from the first thesis will be referred to as "*HikeDroid 1.0*", whereas the current application will be referred to as "*HikeDroid 2.0*" (although those are not official version numbers). In all other cases, the term "*HikeDroid*" is used.

The main feature in *HikeDroid 1.0* is the display of an detailed interactive (i.e. pan-and-zoomable) map of Switzerland, with an optional display of hiking trails. Both the map and hiking trail data is provided as raster graphics by the Swiss Federal Office for Topography (swisstopo)[1]. It is therefore clearly aimed at people in Switzerland.

HikeDroid in its core is similar to the well-known Google Maps[2] Application for Android. The differences are explained in [1].

HikeDroid was initially written by Damian Pfammatter as a part of his bachelor's thesis [1].

2.2 Architecture Overview

The following sections provide a short overview HikeDroid's main components and how they interact. As a visual aid, a simplified UML class diagram is shown in figure 2.1.

[1] http://www.swisstopo.ch/
[2] http://www.google.com/mobile/maps/

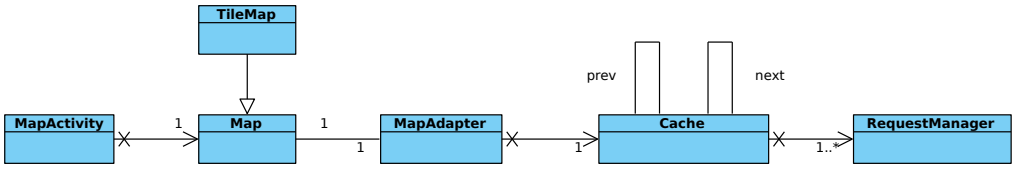


Figure 2.1: Simplified UML class diagram of HikeDroid 1.0

2.2.1 Map Activity

The map activity is HikeDroid's start and main activity. It contains a map display and provides the user with menus such as a selection of displayed layers and handles application startup and shutdown.

2.2.2 Map

The map is an interactive UI component that handles all user events such as panning and zooming. It does ensure that map images corresponding to the currently selected area are eventually displayed on the map, e.g. when a user zooms in, it requests higher resolution images from the *MapAdapter* asynchronously.

A map in HikeDroid is divided into multiple tiles of 256×256 pixels each, allowing it to load separate parts of the map independently.

The map component is the most complex component in HikeDroid as it also deals with a number of other issues such as:

- Coordinate transformations
- Layering
- Tile management
- Asynchronous drawing

2.2.3 Map Adapter

The map adapter is the component that handles all communication between the *Map* and the *Cache* transparently. It maintains data structures that contain information about which tiles have been requested by the map.

It forwards requests to the cache and, as soon as a requested tile is ready, issues a map redraw.

2.2.4 Cache

HikeDroid's cache is built in a hierarchical way. It currently consists of the following caches, in their hierarchical order:

1. Bitmap Cache (a cache holding the actual image data in memory)
2. Byte Array Cache (a cache holding compressed image data in memory)
3. SD Cache (a cache holding compressed image data on the flash storage)
4. Web Cache (not actually a cache, simply downloads files from SwissTopo)

Caches at the top end are faster than caches at the bottom end. The hierarchy is implemented as a doubly-linked list. When a tile is requested, a request is passed to the Bitmap Cache by the Map Adapter.

When a cache receives a request, it checks whether it holds a valid copy of the requested tile. If it does, it propagates the copy to its parent cache (or, the MapAdapter, if the current cache is at the top). Otherwise, it requests the tile from its child cache, which is done asynchronously using one of its RequestManagers.
When a cache receives a tile copy from its child cache, it proceeds to store it itself and propagate it upwards in the hierarchy.

2.2.5 Request Manager

A request manager consists of a worker thread and a LIFO queue. The worker simply waits for new work items to arrive at the queue and executes them, one after another.

Note that this is similar to the concept of a single-threaded executor.

2.3 User Interface

HikeDroid 1.0's user interface consists of a pan-and-zoomable map and a few context menus for layer selection and similar tasks. Its operation is very similar to the well-known Google Maps application for Android.

Figure 2.2 shows a screenshot of HikeDroid's user interface.

2.4 Shortcomings

HikeDroid 1.0 has several architectural and some functional shortcomings, including:

Figure 2.2: HikeDroid 1.0 User Interface [1]

- Tight coupling
- Dependencies on tile based maps throughout the system
- Some large, monolithic classes (especially *Map*)
- Race conditions in cache mechanism
- Aliasing bugs
- Inconvenient[3] implementation of thread pools (Request Manager)
- Malfunctioning GPS tracking functionality
- Very inefficient spatial indexing of points
- Fragile and inconvenient identification of tiles by a string encoding of zoom level and tile index

Some of them have been corrected in HikeDroid 2.0 as explained in chapter 3.

[3] E.g. it is not possible to determine whether a job has completed.

Chapter 3

Added Features and Architectural Changes

3.1 Added Features

3.1.1 Spatial Indexing of Objects

Some of the features described in the following sections require that objects (e.g. trails or points) can be drawn onto the map. Obviously those objects need to be stored somewhere. Note that the user only sees a small (rectangular) section of the map most of the time and that, for efficiency reasons, drawing objects outside the visible map section is undesirable.

A naive approach would simply store objects in a collection, such as a list. To find all objects in the currently displayed map section, one would simply iterate over the collection and collect all objects inside the section by comparing coordinates. The naive approach requires $\mathcal{O}(n)$ space and lookups are performed in $\mathcal{O}(n)$ time, where n is the number of objects to be indexed.

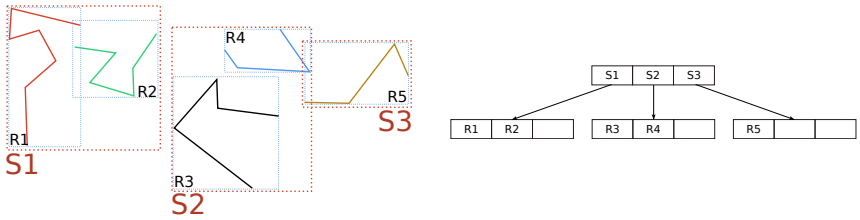
For small n the naive approach might work well. For large n, a time complexity of $\mathcal{O}(n)$ for a lookup is too expensive.

In HikeDroid 2.0, R-Trees are used for fast indexing of map objects.

R-Trees

An R-Tree is, as the name suggests, a balanced tree data structure for spatial indexing of multi-dimensional information[2]. In its basic form an R-Tree stores nested bounding boxes of spatially close objects as inner nodes. Objects are stored as tree leaves.

An example of an two-dimensional R-Tree of page size 3 containing five trail objects is shown in figure 3.1.



(a) Five trails with corresponding bounding boxes

(b) Corresponding R-Tree

Figure 3.1: 2D R-Tree containing five trails

Note that an intersection query for S3 would have to examine both S2 and S3, returning R4 and R5 as results.

While worst-case lookup performance for R-Trees is still $\mathcal{O}(n)$ with n being the number of objects stored, average-case lookup complexity is $\mathcal{O}(\log n)$[3]. Additionally, R-Trees are well-suited for dynamic data due to good insert performance.

In HikeDroid 2.0 R-Trees are currently only used for indexing trails.

Implementation

HikeDroid uses an R-Tree implementation from the Java Spatial Index (jsi) library[1], which in turn is based on the GNU Trove library for Java[2].

