Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3457

Onboard Routenplanung auf dem Smartphone

Stefan Bühler

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Stefan Funke
Betreuer/in:	Dipl.-Inf. Daniel Bahrdt
Beginn am:	2013-02-08
Beendet am:	2013-08-10
CR-Nummer:	H.3.2, G.2.2, H.2.8

Inhaltsverzeichnis

1 Einleitung

Eine Welt ohne Routenplaner scheint heutzutage unvorstellbar. Sie haben den Atlas größtenteils verdrängt, der meist nur noch als absolute Notlösung ein Schattendasein fristet.

Die Navigationsgeräte, die z. B. in Autos verbaut werden, haben dabei Karten für bestimmte Bereiche (z. B. Deutschland, Europa, usw.) in ihrem lokalem Speicher. Weitere Bereiche und Aktualisierungen werden normalerweise von den Herstellern angeboten, die sich ihre Daten aber gut bezahlen lassen. So sind häufig die Kartendaten eines Navigationsgeräts ein paar Jahre veraltet, da die Anwender sich das Geld für die Aktualisierungen gerne sparen.

Als Alternative gibt es im Internet verfügbare Routenplaner (wie zum Beispiel `https://maps.google.com` oder `http://www.bing.com/maps/`). Die meisten Smartphones und Tablets werden mit bereits vorinstallierten Routenplaneranwendungen ausgeliefert, die auf solche Dienste zurückgreifen und dazu bei der Benutzung eine Internetverbindung voraussetzen. Dafür sind die Kartendaten meistens aktuell, und die Verwendung des Onlinedienstes an sich ist kostenlos.

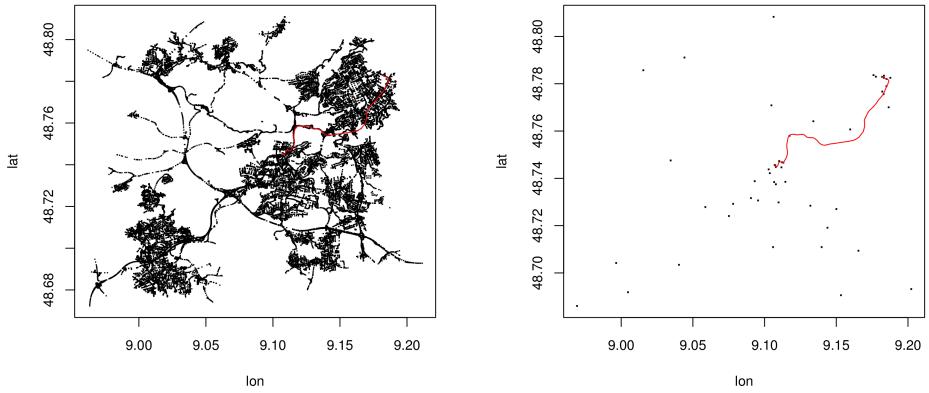
Allerdings steht aus Kostengründen (für den Internettarif auf dem Gerät ganz allgemein oder speziell auch im Ausland), aus Datenschutzgründen (der verwendete Onlinedienst kann das Verhalten einzelner Benutzer verfolgen) oder aus dem Grund, dass die Mobilfunknetzabdeckung den aktuellen Ort nicht oder nur mit geringer Bandbreite erfasst oder dass dem verwendeten Gerät die Hardware dazu fehlt (z. B. Tablets, die nur einen WLAN-Adapter haben), nicht immer eine ausreichende Internetverbindung zur Verfügung.

Die Leistungsfähigkeit der mobilen Geräte ist jedoch gut genug, um Routenplaner komplett offline zu betreiben (abgesehen von der Installation und Aktualisierungen). Mit mapsforge [MFOa] existiert bereits ein Renderer, und mit osmfind [Bah12][OSMb] steht eine fortschrittliche „Point-of-Interest"-Suche zur Verfügung.

In dieser Arbeit wird nun eine Implementierung für die Berechnung von kürzesten Wegen vorgestellt, die auch auf Smartphones und Tablets auf großen Wegenetzen (z. B. das Straßennetz von Deutschland) effizient arbeitet. In Kombination mit mapsforge und osmfind entsteht damit eine Androidanwendung, die auch ohne Internetverbindung einen Routenplaner anbietet.

Eine einfache Methode zum Berechnen von kürzesten Wegen in Graphen ist der Dijkstra-Algorithmus [Dij59]. Er muss dazu jedoch den kompletten Teilgraph um den Startknoten absuchen, der sich im Suchradius befindet (siehe Abbildung 1.1); im Falle von Straßengraphen sind dies ungefähr quadratisch viele Knoten in Relation zum Abstand von Start- und Zielknoten. Auf Smartphones ist dies nicht nur ein Problem für die CPU, die die Daten

verarbeiten muss, sondern bereits das Lesen des Graphen von einem Speichermedium benötigt viel Zeit.



(a) Dijkstra **(b)** contraction hierarchy

Abbildung 1.1: Suchräume eines normalen Dijkstra und einer CH

Die Verwendung von „Contraction-Hierarchies“ (CHs) [GSSD08] ist deutlich effizienter. Die Größe des zur Berechnung eines kürzesten Weges nötigen Teilgraphs, auf den dann der einfache Dijkstra-Algorithmus angewendet werden kann, hängt dabei nicht vom Abstand des Start- und Zielknoten ab. Der entsprechende Teilgraph dazu kann relativ schnell komplett eingelesen werden, anstatt Knoten erst dann zu laden, wenn sie benötigt werden.

Die vorgestellte Implementierung verwendet eine solche CH, um auf Androidsystemen einen „Offline ToureNPlaner“ (siehe Abbildung 1.2) zu implementieren. Die Implementierung wurde auf Grundlage von OpenStreetMap-Daten [OSMa] programmiert, ist aber nicht an diese gebunden. Alle Beispiele und Messwerte in diesem Dokument beziehen sich auf OpenStreetMap-Daten für Deutschland von Anfang 2013. Der Straßengraph enthält nur von Autos offiziell befahrbare Straßen.

1.1 Überblick über die entwickelten Komponenten

Die vorgestellte Implementierung knüpft an verschiedene bereits existierende Projekte an. Teilweise wurden diese im Rahmen der Arbeit erweitert. Im Wesentlichen besteht die Implementierung aus zwei Anwendungen: der Androidanwendung „Offline ToureNPlaner“ und der Hilfsanwendung „CHConstructor“, die die benötigten Daten für die Androidanwendung aufbereitet.

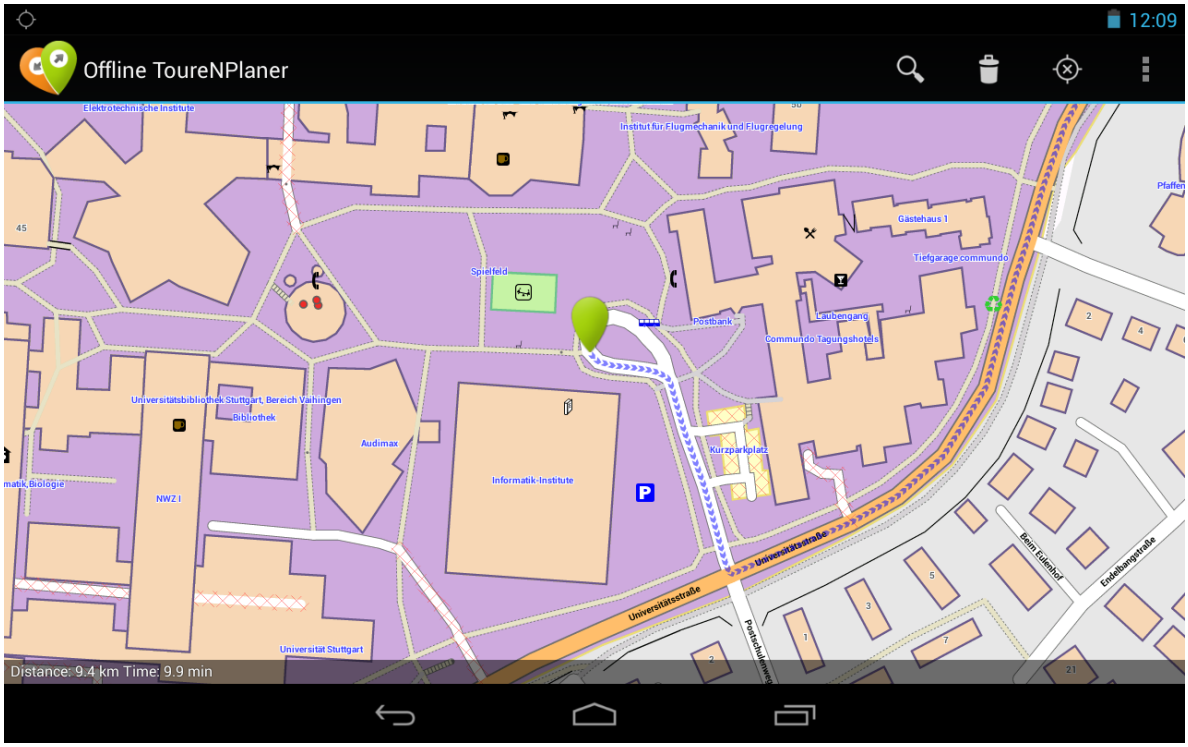


Abbildung 1.2: „Offline ToureNPlaner“ Androidanwendung

Offline ToureNPlaner

Die Androidanwendung verwendet folgende Komponenten:

- Ein modifiziertes mapsforge Plugin [MFOb]. Das Plugin wird zum Rendern der Karte verwendet. Die benötigten Kartendaten können von `http://download.mapsforge.org/` geladen werden.
- Für die Suche nach Koordinaten ausgehend von Ortsbeschreibungen wird das Javaplugin von osmfind[OSMb] verwendet. osmfind verwendet das „Java Native Interface“ (JNI) [JNI], da die eigentliche Implementierung in C++ programmiert wurde. Das Plugin muss für jede Zielarchitektur extra kompiliert werden.
- Für die optionale Kompression der Straßendaten wurde xz-jni [XZJ] entworfen. Auch xz-jni verwendet JNI, umd muss für jede Architektur extra kompiliert werden. xz-jni unterstützt sowohl xz [LZM] und eine eigene Variante von DEFLATE [DEF] mit Indextabelle.
- Das Laden der benötigten Teilgraphen auf Basis der CH-Daten, die mit Hilfe des CHConstructors exportiert wurden, und die Suchen nach kürzesten Wegen darin wurde direkt in der Anwendung implementiert.

CHConstructor

CHConstructor [CHCa] ist ein Projekt der Abteilung Algorithmik des Institut für Formale Methoden der Informatik an der Universität Stuttgart, das für diese Arbeit unter [CHCb] um ein neues Exportformat erweitert wurde. CHConstructor wird z. B. auch für den Onlineservice `http://tourenplaner.informatik.uni-stuttgart.de` verwendet, der von der ToureNPlaner-Androidanwendung [TOU] verwendet wird.

Für das Erstellen der CH benötigt der CHConstructor einige Minuten und rund 20GB Arbeitsspeicher auf einem Intel Core i7-3770 mit 3.40 GHz.

1.2 Gliederung

Die Arbeit ist in folgende Kapitel aufgeteilt: in Kapitel 2 werden die grundlegenden Algorithmen für die Kürzeste-Wege-Suche vorgestellt. Kapitel 3 beschreibt die Struktur der Daten auf der SD-Karte; diese ist die Grundlage dafür, dass die Algorithmen aus Kapitel 4 die Daten effizient verarbeiten können. In Kapitel 5 wird dann die fertige Androidanwendung präsentiert.

2 Präliminarien

In diesem Kapitel werden die grundlegenden Algorithmen und Strukturen für die Suchen nach kürzesten Wegen vorgestellt, die in der Implementierung verwendet werden.

2.1 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus ist das Standardverfahren für die Kürzeste-Wege-Suche, und wird auch für CHs benötigt.

Zunächst benötigen wir einige Definitionen für Graphen als Grundlage:

2.1.1 Definitionen

Definition 2.1. *Sei $G = (V, E, \delta), E \subseteq V \times V, \delta : E \to \mathbb{R}$, dann heißt G gerichteter, kantengewichteter Graph mit Knoten V und durch δ gewichtete Kanten E.*

In dieser Arbeit werden nur endliche Graphen betrachtet, d. h. V endlich und damit zwangsläufig auch E endlich. Zudem sind die Kantengewichte nie negativ, d. h. $\forall e \in E : \delta(e) \geq 0$.