Trails are not indexed as points but rather as rectangles determined by their bounding boxes. Trails possibly contained in the currently displayed map section are then obtained via an intersection query. WGS-84 coordinates are used for indexing and distance calculation.

Definition 3.1 (WGS-84 Coordinates) *WGS-84 coordinates are tuples* (λ, ϕ) *where* λ *and* ϕ *denote a position's latitude and longitude in the World Geodetic System 1984 (WGS-84).* ◇

WGS-84 is the de-facto standard reference system for applications working with geographic coordinates. A prominent application using WGS-84 is the Global Positioning System (GPS) used for navigation in cars and aircraft.

3.1.2 Trails

HikeDroid 2.0 introduces the concept of trails, that is, a sequence of waypoints. Trails are used in various places in HikeDroid:

[1] http://jsi.sourceforge.net/
[2] http://trove4j.sourceforge.net/

- Recorded GPS data is stored as a trail (see section 3.1.3).
- Routes returned by the routing service (see chapter 6) are represented as trails.
- Trails can be easily persisted in the GPX format (see section 3.1.2) for simple import and export.

Management

Users are given the ability to manage their trails, that is, assign a name to them, list them and store them persistently with the option of displaying and deleting.

Trail management is implemented as a separate activity with a context menu offering options available for a specific trail.

Display

Recorded trails can be displayed on the current map. They are drawn on a separate layer by drawing colored points at each recorded waypoint. These points are then connected using colored lines.

Screen Coordinate Computation Since all waypoints are represented as WGS-84 (latitude, longitude) coordinates and the map display is based on pixel coordinates, coordinates need to be transformed.

Pixel coordinates are computed via linear interpolation as follows:

$$x = w \cdot \frac{\lambda - \lambda_{min}}{\lambda_{max} - \lambda_{min}} \tag{3.1}$$

$$y = h \cdot (1 - \frac{\phi - \phi_{min}}{\phi_{max} - \phi_{min}}) \tag{3.2}$$

x, y	Object's pixel offset (horizontal and vertical, respectively)
w, h	Map display size (width, height) in pixels
λ	Object longitude
λ_{min}, λ_{max}	Longitude at the left and right side of the displayed section
ϕ	Object latitude
ϕ_{min}, ϕ_{max}	Latitude at the bottom and top of the displayed section

Note that the equation for y is an inverted interpolation since pixel coordinates are originated at the top left corner of the screen.

This computation is not entirely accurate since 1 degree of latitude distance is not the same as 1 degree of longitude distance, except at the equator. In practice however, the inaccuracy introduced by this fact is on a sub-pixel scale for Switzerland, i.e. the above approximation is good enough for mapping purposes.

Storage

Recorded trails need to be persisted to be useful. The most obvious ways to store trail data are either in a database or as a file. While Android comes with SQLite support, database storage is more complex to implement and complicates archival and import/export of existing trail data.

Therefore a file-based approach is preferrable. In order to allow for simple exchange of data between HikeDroid and other programs, XML-based representations are a good choice. At the time of writing, two formats dominate: The Keyhole Markup Language (KML) and the GPS Exchange Format (GPX). GPX was chosen over KML due to KML not supporting timestamping in its standard representation and timestamping is a key requirement in trail recording.

GPS Exchange Format The GPS Exchange format (GPX) is an XML-based document format for the storage of *waypoints*, *routes* and *tracks*.

Definition 3.2 (Waypoint) *A waypoint is a tuple (λ, ϕ, h) representing a point in the WGS-84 coordinate system, where λ and ϕ denote the point's latitude and longitude respectively and h denotes the point's altitude above mean sea level.* ◇

Definition 3.3 (Track segment) *A track segment is a sequence of spatially and temporally close waypoints, each attributed with a timestamp.* ◇

Definition 3.4 (Track) *A track is a sequence of track segments. In a HikeDroid context, tracks are equivalent to trails.* ◇

Definition 3.5 (Route) *A route is a sequence of waypoints.* ◇

Routes are not used at all in the current HikeDroid implementation. Additionally, *tracks* are constrained to contain only a single *track segment*. An example GPX document produced by HikeDroid is shown in listing 3.1.

```
<gpx creator="HikeDroid" version="1.1"
        xmlns="http://www.topografix.com/GPX/1/1"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <trk>
      <name>sh</name>
      <trkseg>
         <trkpt lat="47.196985" lon="8.488396">
            <time>2011-11-20 11:39:32.14 MEZ</time>
         </trkpt>
         <trkpt lat="47.192635" lon="8.486911">
            <time>2011-11-20 11:39:41.23 MEZ</time>
         </trkpt>
```

```
            <trkpt lat="47.189189" lon="8.489601">
                <time>2011-11-20 11:39:53.30 MEZ</time>
            </trkpt>
            <trkpt lat="47.201899" lon="8.494274">
                <time>2011-11-20 11:40:14.57 MEZ</time>
            </trkpt>
            <trkpt lat="47.213032" lon="8.49398">
                <time>2011-11-20 11:40:23.64 MEZ</time>
            </trkpt>
            <trkpt lat="47.221627" lon="8.497823">
                <time>2011-11-20 11:40:32.68 MEZ</time>
            </trkpt>
        </trkseg>
    </trk>
</gpx>
```

Listing 3.1: A simple GPX document produced by HikeDroid with six waypoints

GPX parsing At the time of writing, no free lightweight Android-compatible GPX parser was available. While one could automatically parse files into objects using the GPX schema and JAXB[3], the parsing library is rather heavyweight[4]. Parsing directly using the `java.xml.*` classes would lead to a lightweight parser, but is very cumbersome to develop and maintain and is therefore out of the question.

HikeDroid uses the Simple[5] XML parsing library to achieve both lightweight and maintainable parsing. Simple requires little space[6] and facilitates parser development by using Java annotations.

3.1.3 GPS Logging

HikeDroid 2.0 comes with GPS logging functionality for recording trails and storing them as GPX files.

Logging Service

The main logging component is implemented as an Android `Service`.[7] This is a clean way to separate it from the rest of the application logic and prevent-

[3] http://jaxb.java.net/
[4] 10.5 MB for JAXB 2.2.4
[5] http://simple.sourceforge.net/
[6] Less than 370KB for version 2.6.2
[7] A service is a component running in the background without a user interface, much like a daemon process.

ing strong coupling. Additionally, a service approach can help increase battery running time: It can be run separately from the application.

Since a GPS receiver also consumes considerable amount of power, the logging service attempts to reduce power consumption by setting an update interval of 60 seconds, which corresponds to 100 meters walking distance for a hiker at an above-average speed of 6 km/h. Since the horizontal error of typical GPS receivers is within 15 meters (95% confidence)[4] it does not make sense to retrieve updates much more frequently.

While setting the GPS update interval to 60 seconds is a reasonable choice for most hikers, this is not necessarily the case for other activities such as mountainbiking. The logging service should thus be improved by allowing users to set the update interval themselves.

3.1.4 Elevation Data

Altitude data obtained via GPS is not sufficiently accurate for some purposes. To be more precise, GPS altitude errors for standard devices are even larger than horizontal errors. Sometimes altitude data might not even be available due to lack of receiver support or simply lack of GPS measurements.

Digital elevation models such as the one obtained by the Shuttle Radar Topography Mission (SRTM) from NASA are a more accurate way of determining the altitude at a certain location.

In the following sections a description of HikeDroid's SRTM integration is provided.

SRTMv3 Data

Data obtained by the "Endeavour" space shuttle in its Shuttle Radar Topography Mission (2000) is freely available from multiple sources[8] in multiple formats at a three-arc-second (about 90m) resolution for most locations[9] in the world, including Switzerland.