Definition 2.2. *In einem Graph $G = (V, E, \delta)$ heißt $p = v_1 \to v_2 \to \ldots \to v_n \in V^+$ Weg von v_1 nach v_n, wenn $\forall i \in \{1, 2, \ldots, n-1\} : (v_i, v_{i+1}) \in E$. Die Weglänge $l(p) := \sum_{i=1}^{n-1} \delta(v_i, v_{i+1})$ ist die Summe aller durchlaufener Kantengewichte.*

Ein kürzester Weg von a nach b hat minimale Weglänge; existiert mindestens ein Weg von a nach b, so existiert auch ein kürzester Weg (es gibt nur endlich viele Wege ohne Zyklen $\ldots xv_1 \ldots v_n x \ldots$, und Zyklen können immer entfernt werden ohne den Weg zu verlängern).

$$d(a,b) := \begin{cases} l(p) & \exists \text{kürzester Weg } p \text{ von } a \text{ nach } b \\ \infty & \text{wenn kein solches } p \text{ existiert} \end{cases}$$

Definition 2.3. *Existiert für alle Knoten a und b ein Weg von a nach b, so heißt G stark zusammenhängend.*

Algorithmus 2.1 Dijkstra-Algorithmus

```
1: function findShortestPath(a, b)
2:     p ← new PriorityQueue()
3:     C ← {a}
4:     ▷ dist(v): Länge des bisher kürzesten gefundenen Weges von a nach v
5:     ▷ Wenn v ∈ C dann ist d(a, v) = dist(v) die Länge des kürzesten Weges von a nach v
6:     ▷ prev(v): Vorgänger von v auf dem bisher kürzesten gefundenen Weg von a nach v
7:     dist : V ↛ ℝ, prev : V ↛ V
8:     ∀v : dist(v) ← ⊥, prev(v) ← ⊥
9:     dist(a) ← 0
10:    foreach n in {n|(a, n) ∈ E} do
11:        if n ∉ C then
12:            dist(n) ← δ(a, n), prev(n) ← a
13:            Insert(p, Element(key ← dist(n), value ← n))
14:        end if
15:    end
16:    while b ∉ C do
17:        if IsEmpty(p) then
18:            return                          ▷ Es existiert kein Weg von a nach b
19:        end if
20:        v ← DeleteMin(p)
21:        C ← C ∪ {v}
22:        foreach n in {n|(v, n) ∈ E} do
23:            if n ∉ C ∧ (dist(n) = ⊥ ∨ dist(n) > dist(v) + δ(v, n) then
24:                d ← dist(n)
25:                dist(n) ← dist(v) + δ(v, n), prev(n) ← v
26:                if d = ⊥ then
27:                    Insert(p, Element(key ← dist(n), value ← n))
28:                else
29:                    DecreaseKey(p, Element(key ← d, value ← n), dist(n))
30:                end if
31:            end if
32:        end
33:    end while
34:    ▷ Rekonstruktion des Weges
35:    p ← b : V⁺, v ← b
36:    while v ≠ a do
37:        v ← prev(v)
38:        p ← (v → p)
39:    end while
40:    return (p, dist(b))                     ▷ gefundener kürzester Weg und dessen Länge
41: end function
```

2.1.2 Algorithmus

Um den kürzesten Weg von a nach b zu suchen, baut der Dijkstra-Algorithmus (siehe Algorithmus 2.1) eine Menge C von Knoten v auf, für die $d(a, v)$ bereits bekannt ist. Am Anfang ist $C = \{a\}$, da $d(a, a) = 0$. Dann sucht er wiederholt einen Knoten $v \notin C$ mit $\exists c \in C : (c, v) \in E$ und $d(a, c) + \delta(c, v)$ minimal. Für einen solchen Knoten gilt $d(a, v) = d(a, c) + \delta(c, v)$ (andere Wege nach v müssten über andere Knoten außerhalb von C gehen, die nicht kürzer sein können), darum kann er der Menge C hinzugefügt werden.

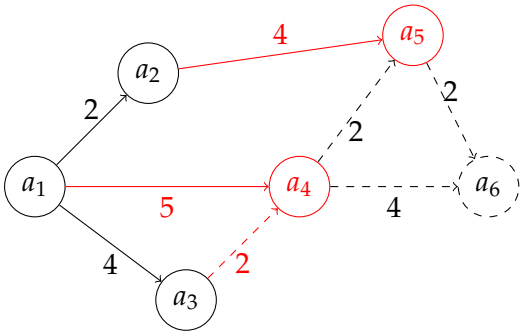
Der Algorithmus bricht ab, wenn er b in C aufgenommen hat oder keine weiteren Knoten erreicht werden können.

Um den kürzesten Weg auszugeben, kann bei der Aufnahme von v in C zu v der verwendete Knoten c gespeichert werden. Damit kann nach erfolgreicher Suche von b rückwärts der kürzeste Weg rekonstruiert werden.

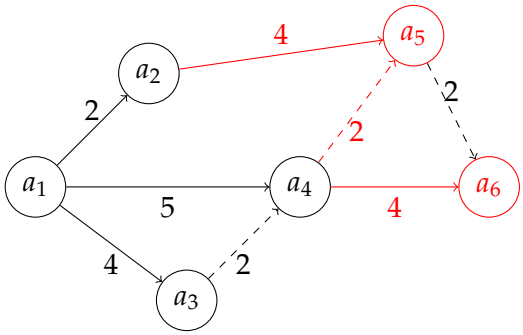
Die von C erreichbaren Knoten, die nicht in C liegen, werden üblicherweise in einer Priority-Queue gespeichert, um den Knoten v schnell zu finden. Die Priority-Queue kann eine besondere Monotonieeigenschaft verwenden: es werden nur Schlüssel einfügt, die mindestens so groß wie das zuletzt entnommene Element sind. In Abschnitt 2.3 wird dazu die Radix-Heap Struktur vorgestellt, die eine solche Priority-Queue implementiert, wenn die Kantengewichte alle ganzzahlig sind, d. h. $\delta(V) \subseteq \mathbb{N}_{\geq 0}$.
Wenn ein Knoten v in C eingefügt wird, müssen alle Kanten von v aus betrachtet werden. Wenn dabei neue Wege oder neue bekannte kürzeste Wege zu Knoten gefunden werden, müssen die Einträge in der Priority-Queue entsprechend angepasst werden.

2.1.3 Beispiel

In folgendem Graph wird der kürzeste Weg von a_1 nach a_6 gesucht. Die Knoten a_1, a_2 und a_3 sind bereits in C aufgenommen. Die Priority-Queue enthält nun alle direkt von C erreichbaren Knoten, die nicht bereits in C sind; dies sind a_4 und a_5 mit den Schlüsseln 5 (der Weg von a_1 aus ist 5 lang, der Weg über a_3 ist länger) und 6 (a_5 ist von C nur über a_2 direkt erreichbar).

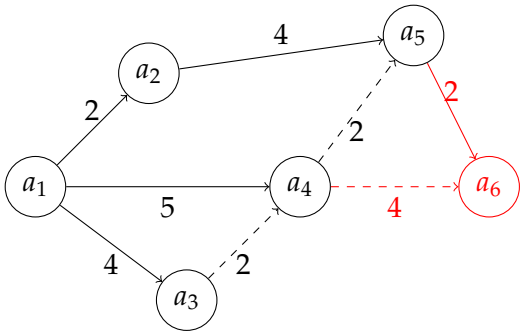


Das minimale Element in der Priority-Queue ist also a_4 mit Schlüssel 5, und wird in der nächsten Runde in C aufgenommen. Außerdem wird als Vorgänger von a_4 auf dem kürzesten Weg von a_1 nach a_4 der Knoten a_1 gespeichert, und $d(a_1, a_4) = 5$:

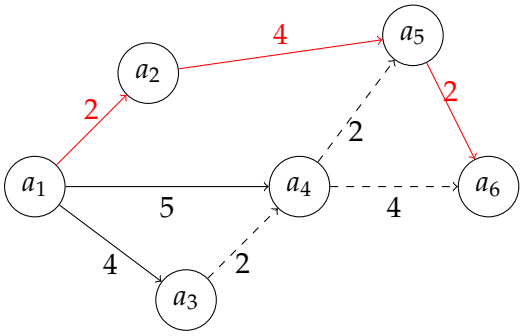


Dabei wird der Knoten a_6 mit Schlüssel 9 in die Priority-Queue aufgenommen, da er von $a_4 \in C$ erreichbar ist. a_5 ist nun auch über a_4 erreichbar, aber der neue Weg ist nicht kürzer wie der bereits bekannte.

Jetzt ist das minimale Element in der Priority-Queue der Knoten a_5 mit Schlüssel 6, und wird in C aufgenommen:



Der Knoten a_6 wird in der Priority-Queue nun verkleinert auf den Schlüssel 8, da über $a_5 \in C$ nun ein neuer kürzerer Weg bekannt ist. Nach der Aufnahme von a_6 als letzten Knoten in C ist der kürzeste Weg von a_1 nach a_6 fertig berechnet:



Der kürzeste Weg $a_1a_2a_5a_6$ von a_1 nach a_6 mit Länge 8.

2.1.4 Laufzeit

Alle von a erreichbaren Knoten v mit $d(a,v) < d(a,b)$ sind am Ende in C gespeichert. Bei Suchen auf einem Straßennetz sind das also alle Knoten innerhalb des Kreises um a mit Radius $d(a,b)$, also im Schnitt quadratisch viele Knoten in Bezug auf den Abstand $d(a,b)$.

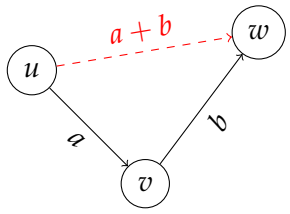
Insbesondere wenn zwischen a und b eine Stadt mit vielen kleinen Nebenstraßen auftaucht, durchsucht Dijkstra die komplette Stadt - es könnte ja sein, dass es mittendrin mit einem „Tunnel“ eine schnelle Direktverbindung zum Ziel gibt.

Mit einer normalen Priority-Queue liegt die Laufzeit in $O(|E| + |V| \cdot \log |V|)$, wobei nur die Knoten und Kanten gezählt werden müssen, die auch besucht werden, d. h. die am Ende von C aus erreichbaren Knoten und Kanten.

2.2 Contraction-Hierarchies

2.2.1 Erstellen einer Contraction-Hierarchy

Die Grundlage für das Erstellen einer Contraction-Hierarchy (CH) ist eine Ordnung auf den Knoten. Das Ziel ist, dass höhere Knoten in der Ordnung zentrale Knotenpunkte darstellen, über die viele kürzeste Wege laufen, während die niedrigen Knoten die Details des Graphen darstellen. Der Algorithmus fängt dann von unten in der Ordnung an, und kontrahiert die Knoten. Der nächste Knoten in der Ordnung wird dabei meistens erst dann festgelegt, wenn die vorigen bereits kontrahiert wurden, eine Heuristik wählt dabei aus welcher Knoten als nächstes kommt. Das Ziel einer Heuristik sollte sein, die Anzahl der Knoten und insbesondere Kanten in den Aufwärts- bzw. Abwärtsgraphen (siehe unten) klein zu halten. Für weitere Details einer solchen Heuristik sei auf [GSSD08] und die CHConstructor-Implementierung (siehe Abschnitt 1.1) verwiesen.



Für die Kontraktion eines Knotens v werden alle Wege $p = u \to v \to w$ betrachtet, für die $v < u, v < w$ gilt. Ist p ein kürzester Weg von u nach w, so wird ein neue Kante von u nach w mit dem Gewicht $l(p)$ eingefügt, falls die Kante nicht bereits existiert und gleich lang ist. Eine solche neue Kante heißt „Shortcut“.

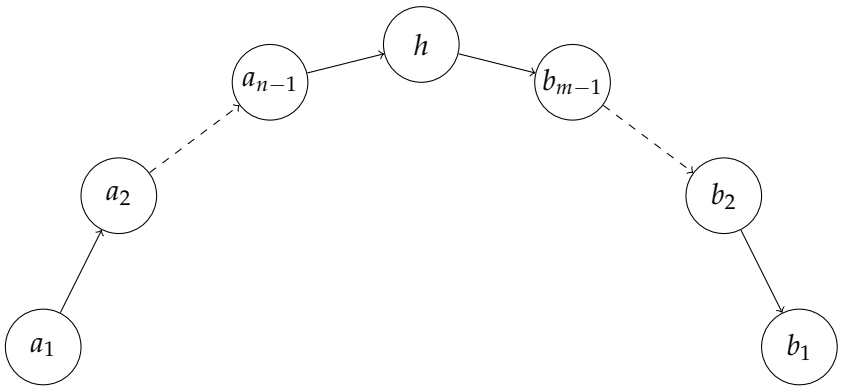
Der nach Kontraktion aller Knoten entstandene Graph ist dann die Contraction-Hierarchy.

Für Wege in der CH kann jeder verwendete Shortcut durch den zugrunde liegenden Weg p ersetzt werden (siehe auch Abschnitt 4.3), ohne die Länge des Weges zu verändern. Nach endlich vielen Schritten enthält ein Weg dann keine Shortcuts mehr, also haben die eingefügten Shortcuts keinen Einfluss auf die Länge der kürzesten Wege in dem Graph.

2.2.2 Verwendung

Wenn nun in einem kürzesten Weg p ein Teilweg $u \to v \to w$ mit $v < u < w$ oder $v < w < u$ auftaucht, so kann dieses Teilstück durch $u \to w$ ersetzt werden, denn $u \to v \to w$ ist auch ein kürzester Weg von u nach w, und nach Kontraktion von v muss es eine direkte Kante von u nach w geben mit Gewicht $l(u \to v \to w)$. Da durch eine solche Ersetzung der Weg um einen Knoten kürzer wird, sind nur endliche viele solcher Ersetzungen möglich.

Sei nun h der größte Knoten in dem Weg p (dieser kann nie bei einer Ersetzung verloren gehen), so kann p durch endlich viele Ersetzungen in einen Weg p_{ch} der Form $a_1 \to \ldots \to a_n = h = b_m \to \ldots \to b_1$ transformiert werden, wobei sowohl die a_i als auch die b_i monoton aufsteigend sortiert sind:



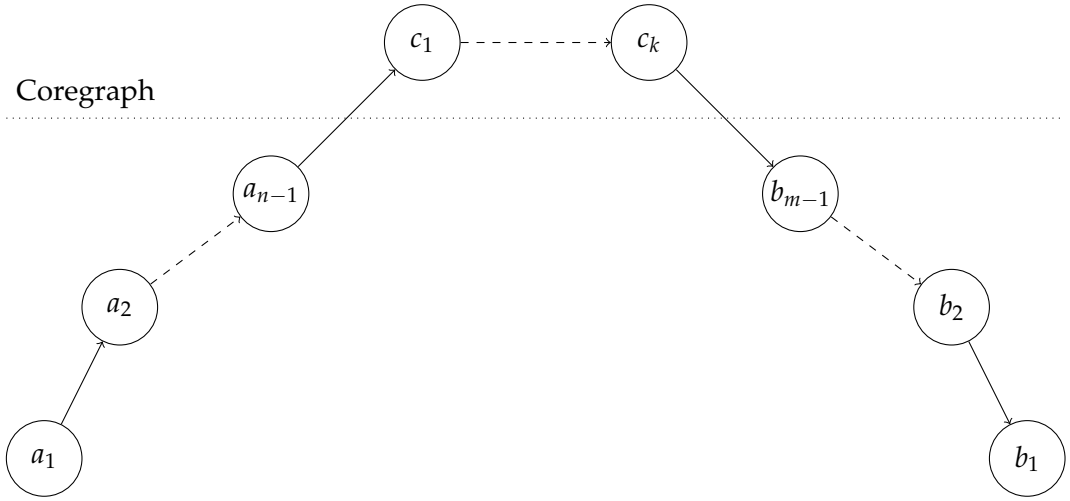
Definition 2.4.