Challenges

Storing and querying a complete digital elevation model of Switzerland brings up two major problems:

[8] HikeDroid downloads SRTM data from
http://dds.cr.usgs.gov/srtm/version2_1/SRTM3/

[9] All areas between 58°S and 60°N

- Storage space: Depending on the resolution and data format, a digital elevation model can take up several 100MB of data. Data sets of such size must be avoided, since storage space on mobile devices is still limited. Furthermore, the data must be downloaded and downloading of large data sets is still slow on mobile devices, leading to user annoyance.
- Memory: Processes on Android have a memory limit of 16 MB[10], thus loading the complete model into memory is out of question.

Both of these problems are solved by dividing the elevation data into tiles, storing data as plain (16-bit) integers and compressing the tiles using ZIP.

The elevation data used by HikeDroid is divided into tiles of 1°, that is, 1201×1201 sized matrices of unsigned 16 bit integers in *big-endian byte order*. Files are named after their lowest left value's coordinate, e.g. the file named "N46E007.hgt.zip" contains the matrix for the area between 46°N and 47°N latitude and 7°E and 8°E longitude.

For Switzerland, the elevation data consists of 18 tiles[11], each requiring 2.9 MB when uncompressed. When ZIP-compressed, the model requires a total of 28.9 MB to store.

SRTM Altitude Provider

An *altitude provider* is a component that, given geocoordinates (λ, ϕ), will return an altitude value.

Interpolation As SRTM data is only a grid of altitude values and looked up coordinates will most likely not coincide with a grid intersection point, the altitude value has to be interpolated. Three ways of interpolation are:

1. Nearest grid value: Round coordinates to the nearest arc-second and return the altitude value at that location.
2. Mean of surrounding values: Compute the altitude at a location as the mean of its four surrounding values from the grid.
3. Least squares: Fit a plane through the four surrounding grid values using least squares and interpolate its altitude by computing the plane's value at the given location.

In HikeDroid, approach 2 is implemented.

[10] http://developer.android.com/reference/android/app/ActivityManager.html#getMemoryClass%28%29

[11] 45°N-48°N (3 tiles), 5°E-11°E (6 tiles), i.e. $3 \times 6 = 18$ Tiles

Caching Subsequent lookups of altitude values often query spatially close coordinates. This is for example due to the fact that trails typically contain spatially close waypoints. It is thus very likely that the next lookup will query for a location in the same tile as the current lookup.

Based on the described observation, a simple 1-slot caching mechanism is implemented:

- Upon startup, the altitude provider allocates a memory slot for a single tile[12].
- When the first lookup request is executed, the altitude provider determines which tile contains the altitude for the coordinates, unzips it and loads it into the memory slot. It does also store information about the currently loaded tile.

  After loading the tile into memory, it looks up the surrounding coordinate values and interpolates them, returning the interpolated value as a result.
- When the next lookup request is executed, the altitude provider again determines which tile to use. If the determined tile matches the one currently held in memory, it directly proceeds with the interpolation, otherwise it first replaces the memory slot's contents with the determined tile's data.

Data Download

Elevation data might not be needed by all users and is therefore not directly delivered with HikeDroid. However, whenever users try to access a feature that depends on elevation data (such as the display of a track's elevation profile), the user is given the ability to download them directly from the device.

For the above purpose, a simple (universal) download dialog was implemented. Given a collection of `DownloadJobs`[13], the dialog performs a sequential download of all Jobs and displays progress in two `ProgressBars`, as shown in figure 3.2.

3.1.5 Elevation Profile

One of the goals for this thesis is the development of an elevation profile display for trails.

With the ability to retrieve altitude values for all locations in Switzerland using the elevation model described in section 3.1.4, the elevation profile display

[12] about 2.9 MB

[13] A DownloadJob is a description of the URL of the file to be downloaded and its download destination.

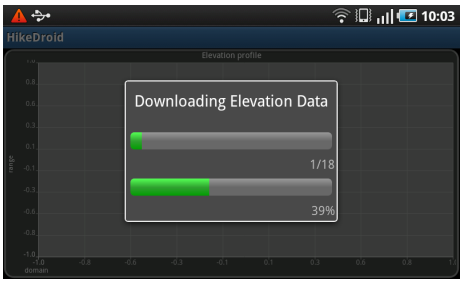


Figure 3.2: Download dialog for elevation data

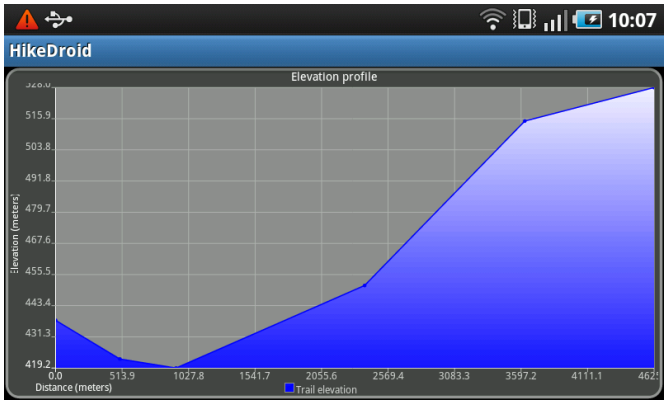


Figure 3.3: An elevation profile plot for a simple trail

boils down to plotting altitude values. To reduce coupling, this is implemented as a separate activity, displaying a graph with the waypoint distances on the x-axis and the altitude values on the y-axis. A sample plot is shown in figure 3.3.

The elevation profile plot is implemented using AndroidPlot[14].

3.1.6 Routing UI and Services

While actual routes are not computed on the device itself for efficiency reasons, a user interface for route selection is still needed to make use of routing services.

HikeDroid's routing UI makes use of the touch screen capabilities in all Android devices by letting users directly point at the route start and destination. To not interefere with the map's pan-and-zoom functionality, HikeDroid distinguishes between short touch events (to pan/zoom) and long touch events (to select route start and destination).

At the time of writing, HikeDroid does not support the use of waypoints other

[14] http://androidplot.com/

than start and destination. However, such a feature could be easily integrated into the existing application.

HikeDroid implements two different routing services, namely the service that is based on Swiss hiking maps and developed as part of this thesis (see chapters 5 and 6) and the Google Directions service[15].

3.1.7 Other Improvements

During the development process, some of the shortcomings of HikeDroid 1.0 were corrected, including:

- A race condition in the caching system
- An aliasing bug in MapAdapter
- Missing persistence of layer selection
- Improvement of responsiveness (side effect of architectural changes)

3.2 Architectural Changes

During the development of HikeDroid 2.0 several architectural problems started hampering the development process and were thus fixed. The two most important changes are described in the following sections.

3.2.1 Replacement of Request Managers

In section 2.2.5 it was noted that Request Managers are essentially executors. However, when compared to other executor interfaces such as `java.util.concurrent.Executor`, they offer much less flexibility: For example, it is only possible to get a call's result via a callback and it is not possible to determine whether a task has finished. Furthermore, there are implementations of `Executor` that can balance their thread count according to the current load[16].

In HikeDroid 2.0, caches only have a single Request Manager that is implemented as an adapter for a Thread Pool Executor of variable size. This replacement significantly improved responsiveness of the mapping user interface when panning and zooming.

[15] http://code.google.com/apis/maps/documentation/directions/

[16] This covers the point "Dynamic Request Manager Instantiation" described in the "future work" of the first HikeDroid thesis[1]

3.2.2 Reduce coupling

While HikeDroid 1.0 was initially designed to support multiple types of maps, the only map implemented is a tile-based map. This is also reflected in the source code: At many points, implementations of presumably map type independent classes simply require tile-based maps to work, either by requiring a "Tile ID" (a string for tile identification) or explicitly casting instances of Map to TileMap. This is especially reflected in the layering-related functionality and the caching mechanism.

The existence of components depending on the internals of other components is commonly referred to as "coupling" and is considered harmful in software engineering.

For HikeDroid 2.0, the dependency on tile-based maps was removed from layering-related functionality, achieving reduced coupling. However, some parts of HikeDroid 2.0 still depend on tile-based maps and could not be tackled due to lack of time.

Chapter 4

Testing

In HikeDroid 1.0, all testing was done manually, imposing a large burden on developers, especially in regression testing. With HikeDroid 2.0, most added functionality was designed with testing in mind and a unit test suite was set up to enable fully automated testing.

The following features in HikeDroid 2.0 are subject to unit tests:

- Altitude Providers
- Digital Elevation Models
- GPX parsing
- Routing

Unit tests in HikeDroid 2.0 are implemented on top of the JUnit[1] testing framework. Not only HikeDroid 2.0 was unit tested, but also parts of the hiking graph computation and routing service, namely:

- All coordinate transformations (between WGS-84, CH1903 and tile coordinates)
- All distance calculations based on coordinates

Since both the computation and routing service are implemented in Python, unit testing was implemented using the built-in unittest module[2].

Note that the described unit tests are only a first step in the right direction. A lot of code, especially from HikeDroid 1.0, is still not automatically tested. Especially most Android-specific components such as Activities, Views and Services are only tested manually so far. This can and should be changed by introducing the use of the Android testing framework[3].

[1] http://www.junit.org/
[2] http://docs.python.org/library/unittest.html
[3] http://developer.android.com/guide/topics/testing/testing_android.html

Chapter 5

The Hiking Graph

5.1 Problem Statement

As previously mentioned, the hiking trail map used in HikeDroid is provided as raster graphics, Portable Network Graphics (PNG) more precisely. The map is partitioned into tiles of size 256×256 px.

In each tile, the hiking trails are drawn as multi-pixel, orange or red lines[1] on a transparent background, so they can be directly drawn onto the roadmap. An example of such a tile is shown in figure 5.1.

Since it is unclear how to directly and efficiently perform tasks such as shortest path computation on raster images, it is desirable to obtain a more convenient and efficient representation of hiking trails, namely a graph.

Let a *branching point* be a point in the trail image where two or more trails cross. The problem that needs to be solved can thus be described as follows:

Definition 5.1 (Hiking Graph Reconstruction Problem) *Given an image I divided into a set of tile raster images $i_{1,1}, i_{1,2}, i_{1,3}, \ldots, i_{2,1}, \ldots, i_{m,n}$, extract a*

[1] Orange lines denote regular hiking trails, red lines denote mountain trails

Figure 5.1: An typical example of a tile containing hiking trails (orange)

hiking graph $G = (V, E)$ containing vertices where there are branching points in I and distance-weighted edges between all branching points that are connected in I. m and n are the number of horizontal and vertical tiles in I respectively. $\diamond$

5.2 Computation

5.2.1 Challenges

In order to obtain the graph described previously, several problems need to be solved:

- Detection of *branching points*
- Detection of edges between *branching points*
- Computation of edge distances
- Transformation from tile coordinates to a global coordinate system (WGS-84)

In addition to the problems listed above, a storage problem arises: Putting all tiles together to a single image in memory scales poorly (namely $\mathcal{O}(m \cdot n)$, m and n being the number of horizontal and vertical tiles respectively). At a map scale of 1:50000 this would require about 10 GB of memory. Furthermore most library routines are not programmed with images of such size in mind and will take forever to complete.

To solve the storage problem, the graph computation explained in the following sections works on a per-tile-basis and only combines all results at the very last stage.

5.2.2 Process

The following sections describe the graph computation process in the order it is performed in the current implementation.

Thresholding

In preparation for the following thinning step, the tile image needs to be converted to a binary image, that is, an image with black background and white edge pixels. This is done by setting all transparent pixels to black and all colored pixels to white.

(a) A branching point (b) A regular edge point

Figure 5.2: Branching points compared to edge points

To perform this task efficiently, OpenCV[2] routines are used in the implementation.

Thinning

In order to detect branching points in the following step, edges need to be thinned, i.e. all edges must be exactly one pixel wide. A possible method to achieve thinned edges is non-maximum-suppression as used in the popular Canny edge detector[5].

In the implementation, edge thinning routines from the mahotas[3] image processing library are used.

Finding Branching Points (Subgraph Construction)

The probably most interesting part of graph computation consists of finding the branching points in an image.

Intuitively, to find branching points, one would try to follow paths in an image and create a branching point wherever a path splits. This is the basic idea behind the algorithm used.

As the image's lines have been thinned in the previous step, a branching point is a pixel containing more than one unvisited pixel in its neighbourhood[4]. An example of a branching point is shown in figure 5.2.

[2] http://opencv.willowgarage.com/wiki/
[3] http://luispedro.org/software/mahotas
[4] A pixel's neighbourhood is the set of its adjacent pixels

The algorithm used for subgraph construction can be seen as an application of Depth-First-Search (DFS) to thinned images. Essentially, starting from a designated starting pixel, DFS first marks the current pixel as visited. It then calls DFS recursively on all *unvisited* edge pixels in the neighbourhood (see figure 5.2). As a result, a vertex object containing a list of adjacent vertices (obtained from the recursive DFS call on neighbour pixels) is returned.

The pseudocode for the subgraph construction is shown in algorithm 1 (distance computation is not considered to simplify matters). An example computation is illustrated in figure 5.3 with a red dot pointing out the position of the DFS routine at step i and gray lines denoting already visited pixels.

Algorithm 1 Pseudocode for the subgraph construction algorithm

```
function ConstructSubgraph(I)
    img ← copy(I)
    g ← empty Graph
    start_px ← find_start_pixel(img)
        ▷ Returns an unvisited pixel's coordinates in img if one exists, ∅ otherwise
    while start_px ≠ ∅ do          ▷ A tile may contain multiple unconnected trails
        tree_root ← DFS(img, start_px)
        tree_root.visit(g.add_vertex)
        tree_root.visit(g.add_edges)
        start_px ← find_start_pixel(img)
    end while
    return g
end function

function DFS(img, coords)
    mark(img, coords)                          ▷ Mark current pixel as visited
    neighbours ← get_unvisited_neighbours(img, coords)
        ▷ Returns all unvisited pixels north, east, south or west of the current pixel
    if |neighbours| = 1 then
        return DFS(neighbours[0])              ▷ Non-branching points are skipped
    else
            ▷ Either no neighbours (dead end) or more than one (branching point)
        v ← new Vertex
        v.adjacent ← ∅
        for all n ∈ neighbours do
            successor ← DFS(n)
            v.adjacent ← v.adjacent ∪ {successor}
        end for
        return v
    end if
end function
```

Vertex.visit is a simple visitor pattern application that first calls the passed function on the vertex itself and then on all adjacent vertices.