- *Aufwärtskanten von v:* $E_{up}(v) := \{(v,w) \in E \mid v < w\}$
- *Abwärtskanten zu v:* $E_{down}(v) := \{(w,v) \in E \mid v < w\}$
- *Aufwärtsgraph von v:* $E^*_{up}(v) := E_{up}(v) \cup \bigcup_{(v,w) \in E_{up}(v)} E^*_{up}(w)$
- *Abwärtsgraph zu v:* $E^*_{down}(v) := E_{down}(v) \cup \bigcup_{(w,v) \in E_{down}(v)} E^*_{down}(w)$

Nun kann der transformierte Weg p_{ch} bereits in $E^*_{up}(a_1) \cup E^*_{down}(b_1)$ gefunden werden, d. h. für die Suche nach einem kürzesten Weg von a nach b genügt der Teilgraph $E^*_{up}(a) \cup E^*_{down}(b)$. Durch Ersetzung der Shortcuts kann danach wieder der dazugehörige kürzeste Weg im Originalgraph gefunden werden.

2.2.3 Coregraph

Die Aufwärts- bzw. Abwärtsgraphen für verschiedene Knoten enthalten einen sehr großen Teil aller höheren Knoten. Es bietet sich an, ab einem bestimmten Knoten c beim Erstellen der CH die Knoten nicht mehr zu kontrahieren [Sch13]. Die Knoten $v \geq c$ bilden dabei den Coregraphen C (mit den Kanten $\{(v,w) \in G \mid v,w \geq c\}$), ihre Ordnung untereinander spielt keine Rolle. Falls ein kürzester Weg p durch den Coregraphen geht, kann er zu einem Weg $p_{ch'}$ der Form $a_1 \to \ldots \to a_n = c_1 \to \ldots \to c_k \to b_m \to \ldots \to b_1$ transformiert werden, wobei a_i und b_i wieder aufsteigend sortiert sind und $c_i \geq$ Knoten im Coregraph sind.



Definition 2.5.

- *Aufwärtsgraph ohne Core von v:* $E'^*_{up}(v) := \begin{cases} \emptyset & \text{wenn } v \in C \\ E_{up}(v) \cup \bigcup_{(v,w) \in E_{up}(v)} E'^*_{up}(w) & \text{wenn } v \notin C \end{cases}$

- *Abwärtsgraph ohne Core zu v:* $E'^*_{down}(v) := \begin{cases} \emptyset & \text{wenn } v \in C \\ E_{down}(v) \cup \bigcup_{(w,v) \in E_{down}(v)} E'^*_{down}(w) & \text{wenn } v \notin C \end{cases}$

$p_{ch'}$ kann nun in $E'^*_{up}(a_1) \cup E'^*_{down}(b_1) \cup C$ gefunden werden; wenn p nicht durch den Coregraph geht, so funktioniert obige Transformation, und p_{ch} kann in $E'^*_{up}(a_1) \cup E'^*_{down}(b_1)$ gefunden werden. Für die Suche nach einem kürzesten Weg von a nach b genügt also der Teilgraph $E'^*_{up}(a) \cup E'^*_{down}(b) \cup C$.

Die Aufwärts- bzw. Abwärtsgraphen ohne Core sind dabei (je nach Wahl von c) deutlich kleiner, dafür wird der Coregraph natürlich auch größer. Das Einlesen des Coregraphen von langsamem Speicher ist dabei jedoch deutlich schneller (er kann im Gegensatz zu den Aufwärts- bzw. Abwärtsgraphen in einem Stück gelesen werden), und er kann für mehrere Suchen in schnellerem Speicher gehalten werden (solange er nicht zu groß wird).

2.3 Radix-Heaps

Eine Priority-Queue bietet normalerweise folgende Operationen:

- Insert: Fügt ein neues Element ein
- DecreaseKey: Verkleinert ein bereits vorhandenes Element
- DeleteMin: Entfernt das kleinste Element und gibt es zurück
- IsEmpty: Testet ob die Priority-Queue leer ist

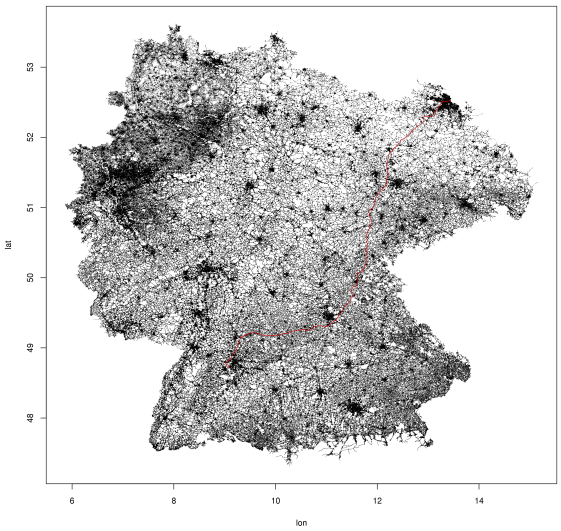
Für den Dijkstra-Algorithmus aus Abschnitt 2.1.2 besteht ein Element aus zwei Teilen:

- Einer positiven Ganzzahl als „Schlüssel“, nach dem sortiert wird
- Einem zugeordneten Wert für den Knoten

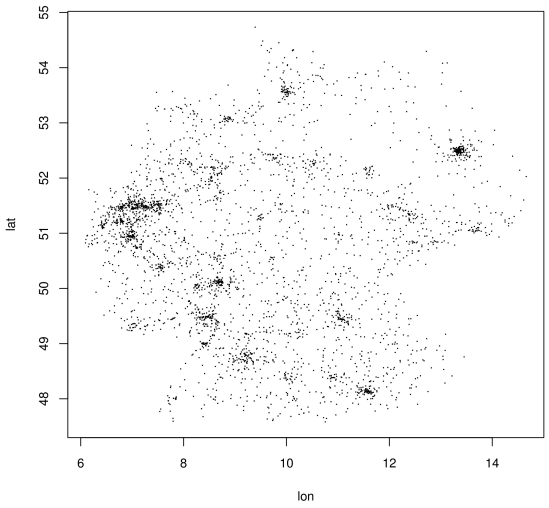
Im Radix-Heap[AMOT90] werden die Elemente in Buckets einsortiert; jeder Bucket bekommt dabei ein Intervall von Schlüsseln zugewiesen. Die Intervalle werden mit wachsendem Abstand zum zuletzt entfernten Element exponentiell größer und sind aufsteigend sortiert. Das minimale Element befindet sich also in dem ersten nicht leeren Bucket. Es kann kein Element eingefügt werden, das kleiner als das zuletzt entfernte Element ist; dieselbe Einschränkung gilt für die DecreaseKey-Operation: das verkleinerte Element darf nicht kleiner als das zuletzt entfernte Element sein.

Beim Entfernen des minimalen Elements müssen die anderen Elemente aus dem Bucket, in dem das minimale Element war, neu einsortiert werden. Die Intervallzuordnung muss aber so gewählt werden, dass die anderen Buckets weiter hinten unverändert bleiben. Damit bewegt sich jedes Element beim neu einsortieren weiter nach vorne in der Liste von Buckets.

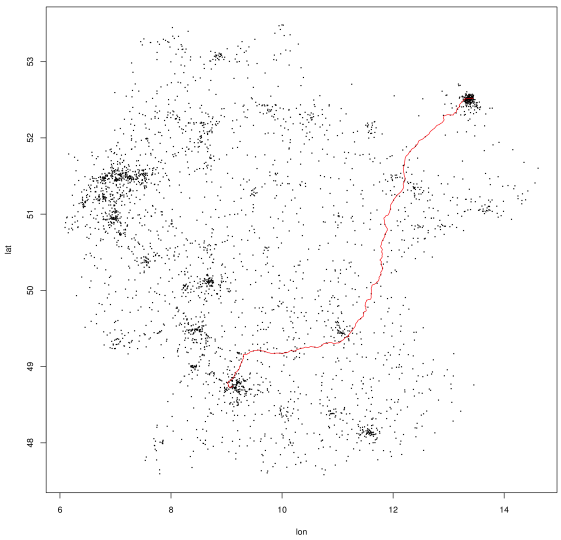
Algorithmus 2.2 (eine Variante nach [Sø]) verwendet für die Bucketzuordnung den Index des größten unterschiedlichen Bits (Zählung ab 1, 0 für keinen Unterschied) zum zuletzt



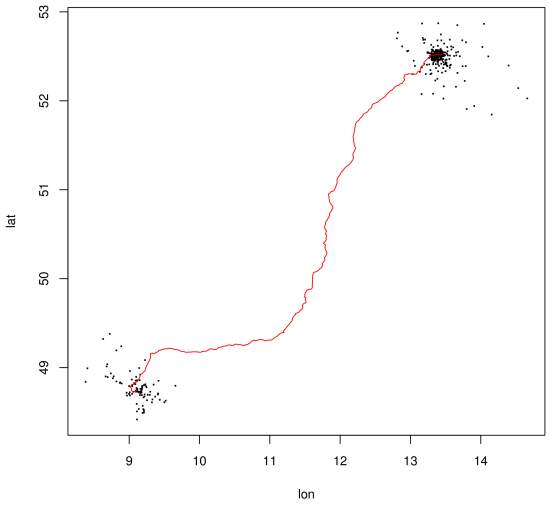
(a) Dijkstra-Suchraum



(b) Knoten im Coregraph



(c) Besuchte Knoten in der CH



(d) Knoten im Aufwärts- bzw Abwärtsgraph

Abbildung 2.1: Suchräume für kürzeste Wege-Suche von Stuttgart nach Berlin

entfernten Element. Alle Schlüssel im gleichen Bucket i haben also für Bits $\geq i$ die gleichen Werte; nämlich für Bits $> i$ die gleichen Werte wie im zuletzt entfernten Element und, für $i \geq 1$, das i. Bit negiert . Durch diese Methode gibt es je nach Bitmuster des zuletzt entfernten Elements einige Buckets, deren Intervalle vor dem zuletzt entferntem Element liegen; diese bleiben leer. Ohne diese nicht verwendbaren Intervalle sind die Intervalle dann wie gefordert aufsteigend sortiert.

Wenn nun das nächste kleinste Element entfernt wird, so liegt es in einem bestimmten Bucket i. Alle Buckets davor müssen leer sein, da es sonst nicht das kleinste Element ist. Die Intervalle der Buckets danach ändern sich nicht, da für $j > i$ das j-te Bit im entfernten Element den gleichen Wert hat wie im davor entfernten Element. Wenn das kleinste Element nicht in Bucket 0 lag, so werden alle Elemente aus dem Bucket i in Buckets davor eingeordnet, da nur Bits kleiner i sich von dem kleinsten Element unterscheiden können.

2.3.1 Beispiel

Wenn das zuletzt entfernte Element $10110_2 = 22$ ist, so ergibt sich folgende Zuordnung der Intervalle auf die Buckets:

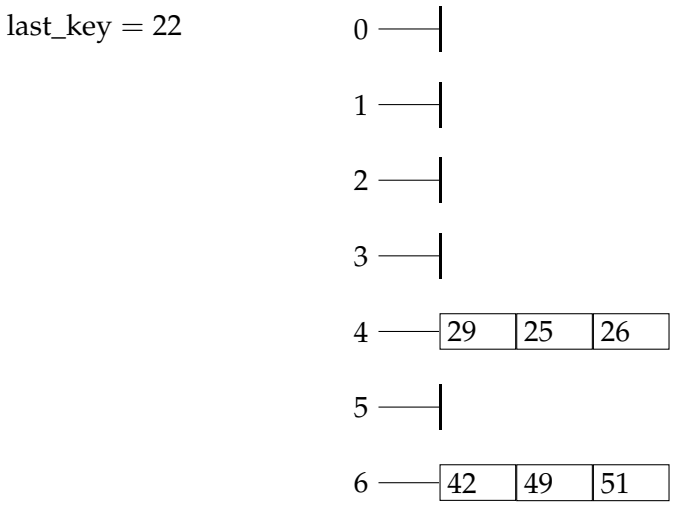
Index	Intervall	Intervallgröße
0	$10110_2(22)$	1
1	$10111_2(23)$	1
2	$10100_2 \ldots 10101_2(20 \ldots 21)$	2
3	$10000_2 \ldots 10011_2(16 \ldots 19)$	4
4	$11000_2 \ldots 11111_2(24 \ldots 31)$	8
5	$00000_2 \ldots 01111_2(0 \ldots 15)$	16
6	$100000_2 \ldots 111111_2(32 \ldots 63)$	32
...		
i	$2^(i-1) \ldots 2^i - 1$	$2^(i-1)$

Die Buckets 2, 3 und 5 (entsprechen den Positionen der „1“-Bits in 10110_2) werden nicht verwendet, da ihre Intervalle vor 22 liegen. Die verbleibenden Intervalle beginnen bei $22, 23, 24, 32, 64, \ldots$.