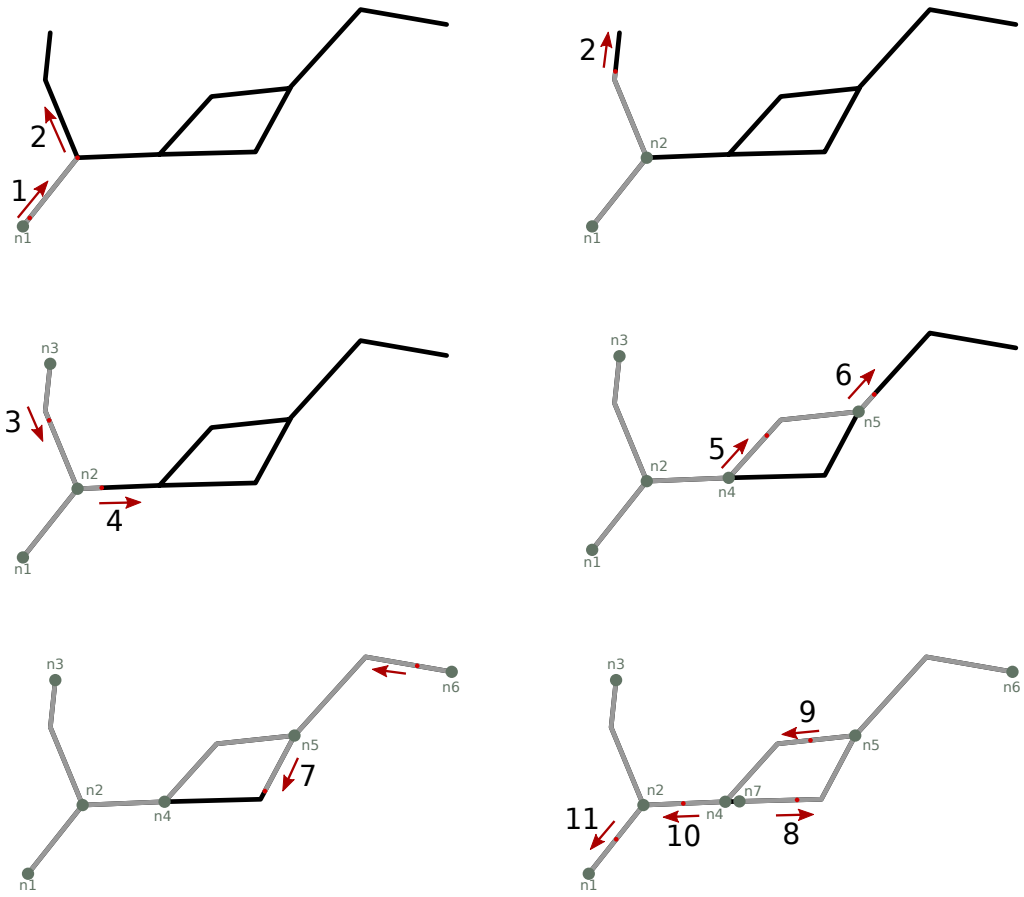


Figure 5.3: Subgraph computation example

The computation in figure 5.3 can be described as follows:

1. DFS is called on the start pixel and creates the tree root n1.
2. The DFS call is propagated up to the first branching point. n2 is created. DFS is called recursively on its north edge.
3. DFS finds no more unvisited neighbours and creates n3, returning it as a result for the call sequence initiated at n2. n2 adds n3 as an adjacent vertex.
4. n2's DFS call returned with n3 as a result. DFS is called on its east edge.
5. DFS finds multiple neighbours, creating a branching point at n4. DFS is called recursively on its north edge.
6. DFS discovers another branching point, creates n5 and calls DFS on its north edge.
7. DFS finds a dead end and creates n6. n5 adds n6 as an adjacent vertex. DFS is called on n5's south edge.
8. DFS finds a dead end (because the pixel area at n4 is already marked as visited), creates vertex n7 and returns it. n5's DFS call returns n7. n7 is added to the set of n5's adjacent vertices. n5 returns itself as a result.
9. The DFS call originating from n4 returns n5 as a result. n5 is added to n4's adjacency set. n4 tries to call DFS on its east edge, but discovers that the pixel located there was already visited. n4 is returned as a result.
10. The second DFS call originating from n2 returns n4, which is added to its adjacent vertices. n2 is returned.
11. The DFS result arrives at the root, returning n2. n2 is added as an adjacent vertex for n1. The function call terminates, returning the tree root n1 as a result.

The resulting tree is shown in figure 5.4. Table 5.1 lists each vertex' adjacent vertices at each step.

Note that, at the tile borders, vertices are automatically created since they will be needed to join tile graphs at a later step.

This step results in a set of weighted **trees** (because hiking trails need not necessarily be connected) containing branching points as vertices and hiking paths as edges.

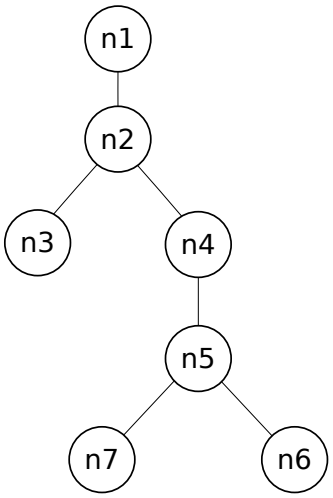


Figure 5.4: Tree resulting from the computation in figure 5.3

Step	n1	n2	n3	n4	n5	n6	n7
1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\{n3\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
5	$\emptyset$	$\{n3\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
6	$\emptyset$	$\{n3\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
7	$\emptyset$	$\{n3\}$	$\emptyset$	$\emptyset$	$\{n6\}$	$\emptyset$	$\emptyset$
8	$\emptyset$	$\{n3\}$	$\emptyset$	$\emptyset$	$\{n6\}$	$\emptyset$	$\emptyset$
9	$\emptyset$	$\{n3\}$	$\emptyset$	$\emptyset$	$\{n6, n7\}$	$\emptyset$	$\emptyset$
10	$\emptyset$	$\{n3\}$	$\emptyset$	$\{n5\}$	$\{n6, n7\}$	$\emptyset$	$\emptyset$
11	$\emptyset$	$\{n3, n4\}$	$\emptyset$	$\{n5\}$	$\{n6, n7\}$	$\emptyset$	$\emptyset$
end	$\{n2\}$	$\{n3, n4\}$	$\emptyset$	$\{n5\}$	$\{n6, n7\}$	$\emptyset$	$\emptyset$

Table 5.1: Adjacent vertex table for the computation in figure 5.3

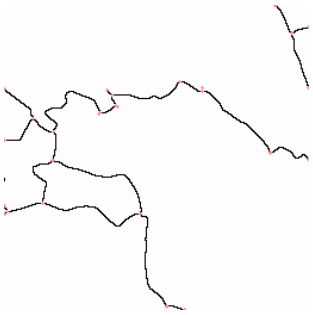


Figure 5.5: Hiking trails forming a cycle

Merging Internal Vertices

The result of the preceding step is a set of **trees**, i.e. a set of undirected acyclic connected graphs. Hiking trails however can indeed form cycles, as shown in figure 5.5. Therefore trees take too strong assumptions for trail graph representation.

To correct this, we observe what happened in the previous step when cycles exist: Because each vertex is marked upon visit, the algorithm concludes that there is a trail that ends just before the branching point (n4), as seen in figure 5.3 between step 7 and 8. It thus created two vertices (n4 and n7), lying just 1 pixel apart from each other.

Cycles are now introduced by merging all spatially close internal (i.e. non-border vertices) vertices and inserting edges accordingly.

The result of this step is a weighted, undirected graph (possibly unconnected and/or containing cycles). At this point, graphs for each tile are stored as GraphML files separately.

Merging Subgraphs and Coordinate Transform

As a result of the previous step, each map tile's graph is available. Routing computations can however only be performed on a single graph. It is thus necessary to compile the individual tile graphs (in the following referred to as "subgraphs") into a single large graph. That is, one needs to connect border vertices of a subgraph to the border vertices of its surrounding subgraphs.

Additionally, up to this point, all coordinates are essentially pixel coordinates on a per-tile-basis. For the resulting graph, WGS-84 coordinates are desirable.

While this processing step is conceptually similar to the previous one (merging of internal vertices), more problems arise:

- Memory: Loading all graphs into memory at the same time would hardly

scale. This problem is solved by per-tile processing.

- Quick vertex lookup: The algorithm presented in the following often checks whether a vertex already exists in the constructed graph. For performance reasons, such a lookup should take less than $\mathcal{O}(n)$ time. This is achieved by the use of a HashSet.
- Quick nearest neighbour search: The graph merging algorithm relies on a quick way to find spatially close points within a specified distance. This is achieved by using k-d trees.

The basic idea behind the merging algorithm is that if two subgraphs are to be connected, they both have vertices of degree 1 located at their tile's common border. The algorithm now tries to find and merge these vertices using k-d trees[6].

The algorithm's pseudocode is shown in algorithm 2.

Algorithm 2 Graph merging algorithm pseudocode

```
function MergeSubgraphs(subgraphs)
    g ← new Graph
    for all subgraph ∈ subgraphs do
        surrounding ← Get_Surrounding_Graphs_Vertices(subgraph)
            ▷ Get all vertices of the bordering graphs northwest, north and west of the
current subgraph
        surrounding ← surrounding ∩ g.vertices
        g ← g ∪ {subgraph}
        for all vertex ∈ subgraph.vertices do
            close_vertices ← Get_Close_Vertices(surrounding, vertex)
                ▷ Get all vertices in surrounding that are less than 3 pixels away from
vertex w.r.t. to the Manhattan distance (|| · ||_1)
            Merge_Vertices(g, close_vertices ∪ {vertex})
            surrounding ← surrounding \ close_vertices
        end for
    end for
    return g
end function
```

Subgraphs must be ordered from north to south and west to east in order for the above algorithm to work

For close vertex search, SciPy's cKDTree[5] (a k-d tree implemented in C) is used.

When all tiles are processed, the graph is complete. It is then stored in the GraphML format, a popular XML-based storage format for graphs, for later use in the routing service.

[5] http://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html

5.3 Performance

The Swiss hiking map consists of 29734 non-empty tiles of 256×256 pixels each at a scale of 1:50000, corresponding to 187 MB of compressed image data.

Processing is done in two steps:

- **Step 1** includes thresholding, thinning, subgraph construction and merging of internal vertices and is easily parallelizable since each tile can be processed independently of its surrounding tiles. Parallelization is implemented using Python's multiprocessing[6] library.
- **Step 2** consists of the subgraph merging and coordinate transformation. It is not easily parallelizable and is thus executed sequentially.

Computations were executed in the following environment:

OS	Ubuntu Linux 11.10 x86_64, kernel 3.0.0-14
CPU	Intel Core i5 2500K (4 cores, 3.3 GHz, 6.0 MB Cache)
RAM	12 GB at 1333 MHz
Python	CPython v2.7 (no Psyco or other optimizations)
	SciPy v0.9
	OpenCV v2.1
	PyGraph v1.8.0
	igraph v0.5.4
	mahotas v0.6.6

Step 1 took 1 hour and 16 minutes to complete using four parallel processes.

Step 2 took 5 hours and 31 minutes to complete using a single process.

5.4 Graph Information

File size	49.9 MB (uncompressed), 7.8 MB (compressed)
Graph type	undirected
Vertex count	267'133
Edge count	290'002
Total edge length	58'116.215 km
Average edge length	0.2 km
Number of components	717
Largest component	258'052 vertices (96.6%)

Note that, according to Wikipedia[7], there exist about 62'000 km of hiking trails in Switzerland (6.7% more than computed).

[6] docs.python.org/dev/library/multiprocessing.html
[7] http://de.wikipedia.org/wiki/Schweizer_Wanderwege

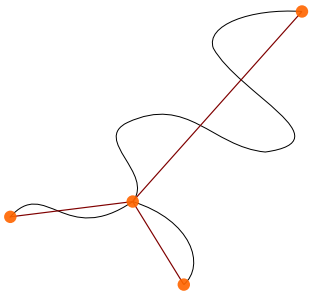


Figure 5.6: Twisted trails are not well approximated by straight lines

5.5 Possible Improvements

While the algorithm presented delivers good results and enables shortest-path computations, it can still be improved. Three possible starting points for improvements are described in the following sections.

5.5.1 Resolution

The currently computed graph contains only *branching points* as vertices. For route computations, branching points contain enough information. However, for navigation purposes this is not enough: Users want to see the complete path they need to walk. Thus if a trail between two branching points is not straight but rather twisted, drawing a straight line between those two points is not a good approximation for said trail. The problem is illustrated in figure 5.6 where orange circles are vertices created and dark red lines are edges in the graph.

In order to better approximate twisted trails, the graph's resolution has to be improved: Additional vertices need to be inserted. This immediately leads to the problem of where to place additional vertices without increasing the vertex count by too much[8].

Three possible approaches to solve this problem are:

- Constant distance resolution: New vertices are created when a fixed threshold for the distance to a vertex predecessor is exceeded.
- Twistedness coefficient: A new vertex is created when the currently followed trail's twistedness coefficient (see equation 7.1) exceeds a certain threshold.
- Area between line and trail: If the area between the real trail and its line approximation gets too large, a new vertex is created.

[8]More vertices lead to more memory consumption and longer running times for most algorithms, which is undesirable.

All described approaches help solving the approximation problem but also introduce different tradeoffs between efficiency, accuracy and algorithm complexity.

5.5.2 Distance Accuracy

The currently computed graph is based on the swiss hiking map 1:50000, i.e. 1 pixel in the map corresponds to 5 meters of real distance.

Recently SwissTopo has started publishing hiking maps at higher resolutions, e.g. 1:25000 and more. To improve accuracy in route computations, one could compute the graph at higher resolutions.

5.5.3 Performance

While the implementations already provide satisfying performance, especially since large parts are implemented in Python, an interpreted scripting language, performance might not be sufficient for computations at larger resolutions.

The most obvious staring points for optimizations are:

- Replace PyGraph by iGraph (already done to some extent): Computation of **step 1** relies on PyGraph[9], a pure Python graph library.

  While its interface is very user friendly, its performance suffers since its code needs to be interpreted. Replacing it with igraph[10], a graph library implemented in C already brought large performance gain for **step 2**. **Step 1** could therefore also profit from igraph in terms of performance.

- Parallelization: Parallelizing the subgraph merging process will speed up the graph computation.

- Depth First Search implementation in C: The DFS algorithm's implementation used for tree construction has a large contribution to the processing time of **step 1**. Implementing it in C will speed-up the **step 1** computation.