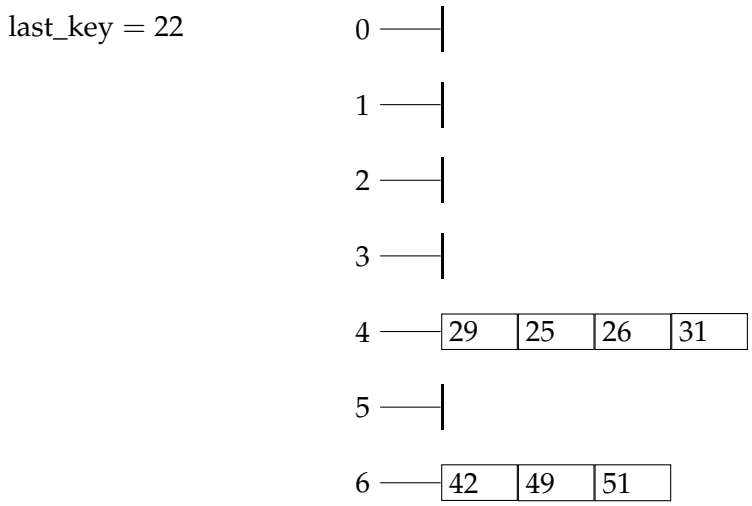
Damit könnte der Radix-Heap folgendermaßen aussehen:

Algorithmus 2.2 Radix-Heaps

```
1:  struct Element is
2:      key : unsignedinteger
3:      value : object
4:  end
5:  struct RadixHeap is
6:      last_key : unsigned integer ← 0
7:      ▷ Jeder „Bucket“ ist ein dynamisches Array von Elementen.
8:      ▷ Schlüssel in buckets[i] unterscheiden sich von last_key im (i − 1)-ten Bit
9:      ▷ und möglicherweise in kleineren Bits, und gar nicht für i = 0.
10:     ▷ Schlüssel innerhalb eines Buckets buckets[i] unterscheiden sich höchstens
11:     ▷ in den Bits 0 bis i − 2, und decken damit ein Intervall der Größe 2^max(0,i−1) ab
12:     buckets : Element[][]
13: end
14: function BucketIndex(heap, key)
15:     if heap.last_key = key then
16:         return 0
17:     end if
18:     return 1 + ⌊log_2(heap.last_key xor key)⌋
19: end function
20: procedure Insert(heap, element)
21:     bucket ← BucketIndex(heap, element.key)
22:     Append(buckets[bucket], element)
23: end procedure
24: procedure DecreaseKey(heap, oldelement, newkey)
25:     oldbucket ← BucketIndex(heap, oldkey)
26:     newbucket ← BucketIndex(heap, newkey)
27:     RemoveValue(buckets[oldbucket], oldelement)
28:     Insert(heap, Element(key ← newkey, value ← oldelement.value))
29: end procedure
30: function DeleteMin(heap)
31:     bucket ← min{i | Length(buckets[i]) > 0}
32:     if bucket ≠ 0 then
33:         heap.last_key ← min{e.key | e ∈ buckets[bucket]}
34:         foreach e in buckets[bucket] do
35:             ▷ Elemente landen in Buckets mit kleinerem Index
36:             ▷ Die kleinsten Elemente landen in Bucket 0, eins davon wird unten entfernt
37:             Insert(heap, e)
38:         end
39:         Clear(buckets[bucket])
40:     end if
41:     element ← RemoveLast(buckets[0])
42:     return element
43: end function
44: function IsEmpty(heap)
45:     return ⋀{Length(b) = 0 | b ∈ buckets}
46: end function
```



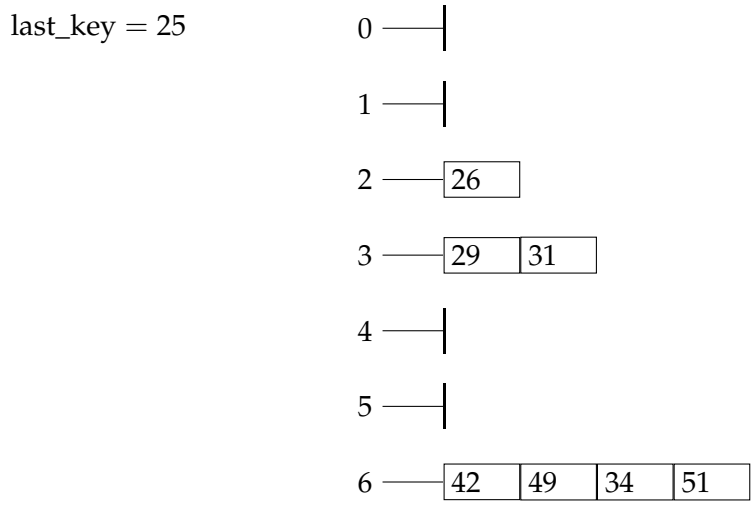
Wenn nun ein Element mit Schlüssel 31 eingefügt wird, wird es einfach im richtigen Bucket angehängt. Die Reihenfolge innerhalb eines Buckets spielt keine Rolle:



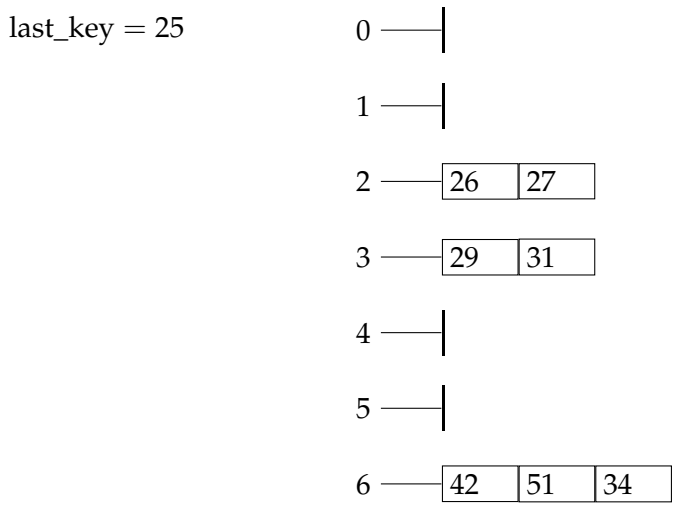
Das kleinste Element ist im ersten nicht leeren Bucket; in diesem Fall das Element 25 in Bucket 4. Beim Entfernen dieses Elements werden die anderen Elemente aus Bucket 4 auf die Buckets 0 bis 3 neu sortiert.

Zuerst die neuen Intervallgrenzen (ab Bucket 5 ändern sich diese nicht), danach der neue Radix-Heap.

Index	Intervall	Intervallgröße
0	$11001_2(25)$	1
1	$11000_2(24)$	1
2	$11010_2 \dots 11011_2(26 \dots 27)$	2
3	$11100_2 \dots 11111_2(28 \dots 31)$	4
4	$10000_2 \dots 10111_2(16 \dots 23)$	8



Für die DecreaseKey-Operation muss das alte Element im alten Bucket entfernt werden, und dann entsprechend dem neuen Schlüssel neu eingefügt werden. Wenn also das Element 49 auf 27 verkleinert wird, könnte folgender Radix-Heap entstehen:



Um die Anzahl der Speicherzugriffe zu verringern, kann in einem Bucket das zu löschende Element durch das letzte Element ersetzt werden, anstatt alle Folgeelemente um eins zu verschieben; die Reihenfolge der Elemente innerhalb eines Buckets spielt keine Rolle.

3 Organisation der Daten

In diesem Kapitel soll nun das Binärformat erläutert werden. Dazu wird erst ein Überblick über die Daten gegeben, die zu speichern sind. Dann wird die Anordnung der Daten beschrieben, für die ein spezielles Gitter verwendet wird, und danach die konkrete Binärrepräsentation. Letzendlich wird die Auswirkung einer optionalen Kompression der Daten untersucht.

3.1 Übersicht über die benötigten Daten

Zur Beschreibung des CH-Graphen werden folgende Eingabedaten für die Knoten und Kanten benötigt:

- Knoten
  - Geographische Lage (Längen- und Breitengrad)
  - Rang in der CH-Ordnung
- Kanten
  - Referenz auf den Startknoten
  - Referenz auf den Zielknoten
  - Kantengewicht für die kürzeste Wege-Suche (z. B. die Reisezeit)
  - Euklidische Länge des Weges (für Shortcuts: die Summe der abgekürzten Wege)
  - Für Shortcuts: Referenz auf die zwei durch diesen Shortcut abgekürzten Kanten

Die Knoten werden nun auf Blöcke verteilt (siehe nächster Abschnitt), jeder Knoten wird also durch Blocknummer und Index innerhalb eines Blockes identifiziert.

Die Kanten werden dem niedrigeren Knoten zugewiesen (haben beide Knoten einer Kante denselben Rang, so sind die Knoten im Coregraph; die Kante wird dann dem Startknoten zugewiesen). Jeder Knoten hat dann ihm zugewiesene ausgehende bzw. eingehende Kanten (Knoten im Coregraph werden keine eingehende Kanten zugewiesen). Diese Zuweisung erfolgt in Hinblick auf das Auslesen des Aufwärts- bzw. Abwärtsgraphen: für jeden Knoten sind nur die Kanten interessant, die in der CH nach oben gehen.

Knoten	Koord.	1. ausg. Kante	1. eing. Kante	Kante	Gewicht	Knoten
(B, 0)	(lon, lat)	$X+0$	$X+2$	$X+0$	4	(102, 4)
				$X+1$	10	(1943, 6)
				$X+2$	7	(374, 24)
(B, 1)	(lon, lat)	$X+3$	$X+7$	$X+3$	2	(102, 4)
...						
				$X+k$	40	(583, 42)
$\perp$	$\perp$	$X+k+1$	$\perp$	$[X+k+1]$	$\perp$	$\perp$

(a) Knoten mit Kantenzuordnung

Kante	Eukl. Abstand	SC: 1. Kante	SC: 2. Kante	SC: Knoten
$X+0$	100			
$X+1$	150	823	9781	(0, 12)
$X+2$	150			
$X+3$	50			
...				
$X+k$	400	413	132	(1, 39)

(b) Euklidische Länge und Shortcutdaten für Kanten

Abbildung 3.1: Beispiel: Knoten eines Blocks mit zugewiesenen Kanten

Nun werden die zugewiesenen Kanten für die Knoten eines Blockes hintereinander angeordnet. Für jeden Knoten kommen zuerst die ausgehenden und dann die eingehenden zugewiesenen Kanten.

Da beim Erstellen der Aufwärts- bzw. Abwärtsgraphen der niedrigere Knoten der Kante bereits bekannt ist (über diesen wird die Kante gefunden), genügt es, den jeweils anderen Knoten zu speichern. Bei der Ersetzung der Shortcutkanten fehlt dann aber der kontrahierte Knoten, darum muss dieser für Shortcutkanten gespeichert werden.

Für jeden Knoten muss nur der Index der ersten ausgehenden bzw. eingehenden Kante gespeichert werden, der Index der jeweils letzten Kante ergibt sich aus dem Startindex der nächsten Kantenliste. Am Ende eines Blocks wird dann zusätzlich ein abschließender Index für die Kantenliste des Blocks benötigt.

Jeder Knoten und jede Kante haben eine fixe Größe (Kanten, die keine Shortcuts sind, haben speziell markierte Einträge für die Shortcutdaten). Für jede Datei ist die Anzahl der Knoten pro Block fest, also haben auch alle Blöcke die gleiche Größe; für jeden Block wird gespeichert, wie viele Knoteneinträge tatsächlich verwendet werden. Darum kann direkt aus Blocknummer und Index innerhalb des Blocks bzw. Kantennummer die Position der Daten in der Datei direkt ohne Nachschlagen in einem Index berechnet werden.

3.2 Gitter

Um den Start- bzw. Zielknoten mit Hilfe der jeweiligen geographischen Lage zu finden, werden die Knoten in einem Gitter abgespeichert. Da höhere Knoten bzw. deren Kanten beim Aufbau der Aufwärts- bzw. Abwärtsgraphen öfters benötigt werden, sollten diese jedoch möglichst nahe beieinander liegen. Darum wird für verschiedene Rangintervalle ein eigenes Gitter verwendet, wobei die höheren Gitter gröber auflösen; jede Gitterzelle liegt dabei komplett in einer Gitterzelle in der Ebene darüber. Von oben betrachtet werden also Zellen auf dem Weg nach unten weiter unterteilt.

Wenn für eine Gitterzelle Knoten auf einer höheren Ebene im gleichen Raster liegen, so verweist die Gitterzelle auf die niedrigste Gitterzelle darüber, in der ein solcher Knoten enthalten ist. So können von einer Gitterzelle auf der untersten Ebene aus alle Knoten gefunden werden, deren geographische Lage sich darin befindet (auf höheren Ebenen sind auch Knoten dabei, die in der untersten Ebene in Zellen daneben liegen).

Eine Gitterzelle wird dann in mehrere Blöcke zerlegt, wobei auch die einzelnen Blöcke wieder einfach verkettet sind. Der letzte Block einer Gitterzelle zeigt dann auf den ersten Block der nächsten Gitterzelle; jeder Block zeigt also auf einen Nachfolgeblock oder keinen, aber mehrere Blöcke können auf denselben Folgeblock zeigen.

Die Knoten aus dem Coregraphen liegen in einer eigenen Ebene, die über allen andere liegt, und belegen die Blöcke am Ende, d. h. es gibt einen Blockindex, so dass alle Blöcke ab diesem Index nur Knoten aus dem Coregraphen enthalten. Die Kanten für diese Blöcke müssen in der gleichen Reihenfolge am Ende der Kantenliste liegen. Damit kann der Coregraph sehr einfach und schnell eingelesen werden.

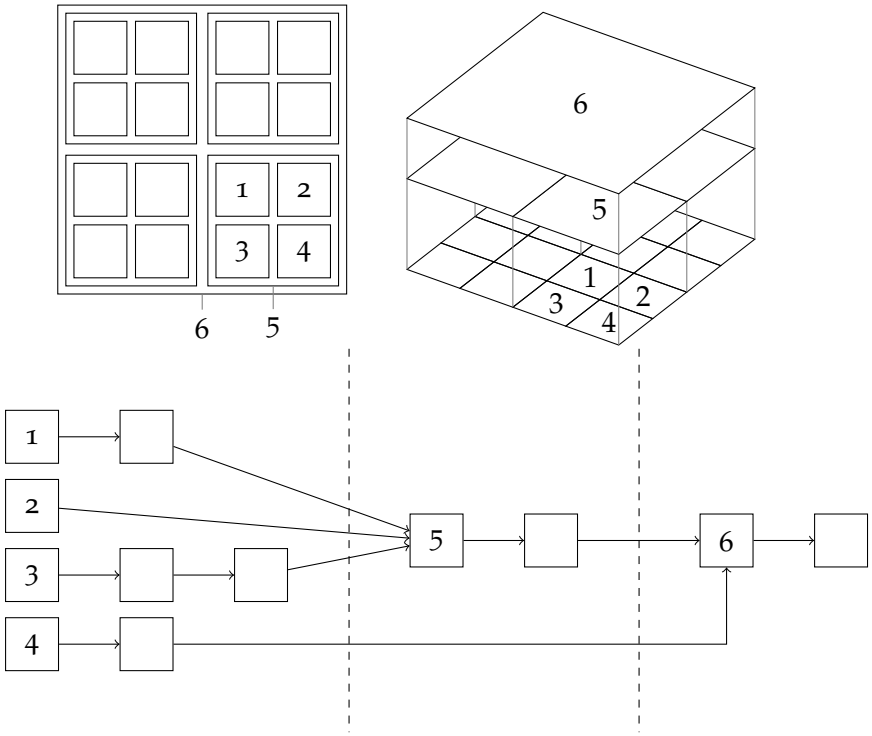
Für jede Zelle in der untersten Ebene existiert genau ein Block, der als Einstieg dient; die Zellen werden dazu einfach durchnummeriert, und die Zellnummer wird als Blocknummer verwendet. Nur die Daten des untersten Gitters (Eckkoordinaten und Gitterdimension) werden gespeichert, um die Zellnummer für den Einstieg berechnen zu können. Alle Knoten, die geographisch in einer Zelle auf der untersten Ebene liegen (aber eventuell tatsächlich in Gitterzellen darüber gespeichert wurden), sind von dem Block der Zellnummer aus auffindbar (siehe Abschnitt 4.1).

3.3 Binärformat

Alle Daten werden als 32-Bit Ganzzahl (Big-Endian, d. h. das höchstwertigste Byte zuerst) gespeichert. Die Geokoordinaten werden in $10^{-7}\,^{\circ}$ auf die nächste Ganzzahl gerundet.

Eine Datei besteht aus fünf aufeinander folgenden Abschnitten, die unten genauer beschrieben werden. Jeder Abschnitt ist dabei an 4 KiByte Blöcken ausgerichtet.

Durch die Aufteilung der Daten für Knoten und Kanten liegen die Daten, die im jeweiligen Schritt benötigt werden, dichter beieinander, und es werden weniger Daten gelesen, die nicht benötigt werden.



Gitterzelle 1 benötigt zwei Blöcke, Zelle 2 nur einen und Zelle 3 drei Blöcke. Gitterzelle 5 enthält Knoten, die von den Position her in den Zellen 1, 2 und 3 liegen, darum zeigt der jeweils letzte Block dieser Zellen auf den ersten Block der Zelle 5. Zelle 4 ist auf der Ebene von Zelle 5 leer, und kann darum direkt auf Zelle 6 zeigen.
Alle Knoten, die im Bereich der Zelle 1 aber auf evtl. höherem Level (d. h. in Block 5 oder 6) liegen, sind vom ersten Block der Zelle 1 aus erreichbar.

Abbildung 3.2: Beispiel: Gitterzerlegung in Blöcke

3.3.1 Metadaten

- Zwei speziell gewählten Zahlen ($4348474F_{16}$ und 66665450_{16}), die sich in ASCII Repräsentation als „CHGOffTP“ lesen. Durch diese Zahlen kann mit großer Wahrscheinlichkeit ausgeschlossen werden, dass unbeabsichtigt eine anders formatierte Datei eingelesen wird.
- Eine Versionsnummer für das Format (1)
- Die Eckdaten für das unterste Gitter:
  - Längengrad für die linke Kante
  - Breitengrad für die untere Kante
  - Breite einer Zelle

  - Höhe einer Zelle
  - Anzahl der Zellen pro Zeile
  - Anzahl der Zellen pro Spalte
- Maximale Anzahl von Knoten pro Block (daraus ergibt sich der Platzbedarf eines Blockes)
- Anzahl der Blöcke in der Datei (jede Zelle benötigt einen Startblock, danach können weitere kommen)
- Index des ersten Coreblocks
- Anzahl der Kanten

3.3.2 Geographische Positionen der Knoten

Für jeden Block:

- Index des nächsten Blocks (oder -1)
- Anzahl der gültigen Einträge in diesem Block
- Für jeden gültigen Eintrag:
  - Längengrad
  - Breitengrad
- Die ungültigen Einträge werden mit Nullen aufgefüllt.

3.3.3 Kantenindizes für die Knoten

Für jeden Block:

- Platzhalter (0)
- Für jeden Knoten (keine Spezialbehandlung für ungültige Einträge):
  - Index der ersten ausgehenden Kante
  - Index der ersten eingehenden Kante
- Index der Kante nach der letzten eingehenden Kante des letzten Knotens.

3.3.4 Kantendaten

Für jede Kante:

- Knoten. Für ausgehende Kanten der Zielknoten, für eingehende der Startknoten.
- Kantengewicht (Reisezeit gerundet in $\frac{9}{325}s$)

3.3.5 Erweiterte Kantendaten

Für jede Kante:

- Länge des nach Reisezeit kürzesten Weges in m
- Für Shortcuts: (ansonsten mit -1 aufgefüllt)
  - Erste ersetzte Kante
  - Zweite ersetzte Kante
  - Kontrahierter Knoten

3.4 Kompression

Die so erstellte Datei ist für das Straßennetz von Deutschland 2 GByte groß; mit xz [LZM] lässt sich die Datei auf 360 MByte komprimieren. Da die Datei aber nicht am Stück gelesen wird, muss die Kompression blockweise erfolgen, d. h. es werden mehrere Stellen benötigt, von denen an dekomprimiert werden kann (die Zuordnung von Positionen in der unkomprimierten Datei und den Einstiegsstellen in der komprimierten Datei wird in einer Indextabelle gespeichert).

Je kleiner die Blöcke, desto schneller ist der Zugriff, wenn nicht am Stück gelesen wird. Allerdings sinkt die Kompressionsrate mit kleineren Blöcken.

Mit einer Blockgröße von 64 KiByte lässt sich die Datei mit xz auf 409 MByte und mit DEFLATE [DEF] auf 640 MByte komprimieren.

In Tabelle 3.1 und Abbildung 3.3 sind Testläufe für verschiedene Strecken mit den verschiedenen Kompressionsoptionen und ohne dargestellt.

Die Unterstützung der ausgewählten Kompressionsformate (xz und eine Variante von DEFLATE mit Indextabelle) ist nicht in Java implementiert, sondern in C++ und wird über Java Native Interface (JNI) eingebunden.

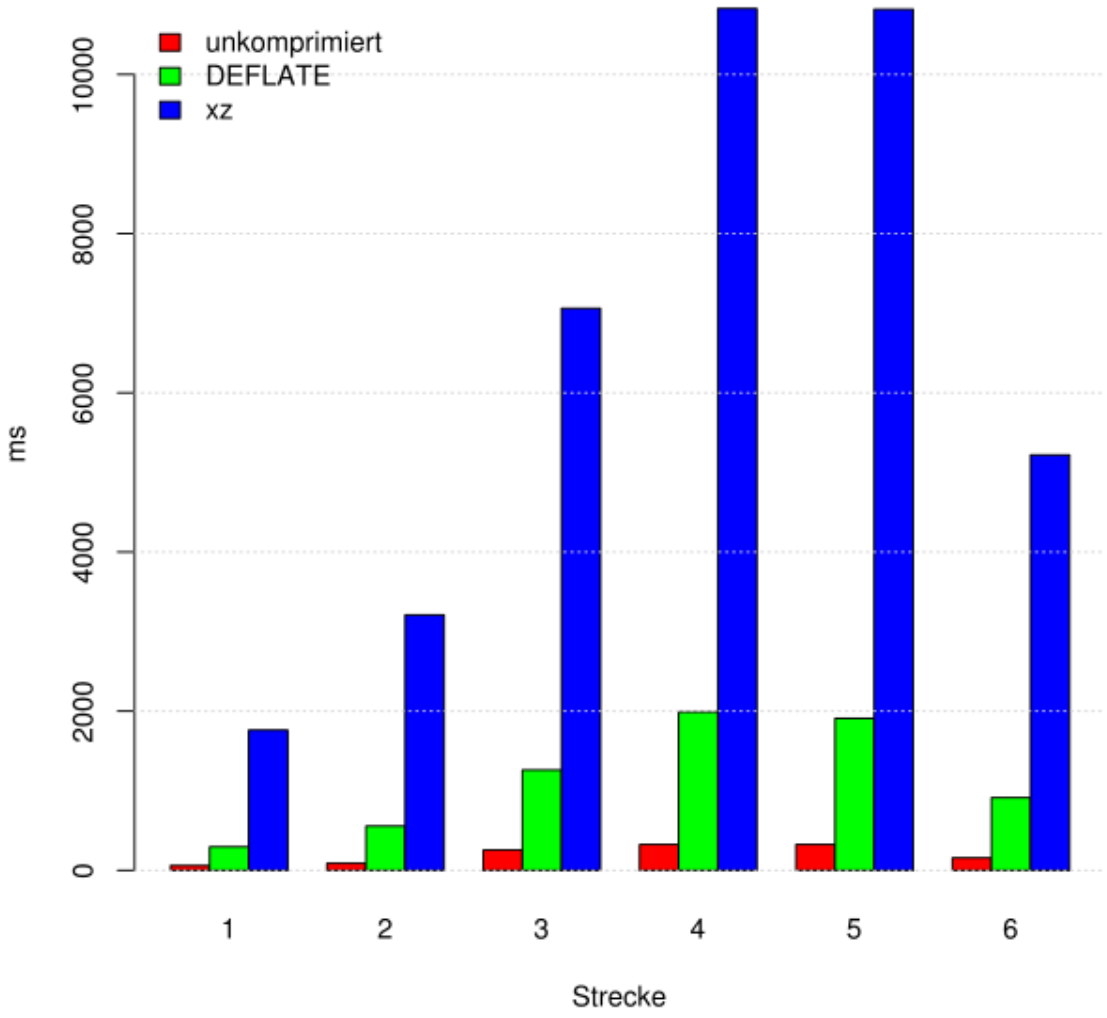


Abbildung 3.3: Durchschnittliche Zeit aus 5 Läufen (nach einem Lauf für den Cache) in ms für verschiedene Strecken

	1. Lauf	2. Lauf	3. Lauf	4. Lauf	5. Lauf	6. Lauf	(jeweils in ms)
Strecke 1	*Informatikgebäude → Stuttgart HBF*						
	48.7456169 °N 9.1070623 °W → 48.7831573 °N 9.1816587 °W						
unkomprimiert	1606	88	58	53	51	50	
DEFLATE	2172	291	291	288	298	308	
xz	3229	1794	1735	1809	1734	1735	
Strecke 2	*Informatikgebäude -> Karlsruhe*						
	48.7456169 °N 9.1070623 °W → 49.0107460 °N 8.4040517 °W						
unkomprimiert	3334	89	86	85	92	87	
DEFLATE	3912	561	554	552	553	554	
xz	4129	3213	3203	3206	3207	3214	
Strecke 3	*Informatikgebäude -> München*						
	48.7456169 °N 9.1070623 °W → 48.1370124 °N 11.5758237 °W						
unkomprimiert	5396	254	258	260	251	249	
DEFLATE	5213	1271	1262	1255	1256	1260	
xz	7974	7052	7094	7052	7049	7054	
Strecke 4	*Informatikgebäude -> Berlin*						
	48.7456169 °N 9.1070623 °W → 52.5199928 °N 13.4385576 °W						
unkomprimiert	9094	325	325	324	322	322	
DEFLATE	6650	2383	1890	1888	1889	1897	
xz	11453	10827	10817	10835	10805	10848	
Strecke 5	*Informatikgebäude -> Hamburg*						
	48.7456169 °N 9.1070623 °W → 53.5438613 °N 10.0104999 °W						
unkomprimiert	335	325	324	324	329	322	
DEFLATE	1896	1905	1916	1891	1920	1918	
xz	10803	10788	10788	10784	10815	10913	
Strecke 6	*Aachen -> Berlin*						
	50.7773246 °N 6.0779156 °W → 52.5199928 °N 13.4385576 °W						
unkomprimiert	2882	153	153	155	156	16	
DEFLATE	2183	921	905	906	905	92	
xz	5461	5221	5217	5219	5221	522	

Je nach vorhergehender Nutzung ist der 1. Lauf wenig aussagekräftig.

Tabelle 3.1: Messdaten für Testläufe mit verschiedenen Kompressionen

4 Algorithmen

In diesem Kapitel werden nun einige zentrale Algorithmen beschrieben, die die in Kapitel 3 beschriebenen Daten einlesen bzw. verarbeiten. Die grundlegenden Algorithmen für die Wegsuche sind bereits in Kapitel 2 vorgestellt worden. Im letzten Abschnitt 4.4 werden Optimierungen für die Darstellung gefundener Wege gezeigt.