5.6 Route Comparison to Google Maps

For a quality assessment of the hiking graph, computed routes from the hiking graph are compared to pedestrian routes computed by Google Maps. Dijkstra's

[9] http://code.google.com/p/python-graph/
[10] http://igraph.sourceforge.net/

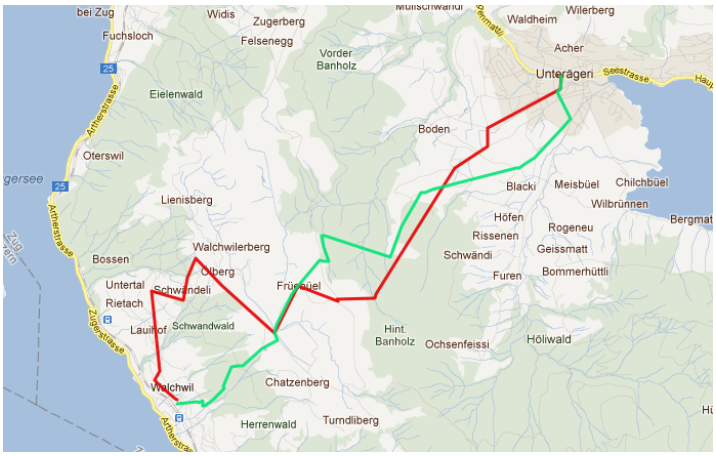


Figure 5.7: Visual route comparison for experiment #2 between Google Maps (red) and the Hiking Graph (green)

shortest-path algorithm is used for route computation on the hiking graph as explained in chapter 5.

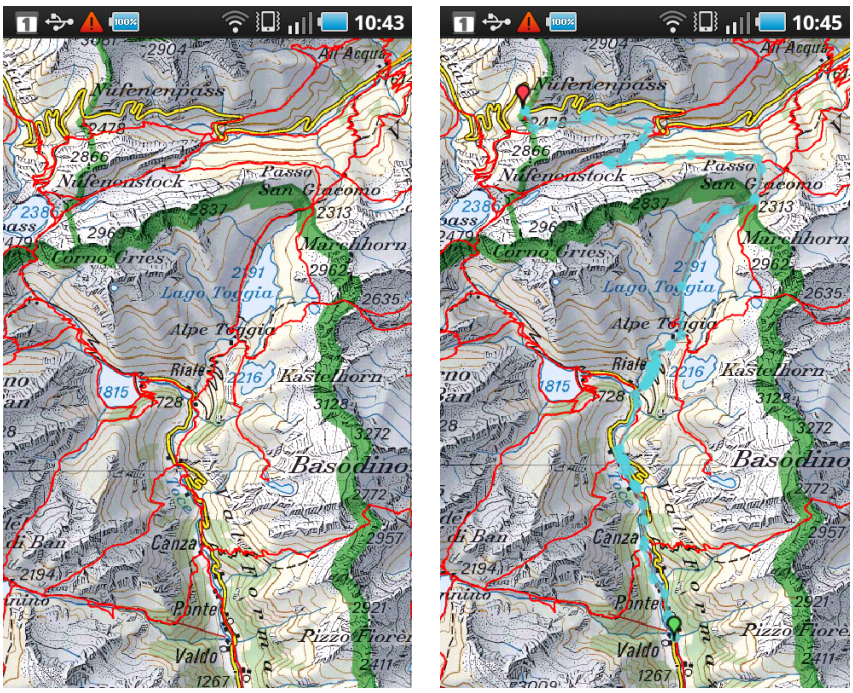
The comparison results are shown in table 5.2 and routes for experiment #2 and #4 are visualized in figure 5.7 and 5.8.

#	Start	Destination	Length (Google)	Length (Graph)	Diff
1	47.1986, 8.5290	47.2269, 8.4826	5.2 km	6.6 km	+27%
2	47.1379, 8.5804	47.1007, 8.5166	11.8 km	10 km	-15%
3	47.1393, 8.7559	47.2233, 7.7795	95.9 km	117.7 km	+22%
4	46.4775, 8.3866	46.3725, 8.4274	No result	24.3 km	—

Table 5.2: Route comparison to Google Maps

Note that the hiking graph only contains official Swiss hiking trails, whereas Google Maps' routes contain regular streets as well.

As expected, the hiking graph routes are longer in urban areas (#1, #3) when compared to Google Maps routes. In mountainous settings (#2, #4) the hiking graph routes can outperform Google Maps (#2, see figure 5.7) and even manage to find routes where Google Maps is unable to do so (#4, see figure 5.8(b)).



(a) Map section for experiment #4 with red colored mountain trails

(b) Graph route for experiment #4

Figure 5.8: Hiking Graph route for experiment #4

Chapter 6

Routing Service

6.1 Interface

The routing service is a RESTful webservice similar to Google Directions that, given a starting latitude/longitude pair and an destination latitude/longitude pair (passed as GET-parameters via the URL), returns a sequence of latitude/-longitude pairs and their distances between each other in a JSON representation.

6.2 Nearest Neighbour Search

A query's starting point or destination point will most likely not directly coincide with one of the graph's vertices. It is however necessary to choose start and destination vertices that are contained in the graph, otherwise shortest path computation can not be performed. Suitable vertex choices for start and end vertices are those located closest to the starting point / ending point, with the notion of distance being measured by some norm, e.g. the euclidean norm $||\cdot||_2$.

Let a point be given as a vector of latitude and longitude in $\mathbb{R}^2$, i.e.

$$p = \begin{pmatrix} \lambda \\ \phi \end{pmatrix} \tag{6.1}$$

The problem that needs to be solved can now be described as follows:

Definition 6.1 (2D Nearest Neighbour Problem) *Given a point in its vector representation p, a graph $G = (V, E)$, and a coordinate mapping $c : V \to \mathbb{R}^2$, find a vertex $v \in V$ such that $||p - c(v)||_2$ is minimal.* ◇

A naive approach would simply test all vertices in the graph and choose the one with minimal distance. This approach has time complexity $\mathcal{O}(|V|)$ and is thus sub-optimal.

The routing service uses a more sophisticated approach using a two-dimensional k-d tree based on SciPy's cKDTree[1] that can solve this problem in $\mathcal{O}(\log |V|)$ time at the cost of $\mathcal{O}(|V|)$ additional space.

6.3 Shortest Path Computation

The problem of finding a shortest path in the graph can now be defined as follows:

Definition 6.2 (Single-Pair Shortest Path for Undirected Graphs) *Given an undirected graph $G = (V, E)$, two vertices $v_1, v_2 \in V$ and an edge weighting function $w : E \to \mathbb{R}$, find a path P (i.e. a sequence of edges) from v_1 to v_2 such that $\sum_{e \in P} w(p)$ is minimal over all paths connecting v_1 and v_2.* ◇

This problem is solved by the well-known shortest path algorithm developed by Edsger W. Dijkstra[7].

The routing service uses Dijkstra's algorithm implementation from igraph[2].

6.4 Web Application

To provide a RESTful interface to possible users of the routing service (especially HikeDroid users) independent of implementation language, routing functionality is exported as a web service.

The routing service is deployed as a FastCGI web application using flup[3] and can be served by a multitude of popular webservers, including Apache[4], lighttpd[5], nginx[6], IIS[7], etc.

6.5 Performance

Routing performance is a critical for usability. After all, nobody enjoys waiting half a minute or even a minute to compute a hiking route.

[1] http://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html
[2] http://igraph.sourceforge.net/
[3] http://trac.saddi.com/flup
[4] http://httpd.apache.org/
[5] http://www.lighttpd.net/
[6] http://www.nginx.org/
[7] http://www.iis.net/

Some performance measurements for routes of varying length are listed in table 6.1. The runtime environment is identical to the one listed in section 5.3. Timing measurements are repeated (best of three).