4.1 Knotensuche

Algorithmus 4.1 wird zum Finden des zu Geokoordinaten nächstgelegenen Knotens verwendet. Liegen die Geokoordinaten innerhalb des Gitters in einer nicht leeren Gitterzelle, so ist das Ergebnis korrekt.Andernfalls sucht der Algorithmus von einer zufälligen Position aus in Richtung des Ziels nach einem passenden Knoten. Die Suche bricht ab, sobald auf diesem Weg leere Gitterzellen gefunden werden.

Der Algorithmus geht dazu eine Menge von Blöcken durch. Wenn ein Block auf einen weiteren Block zeigt, so wird auch dieser verarbeitet. Da mehrere Blöcke auf denselben Block zeigen können, merkt sich der Algorithmus, welche Blöcke er schon durchsucht hat, damit jeder Block höchstens einmal durchsucht wird. Aus allen Knoten, die beim Durchsuchen der Blöcke angetroffen werden, wird derjenige ausgewählt, der der gesuchten Position am nächsten kommt.

Der Algorithmus startet mit dem ersten Block der Gitterzelle, die entweder die gesuchte Position enthält oder ihr am nächsten kommt. Wenn er von dort aus keinen Knoten findet, wird der erste Block des Coregraphen geladen. Jedes Mal, wenn ein neuer Knoten gefunden wurde, werden anschließend die Nachbarblöcke der Zelle des gefundenen Knotens durchsucht (siehe Abbildung 4.1).

4.2 Laden der für Dijkstra benötigten Teilgraphen

Wie in Abschnitt 2.2.3 beschrieben, wird für die Suche des kürzesten Weges von a nach b der Aufwärtsgraph von a und der Abwärtsgraph zu b, jeweils ohne Coregraph, und der Coregraph benötigt.

Algorithmus 4.2 beschreibt das Laden des Auf- bzw. Abwärtsgraphen. Um langsames Springen in der Datei zu vermeiden, wird die aktuelle Liste der zu ladenden Knoten sortiert

Algorithmus 4.1 Algorithmus zum Finden des zu einer Position nächstgelegenen Knotens

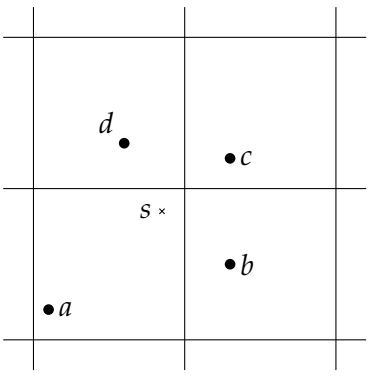
```
1:  function FindPoint(point)
2:      ▷ Der Iterator durchläuft alle Knoten aus einem Block und lädt automatisch den
3:      ▷ Folgeblock. Er bricht ab, wenn er einen Block nach einem Aufruf von continueWith
4:      ▷ nochmal trifft. Es wird also kein Knoten mehrmals geladen.
5:      i ← new NodeGeoIterator
6:      nodeID ← −1, nodeDist ← ∞
7:      nodePoint ← point                    ▷ noch kein Knoten, wähle Zielposition als Basis
8:      loop
9:          lastNodeID ← nodeID
10:         ▷ Suche ausgehend von aktuellem Knoten
11:         gridx ← clip([0 ... grid.width − 1], (nodePoint.x − grid.base_x)/grid.cellWidth)
12:         gridy ← clip([0 ... grid.height − 1], (nodePoint.y − grid.base_y)/grid.cellHeight)
13:         base ← gridy · grid.width + gridx
14:         i.continueWith(base)
15:         while i.next() do
16:             d ← ||i.nodePoint − point||
17:             if d < nodeDist then
18:                 nodeDist ← d, nodePoint ← i.nodePoint, nodeID ← i.nodeID
19:             end if
20:         end while
21:         if nodeID ≠ lastNodeID then
22:             next    ▷ Neuen Knoten gefunden. Starte Suche von dessen Position aus neu.
23:         end if
24:         if nodeID = −1 then
25:             ▷ Kein Knoten gefunden. Wähle zufälligen Knoten und starte von dort
26:             ▷ Der Algorithmus konvergiert dann Richtung Zielposition,
27:             ▷ solange keine Löcher (leere Blöcke) dazwischen die Suche beenden.
28:             nodeID ← 0, nodePoint ← nodes[0].point
29:             next
30:         end if
31:         ▷ Siehe Fortsetzung...
```

und dann der Reihenfolge nach abgearbeitet. Erst wenn die sortierte Liste leer ist, wird eine neue sortierte Liste erstellt.

Die Repräsentation des Graphen im Speicher ist darauf ausgelegt, zu einem Knoten ausgehende Kanten auszugeben. Mit diesem Graphen berechnet der Dijkstra-Algorithmus (siehe Abschnitt 2.1.2) den kürzesten Weg von a nach b. Der Kantenindex wird benötigt um danach Shortcut-Kanten der CH im gefundenen Weg zu ersetzen.

Algorithmus 4.1 Algorithmus zum Finden des zu einer Position nächstgelegenen Knotens (Fortsetzung)

```
32:         ▷ Da nodeID = lastNodeID ist base der Startblock für nodePoint
33:         ▷ Suche auch in benachbarten Zellen (d. h. von deren Startblock aus),
34:         ▷ da an Zellenrändern Knoten aus der Nachbarzelle näher liegen können.
35:         foreach block in Startblöcken der Nachbarzellen von base do
36:             i.continueWith(block)
37:             while i.next() do
38:                 d ← ||i.nodePoint − point||
39:                 if d < nodeDist then
40:                     nodeDist ← d, nodePoint ← i.nodePoint, nodeID ← i.nodeID
41:                 end if
42:             end while
43:         end
44:         if nodeID = lastNodeID then
45:             ▷ Keinen besseren Knoten gefunden - beende Suche
46:             break
47:         end if
48:     end loop
49: end function
```



Bei der Suche nach dem nächstgelegen Knoten zu s wird zuerst der Knoten a gefunden. Es kann aber in Nachbarzellen Knoten geben, die näher an s liegen - darum müssen auch diese durchsucht werden.

Abbildung 4.1: Zellübergreifende Suche von Knoten

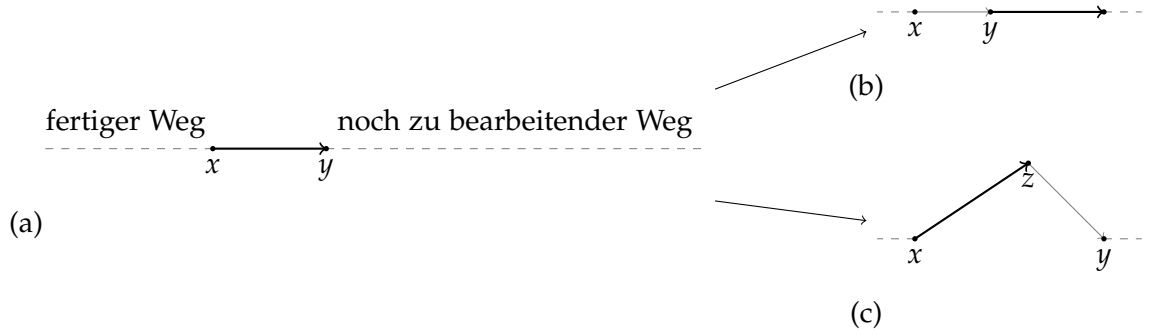
Algorithmus 4.2 Laden des Auf- bzw. Abwärtsgraph von bzw. zu einem Knoten

```
1:  function loadGraph(node, edgeDirection)
2:      graph ← new Graph()
3:      todo ← ∅
4:      if block_no(node) < firstCoreBlock then          ▷ Coregraph wird separat geladen
5:          todo ← todo ∪ {node}
6:      end if
7:      while todo ≠ ∅ do
8:          run ← sort(todo), todo ← ∅
9:          foreach n in run do
10:             foreach (edgePeer, edgeDist, edgeID) in up_edges(n, edgeDirection) do
11:                 if edgeDirection = OUT then
12:                     graph ← graph ∪ {n --edgeDist, edgeID--> edgePeer}
13:                 else
14:                     graph ← graph ∪ {edgePeer --edgeDist, edgeID--> n}
15:                 end if
16:                 if block_no(edgePeer) < firstCoreBlock then
17:                     todo ← todo ∪ {edgePeer}
18:                 end if
19:             end
20:         end
21:     end while
22: end function
```

4.3 Shortcut-Ersetzung

Zur Darstellung des Weges müssen die Shortcuts in dem Weg, der vom Dijkstra-Algorithmus in der Contraction-Hierarchy gefunden wurde, durch die ursprünglichen Kanten ersetzt werden. Auf das Ersetzen eines Shortcuts könnte verzichtet werden, wenn der Shortcut weit genug weg vom aktuell dargestellten Kartenausschnitt entfernt ist, so dass keine Ersetzung in den Kartenausschnitt reichen kann. Auch wenn die euklidische Länge des vom Shortcut abgekürzten Weges klein genug ist, so dass die Ersetzung in der aktuellen Zoomstufe in der Darstellung nur geringen oder gar keinen Einfluss hat, könnte die Ersetzung eines Shortcuts entfallen (siehe auch Abschnitt 4.4).

Zur Ersetzung wird der Weg stückweise zu einem neuen Weg; eine Kante wird vom Anfang des alten Weges entfernt, evtl. wiederholt ersetzt (bei jeder Ersetzung wird eine weitere Kante vor den alten Weg gesetzt), und ans Ende des neuen Weges angehängt (siehe Abbildung 4.2).



Wenn (x, y) kein Shortcut ist oder nicht ersetzt werden soll, wird die Kante direkt übernommen und, falls weitere Kanten kommen, die nächste Kante betrachtet (b). Ansonsten wird der Shortcut (x, y) ersetzt durch die ursprünglichen Kanten (x, z) und (z, y). Die Kante (z, y) wird dazu vor den noch zu bearbeitenden Weg gehängt, und (x, z) wird als nächste Kante betrachtet (c).

Abbildung 4.2: Durchlaufen der Wegkanten zur Shortcut-Ersetzung

4.4 Darstellung des Wegs

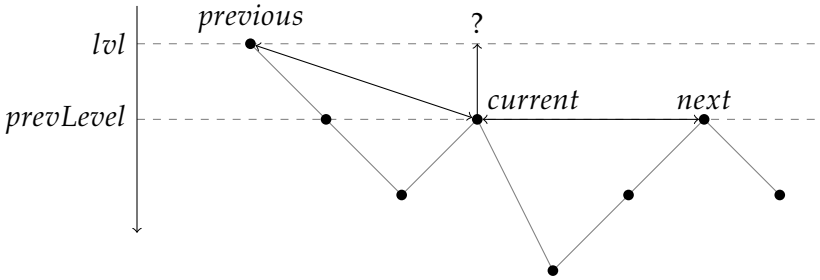
Für die Darstellung des Weges sind nicht alle Knotenpunkte notwendig. Zum einen können Punkte aussortiert werden, wenn sie keine Verbindung in den aktuell angezeigten Kartenausschnitt haben, zum andern können Knoten, die bezüglich der aktuellen Zoomstufe nahe genug an Nachbarknoten liegen, weggelassen werden.

Im Folgenden wird angenommen, dass der anzeigte Kartenausschnitt ein an den Koordinatenachsen ausgerichtetes Rechteck ist, da auch der mapsforge-Renderer nur diesen Modus verwendet. Im Falle einer gedrehten Karte könnte als Kartenausschnitt das begrenzende Rechteck des gedrehten Kartenausschnitts verwendet werden.

Algorithmus 4.3 beschreibt die Berechnung, ab welcher Zoomstufe (und darüber) ein Wegpunkt für die Darstellung verwendet werden soll. Die Entscheidung, ob ein Knoten auf der Zoomstufe lvl bereits sichtbar sein muss (Zeile 31 in Algorithmus 4.3), ist in Abbildung 4.3 visualisiert.

Um nur den Teil des Weges zu zeichnen, der im aktuellen Kartenausschnitt zu sehen ist, wird ein Binärbaum erstellt, in dem jeder Knoten für einen Teilweg steht; wenn ein Knoten Kindknoten hat, so ist dieses Intervall die Vereinigung der Intervalle der Kindknoten, die sich dabei überlappen. Um diesen Baum zu erstellen wird rekursiv der Weg solange halbiert, bis ein Teilweg weniger als 6 Wegpunkte hat. Bei der Aufteilung wird am mittleren Wegpunkt geteilt, der dann in beide Teilwege aufgenommen wird. Für diese Teilwege werden dann die begrenzenden Rechtecke (parallel zum Koordinatensystem) berechnet und mit dem Intervall zusammen im Baumknoten gespeichert. Da der Weg für die Ausgabe geglättet wird, werden die Intervalle nach dem Berechnen der begrenzenden Rechtecke um den vorigen und um den nächsten Wegpunkt erweitert, sofern ein solcher existiert, da die Richtung des Folgewegstücks Einfluss auf die Glättung hat (siehe Algorithmus 4.4).

Zum Darstellen eines Wegknotens wird geprüft, wie der aktuelle Kartenausschnitt zum begrenzenden Rechteck liegt (siehe Abbildung 4.4). Sind sie disjunkt, so gibt es nicht zu zeichnen; liegt das Rechteck komplett im Kartenausschnitt, oder hat der Knoten keine Kindknoten, werden alle Wegknoten gezeichnet – wobei in der Mitte des Weges Knoten, die auf der aktuellen Zoomstufe nicht sichtbar sind, ausgelassen werden. Ansonsten wird die Prozedur rekursiv mit den Kindknoten wiederholt.

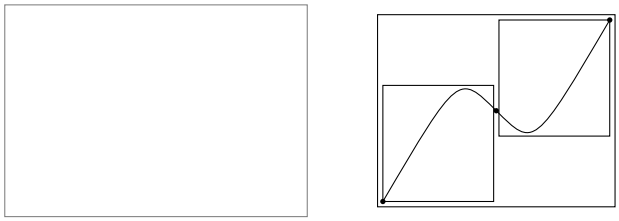


Wenn sowohl der Abstand $d(previous, current)$ als auch $d(current, next)$ auf der Zoomstufe lvl klein genug sind, ist der Wegknoten *current* auf der Zoomstufe lvl nicht mehr sichtbar. Wenn mindestens ein Abstand zu groß ist, wird der Knoten auf die Zoomstufe lvl gehoben.

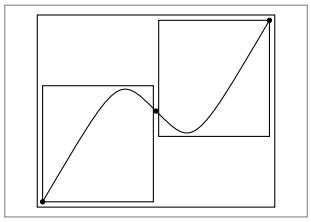
Abbildung 4.3: Entscheidung, ob ein Knoten auf einer kleineren Zoomstufe angezeigt werden soll

Algorithmus 4.3 Berechnen der Zoomstufen ab denen Wegpunkte für die Darstellung verwendet werden

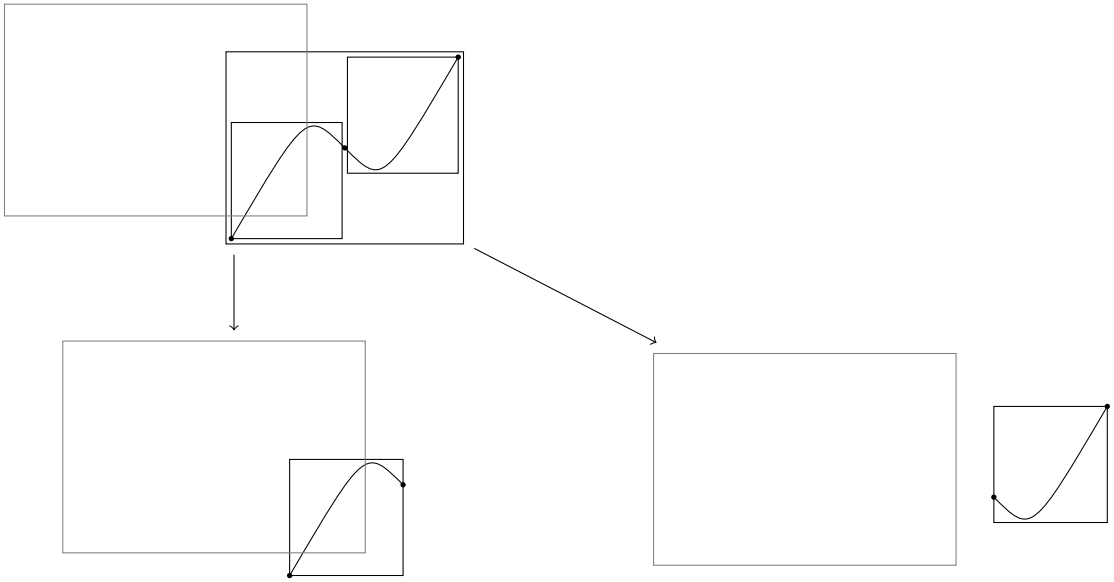
```
1:  function PointsZoomLevel(points)
2:      ▷ Die Punkte in points liegen bereits als Pixelkoordinaten für Zoomstufe 0 vor;
3:      ▷ jede Zoomstufe vergrößert das Bild um Faktor 2.
4:      prevLevel ← 32                          ▷ Zoomstufe ab der immer alles angezeigt wird
5:      zoomMinPixels ← 16
6:
7:      count ← length(points)
8:      ▷ Am Anfang sind alle Punkte nur im „höchsten“ Level sichtbar
9:      ∀0 ≤ i < count : visibleZoomLevel[i] ← prevLevel
10:     ▷ Start und Ende sind aber immer sichtbar
11:     visibleZoomLevel[0] ← 0, visibleZoomLevel[count − 1] ← 0
12:     ▷ Berechne für einige Zoomstufen lvl, welche Punkte aus der vorigen Zoomstufe
13:     ▷ sichtbar bleiben müssen, da sie zu weit weg von Nachbarpunkten liegen.
14:     foreach lvl in { 20, 16, 14, 12, 8, 4, 0 } do
15:         ▷ Der Abstand wird für die Zoomstufe prevLevel − 1 berechnet
16:         zoom ← 2^(prevLevel−1)
17:         ▷ Durchlaufe alle sichtbaren Knoten der Zoomstufe prevLevel mit Nachbarknoten
18:         ▷ previous ist allerdings der Vorgängerpunkt auf Zoomstufe lvl
19:         previous ← 0                        ▷ Punkt 0 ist immer sichtbar
20:         next ← 1
21:         ▷ Bricht ab bei next = count − 1 da visibleZoomLevel[count − 1] = 0
22:         while visibleZoomLevel[next] > prevLevel do
23:             next ← next + 1
24:         end while
25:         while next < count − 1 do
26:             current ← next, next ← next + 1
27:             while visibleZoomLevel[next] > prevLevel do
28:                 next ← next + 1
29:             end while
30:             if ‖points[current], points[previous]‖ · zoom ≥ zoomMinPixels
31:               ∨‖points[current], points[next]‖ · zoom ≥ zoomMinPixels then
32:                 ▷ current muss auch auf Zoomstufe lvl sichtbar sein
33:                 visibleZoomLevel[current] ← lvl
34:                 previous ← current
35:             end if
36:             ▷ Andernfalls verschwindet current ab dieser Zoomstufe und darunter.
37:             ▷ previous ist dann auch in der nächsten Runde der Vorgänger auf Stufe lvl
38:         end while
39:         prevLevel ← lvl
40:     end
41:     return visibleZoomLevel
42: end function
```



(a) disjunkt: es wird nichts gezeichnet



(b) Weg komplett im Kartenausschnitt: der komplette Weg wird gezeichnet



(c) anderweitige Überlappung: Teilwege werden einzeln geprüft. Wenn keine weitere Aufteilung vorhanden ist, wird der komplette Weg gezeichnet

Abbildung 4.4: Mögliche Lagen der begrenzenden Rechtecke zum dargestellten Kartenausschnitt

Algorithmus 4.4 Erstellen des Baums mit begrenzenden Rechtecken für Teilwege

```
1:  struct TreeNode is
2:      first, last
3:      boundingBox
4:      child1, child2
5:  end
6:  function CreateNode(points, first, last)
7:      if last - first > 4 then
8:          mid ← ⌊(first + last)/2⌋
9:          c1 ← CreateNode(points, first, mid), c2 ← CreateNode(points, mid, last)
10:         return TreeNode(first ← min{c1.first, c2.first}, last ← max{c1.first, c2.first},
11:                         boundingBox ← merge(c1.boundingBox, c2.boundingBox),
12:                         child1 ← c1, child2 ← c2)
13:     end if
14:     return TreeNode(first ← max{0, first − 1}, last ← min{count − 1, last + 1},
15:                         boundingBox ← getboundingbox{points[i] | first ≤ i ≤ last},
16:                         child1 ←null, child2 ←null)
17: end function
```

5 Androidanwendung „Offline ToureNPlaner“

In diesem Kapitel werden die verschiedenen Funktionen der Androidanwendung gezeigt. Die Darstellung der Karte reagiert auf die üblichen Gesten zum Zoomen und Bewegen der Karte. Mit einem langen Druck auf die Karte lassen sich zuerst der Ziel- und dann der Startpunkt setzen. Beide können mit „Drag'n'Drop“ auf der Karte verschoben werden, und mit einem Klick auf das „Mülleimer“-Symbol entfernt werden.

Die gesetzten Punkte werden von der Anwendung automatisch auf den nächst gelegenen Wegpunkt verschoben (siehe Abschnitt 4.1).

5.1 Einstellungen

Zur Inbetriebnahme der Anwendung müssen zuerst die gewünschten Daten auf das Androidgerät geladen werden. Danach werden im „Settings“-Dialog die Pfade zu den Dateien auf dem Gerät konfiguriert (siehe Abbildung 5.1). Wenn auf dem Androidgerät auch der „OI File Manager“ [OIF] installiert ist, können die Dateien und Verzeichnisse mit Hilfe eines Dialogs ausgewählt werden (siehe Abbildung 5.2).

5.2 Suche

In Abbildung 5.3 ist eine Beispielsuche zu sehen. Wenn ein Ergebnis ausgewählt wird, wird der entsprechende Ort auf der Karte als Ziel ausgewählt. Die Suche reagiert so schnell wie möglich auf jede Änderung der Eingabe; allerdings kann eine laufende Suche nur nach Rückgabe einer Zeile unterbrochen werden (dies ist eine Limitierung in osmfind [OSMb]), bevor eine neue Suche gestartet wird.

5.3 Routen

Wenn GPS aktiviert ist (siehe Abbildung 5.4), und kein manueller Startpunkt gesetzt wurde oder die Ansicht auf die GPS-Position zentriert ist (zweiter Knopf von rechts in der Menüleiste), wird als Startpunkt die GPS-Position verwendet (siehe Abbildung 5.5).