Start	Destination	Edges	Distance	Time
47.1986, 8.5290	47.2269, 8.4826	24	6.6 km	7.44 ms
47.1379, 8.5804	47.1007, 8.5166	41	10 km	7.78 ms
46.4775, 8.3866	46.3725, 8.4274	81	24.3 km	8.24 ms
47.1393, 8.7559	47.2233, 7.7795	361	117.7 km	53.9 ms
46.5249, 6.6403	47.4467, 9.4089	965	350 km	123 ms

Table 6.1: Performance measurements for shortest path routing

The results show that shortest path performance depends heavily on the distance between two nodes and that shortest path computations are reasonably fast even for unrealistic hiking distances.

Chapter 7

Future Work

7.1 Additional Uses of Elevation Data

7.1.1 Thunderstorm Warning

Newer Android devices such as the Galaxy Nexus[1] include a barometer. Using the air pressure measured by the barometer and the altitude computed from the elevation model, one could detect abnormally low air pressure and (possibly considering more information sources) warn users about incoming thunderstorms[8].

7.1.2 Calculate Fitness Requirements for a Trail

Height differences on a trail make a major contribution to the amount of energy needed to walk on it, or in other words, how exhaustive it is. Based on a given trail (e.g. calculated by the routing service), its length and its height differences, one could rate its fitness requirements and display it to the user, possibly letting it select more or less exhaustive trails by explicitly excluding or including steep subtrails (or edges, in graph theory terms).

7.1.3 Snow Warning / Displaying a Snow Line

Using the snow line altitude (e.g. obtained from a web service) and a trail (e.g. obtained from the routing service), one can compute whether said trail is (partly) above the snow line. One could then warn the user or display the snow line in the trail's elevation profile plot.

[1] http://www.google.com/nexus/

7.2 Different Routing Metrics

Trail distance is not the only possible routing criterion. Often hikers are willing to accept a detour from the shortest trail if certain criteria are met. In the following two sections, two examples of such criteria are provided.

7.2.1 Least Exhausting Route

One could take into account the fitness requirements (as mentioned in section 7.1.2) for routing purposes. That is, instead of directly assigning distances as edge weights in the hiking graph, one could develop a model that combines the steepness of edges with its length, assigning them a new combined (positive[2]) weight value.

7.2.2 Most Twisted Route

Mountainbikers often enjoy riding serpentine trails particularly, since they impose special challenges.

One could then again develop a model taking into account both the actual walking distance and the "twistedness" of a subtrail such that serpentine subtrails result in smaller weights than straight subtrails of the same length.

A possible metric for the "twistedness" of a subtrail might be

$$t = \frac{d_{actual}}{d_{air\_line}} \tag{7.1}$$

where t is the "twistedness"-coefficient, d_{actual} is the actual distance and d_{air_line} is the air-line distance between the subtrails endpoints. Straight edges will result in a low t, where serpentines will result in a high t.

7.3 Resource Constrained Shortest Path (RCSP) Routing

The Resource Constrained Shortest Path (RCSP) problem is a restriction of the general Shortest Path problem (as solved by Dijkstra's algorithm for example).

Consider an augmented graph with both weights and resource requirements attached to each edge (e.g. weight might be its distance where a resource might be the amount of energy it takes to walk said edge).

RCSP is the problem of finding the path with minimal distance between two nodes while obeying resource constraints (e.g. it might use at most 800 kJ of

[2]Positive edge weights are required for shortest path computation using Dijkstra's algorithm

energy) or determining infeasibility. An example instance of this problem will be given in section 7.3.1.

RCSP is NP-Complete as shown by Handler and Zang[9]. However, in the case of hiking graphs this is not a serious limitation, since graphs are usually small. Additionally, heuristics and approximations exist that help solving larger problems[9].

Boost contains an implementation of an RCSP variant[3].

7.3.1 Most Interesting Round Trip Under Timing Constraints

When hiking, one might not want to find the shortest path between two locations, but often hikers do round-trips. Usually hikers have limited time and some subtrails are more interesting to walk (or beautiful) than others.

The problem of finding the most interesting round trip performable in limited time can be expressed as an instance of RCSP as follows:

- Time is a resource constraint.
- Edges are weighted by how interesting they are using negative values (ratings could be obtained online or directly specified by the user). The more interesting an edge (subtrail) is, the lower its weight.
- The resource requirement for an edge is the time it takes to walk said edge.
- The resource limit is the time the hiker can afford to spend (e.g. time until dusk).
- Both start and target node is the node where the hiker wants to start its round trip.

7.4 GPS Tagging and Trail Sharing

In the last few years, social networking platforms such as facebook and Google+ have become more and more popular. Along with their growth, user habits have changed: many people choose to share videos and photos or post status updates about current activities.

As hiking clearly is a leisure activity for the vast majority of hikers, users might want to share information about trails they walked or Points of Interest (POI) they encountered while hiking.

One possibility to enhance user experience in HikeDroid would thus be to give users the ability to mark POI on the map and share them with other hikers

[3] http://www.boost.org/doc/libs/1_41_0/libs/graph/doc/r_c_shortest_paths.html

visiting the same region. Since users can already record their trails, one could add a feature for trail rating and online sharing. Last but not least, existing POI and trail sources could be integrated, such as Hikr[4] or Gipfelbuch[5].

7.5 Integration of Other Map Sources / Offline Mapping

Using raster graphics for the map display works well, but also introduces severe limitations:

- Even with a caching mechanism in place, complete offline mapping is not possible. Storing the complete pixel map directly on the device is currently not feasible[6]. In a vector representation, the complete map could be stored on the device, as shown in MapDroyd[7] for example[8].
- Raster graphics require interpolation to scale, leading to undesirable visual artifacts. Vector maps can be scaled continuously.
- More interesting applications such as routing require vector or graph representations of a map.

It would thus be useful to integrate a vector based map representation into HikeDroid. Free data is available from several sources with Open Street Map[9] being the most popular (and probably most complete) source.

[4] http://www.hikr.org/

[5] http://www.gipfelbuch.ch/

[6] At a map scale of 1:50000 (0.2 px per meter), the complete map of Switzerland requires about 10 GB to store. Note that the application makes use of several map scales.

[7] https://market.android.com/details?id=com.osa.android.mapdroyd

[8] MapDroid requires less than 50 MB to store a detailed map of Switzerland

[9] http://www.openstreetmap.org/

Bibliography

[1] Pfammatter, D.: Hikedroid — gps navigation for hikers on android phones. Bachelor's thesis, ETH Zürich (2011)

[2] Guttman, A.: R-trees: a dynamic index structure for spatial searching. SIGMOD Rec. **14** (June 1984) 47–57

[3] Pei, J.: R-trees. Lecture Slides for Database Systems II, SFU Canada

[4] Wing, M.G., Eklund, A., Kellogg, L.D.: Consumer-grade global positioning system (gps) accuracy and reliability. Journal of Forestry **103**(4) (2005) 169–173

[5] Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **8** (June 1986) 679–698

[6] Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18** (September 1975) 509–517

[7] Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271

[8] Schlatter, T.: Weather queries: Rapid pressure changes near thunderstorms, directional lighting. Weatherwise **40**(2) (1987) 99–100

[9] Handler, G.Y., Zang, I.: A dual algorithm for the constrained shortest path problem. Networks **10**(4) (1980) 293–309

List of Figures