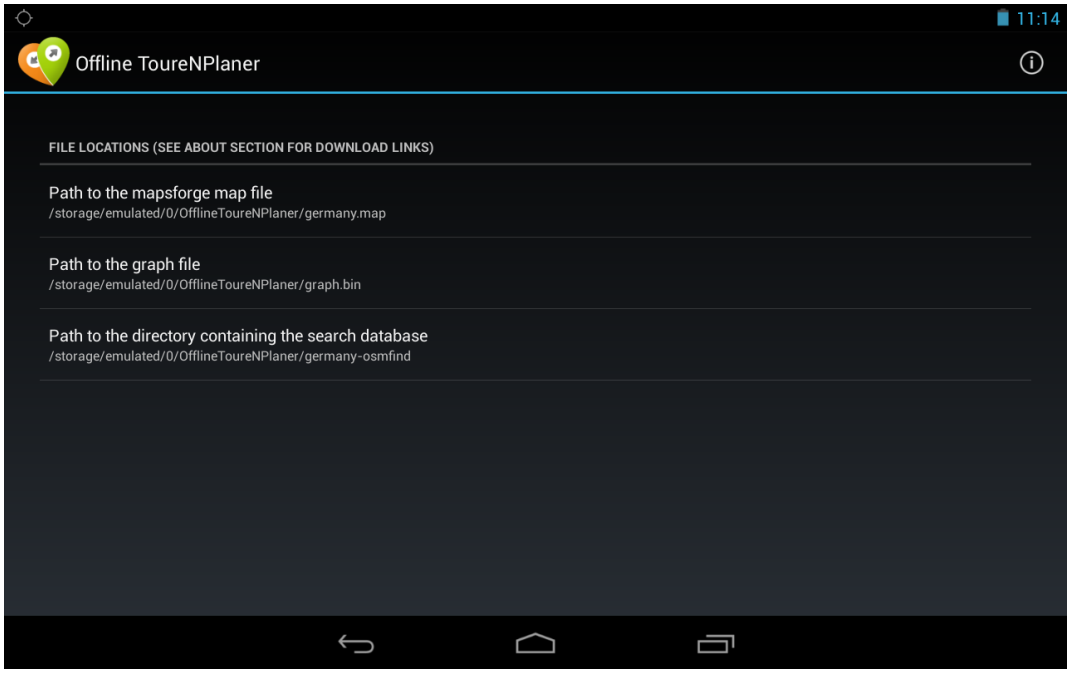


Abbildung 5.1: „Offline ToureNPlaner“ Einstellungen

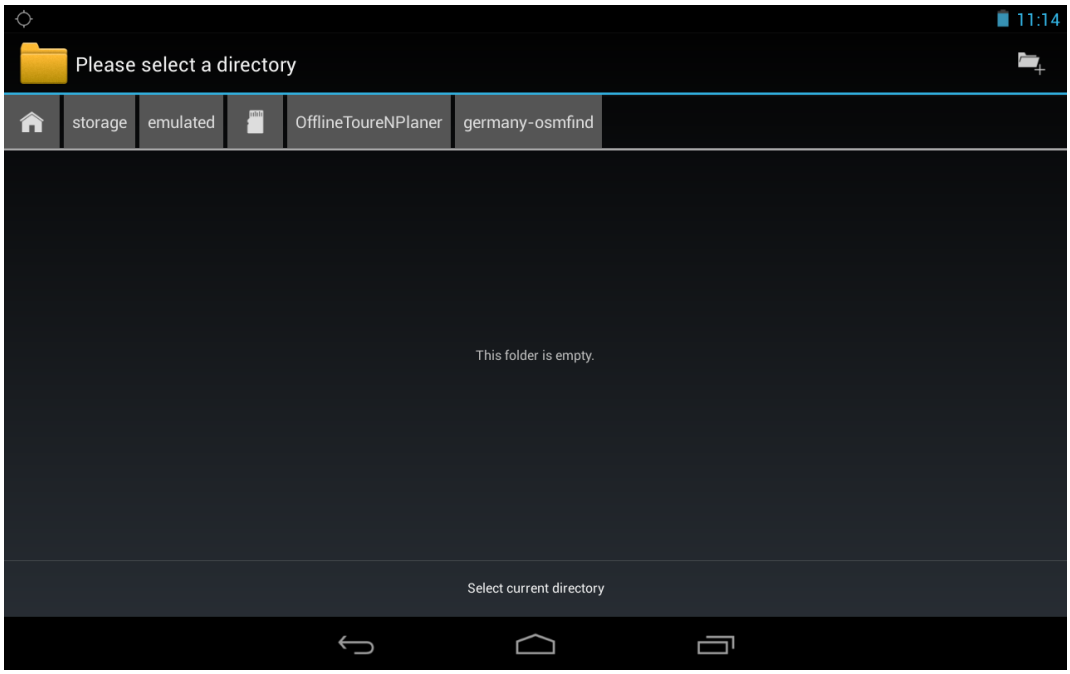


Abbildung 5.2: „Offline ToureNPlaner“ Verzeichnisauswahl

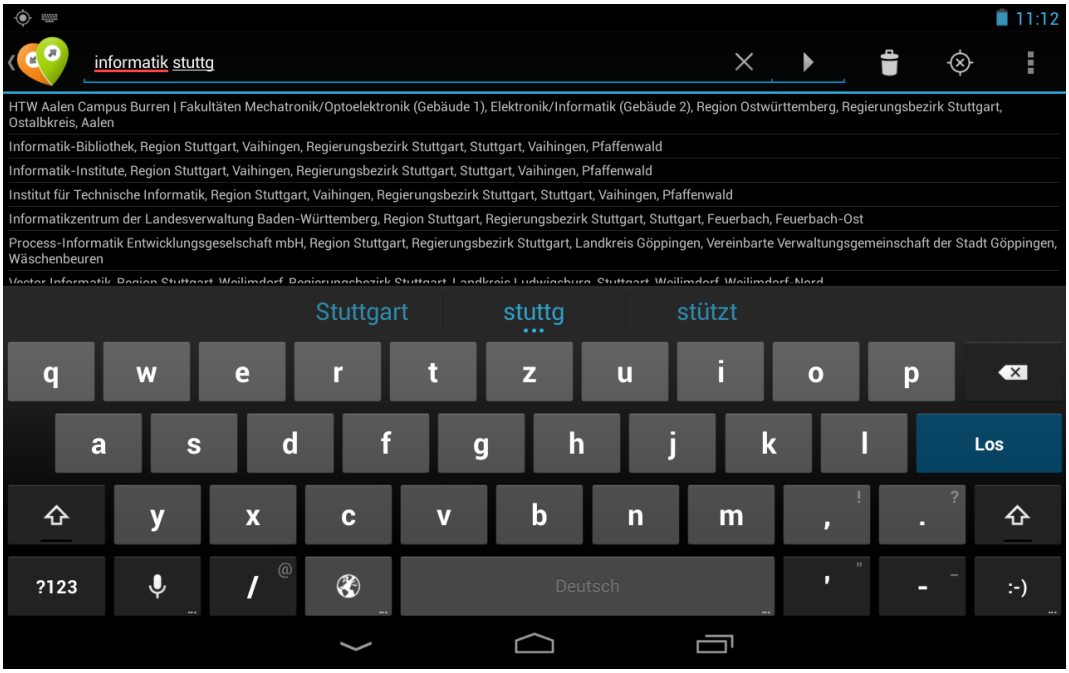


Abbildung 5.3: „Offline ToureNPlaner“ Suche

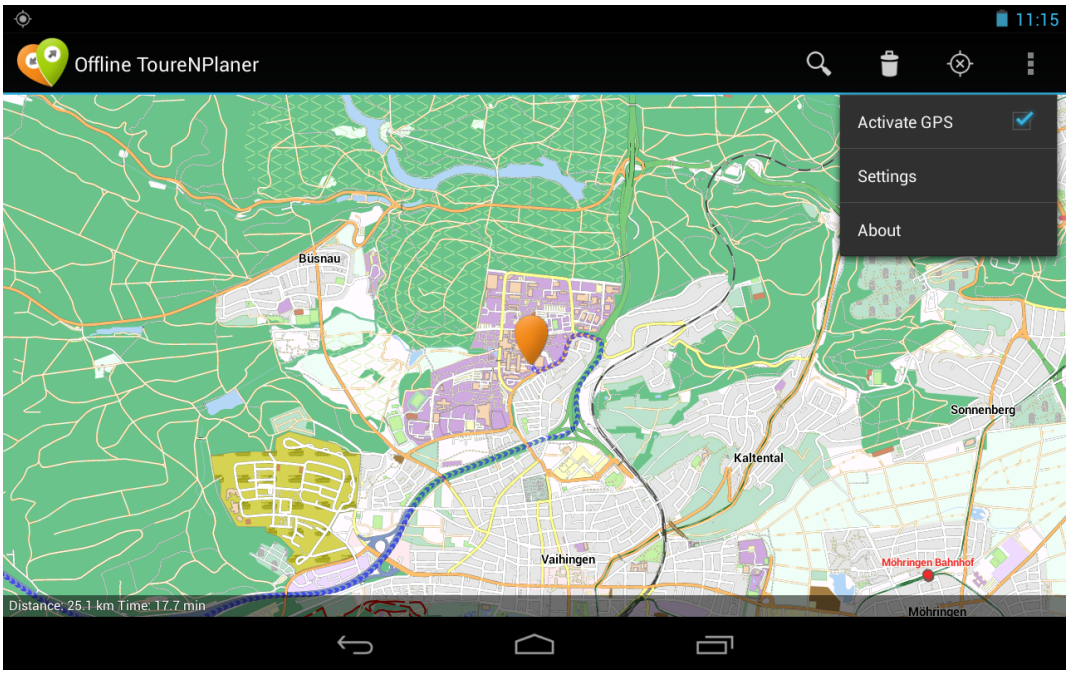


Abbildung 5.4: „Offline ToureNPlaner“ Menü

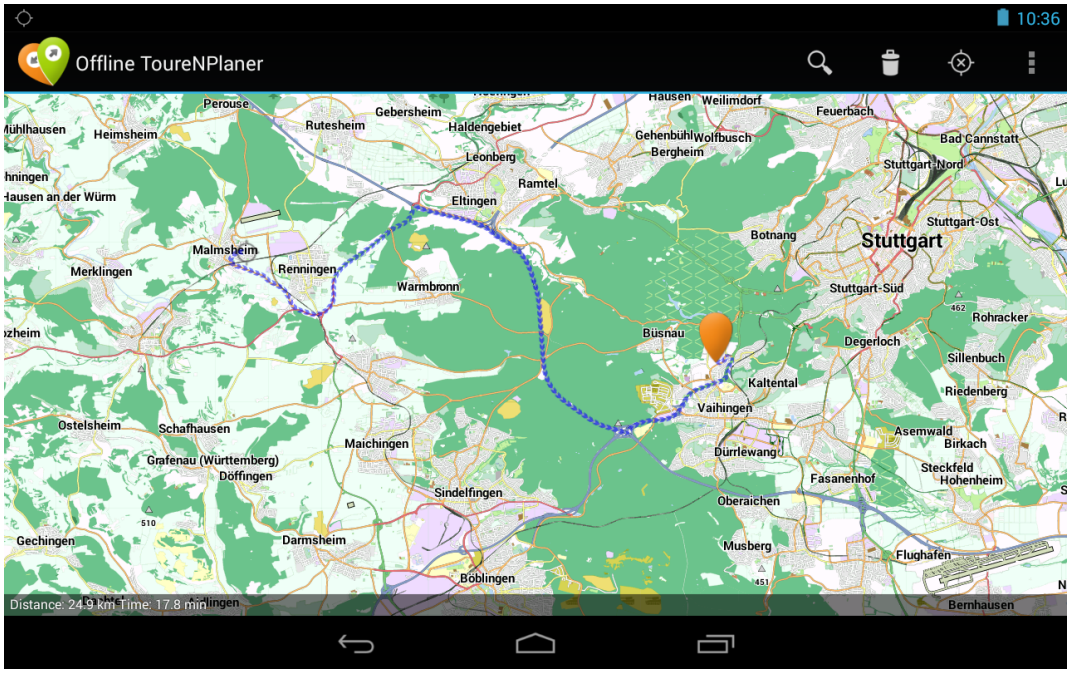


Abbildung 5.5: „Offline ToureNPlaner“ Verwendung der GPS-Position

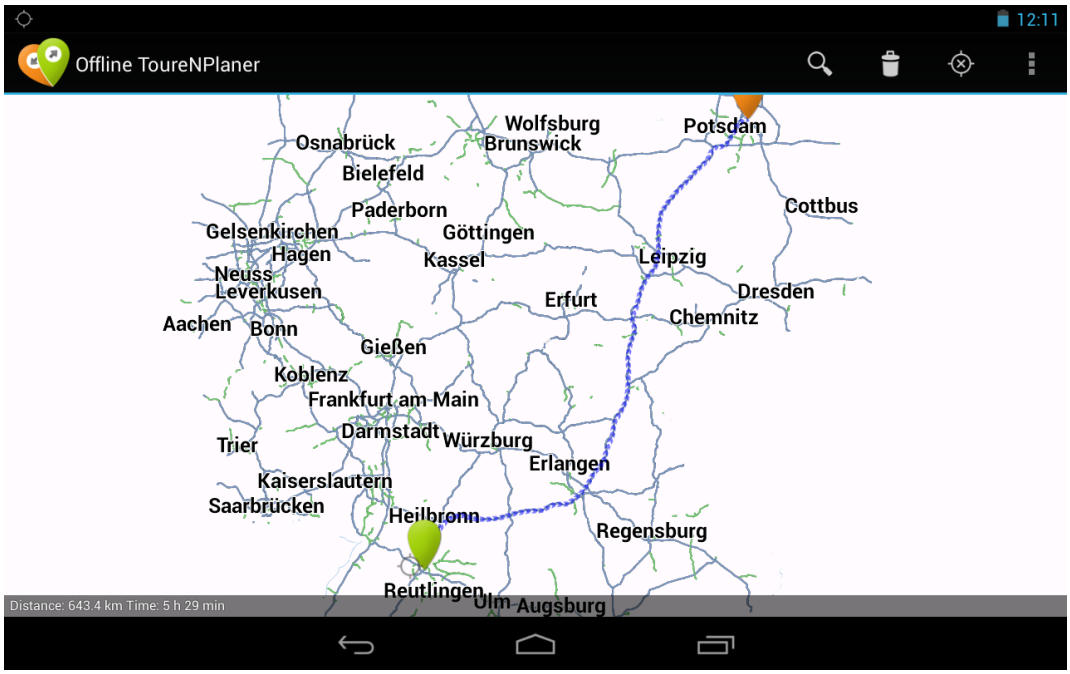


Abbildung 5.6: „Offline ToureNPlaner“ Route mit gesetztem Start- und Endpunkt

Der Startpunkt kann auch manuell gesetzt werden, um z. B. größere Routen unabhängig vom aktuellen Standort zu planen, oder falls auf dem Androidgerät keine GPS-Ortung verfügbar ist (siehe Abbildung 5.6).

6 Zusammenfassung und Ausblick

Mit dem „Offline ToureNPlaner“ steht ein brauchbarer Routenplaner für Androidgeräte zur Verfügung, der ohne Internetverbindung auskommt. Das Suchen des kürzesten Wegs ist schnell, lediglich für lange Strecken ist für das Ersetzen der Shortcutkanten ein paar Sekunden Geduld erforderlich. Durch die Integration von mapsforge als Kartenrenderer, der zwar bei kleinen Zoomstufen etwas langsam ist, dafür aber mit sehr kompakten Kartendaten auskommt, und osmfind als „Point-of-Interest“-Suche entstand eine Androidanwendung, die sich auf Reisen aber auch im täglichen Leben einsetzen lässt.

Ausblick

Um die kürzeste Wege-Suche zu beschleunigen, könnte das Ersetzen der Shortcuts nur soweit vorgenommen werden, wie, abhängig von dem aktuell dargestellten Kartenausschnitt und der Displayauflösung, für eine grobe Darstellung notwendig ist (siehe Abschnitt 4.3). In einem zweiten Schritt kann dann im Hintergrund die komplette Ersetzung vorgenommen werden, um auf Änderungen schnell reagieren zu können.

Interessant wäre auch eine Erweiterung auf mehrere Fortbewegungsmethoden; in der aktuellen Implementierung ist in einem Graph nur eine Methode gespeichert, z. B. für ein normales Auto, für das je nach Straßentyp eine bestimmte Fortbewegungsgeschwindigkeit angenommen wird. Dies könnte ergänzt werden um Wege und Zeiten für Fahrradfahrer und Fußgänger, oder spezielle Einschränkungen wie „Auto, aber nicht schneller als 80 km/h“.

Für den Einsatz zur echten Navigation fehlt eine Darstellung, und teilweise auch die Daten, die auf die konkrete Umgebung eingeht. So fehlt das Einblenden von Hinweisen „Im nächsten Kreisverkehr die 3. Ausfahrt“ oder „in 100m in die Straße Richtung ... einbiegen“, das Vorlesen dieser Hinweise durch eine „Text-To-Speech“-Software und die visuelle Umsetzung dazu, d. h. Schrägansicht auf die Karte und eine Orientierung der Karte, so dass „Oben“ auf der Karte dem Blick nach vorne entspricht.

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

Literaturverzeichnis

[AMOT90] R. K. Ahuja, K. Mehlhorn, J. Orlin, R. E. Tarjan. Faster algorithms for the shortest path problem. *J. ACM*, 37(2):213–223, 1990. doi:10.1145/77600.77615. URL http://doi.acm.org/10.1145/77600.77615. (Zitiert auf Seite 16)

[Bah12] D. Bahrdt. *Multimodale Bereichsanfragen im Kontext von Routenplanern*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2012. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3326&engl=0. (Zitiert auf Seite 5)

[CHCa] CHConstructor. https://github.com/ToureNPlaner/CHConstructor. (Zitiert auf Seite 8)

[CHCb] Für Offline ToureNPlaner erweiterter CHConstructor. https://github.com/stbuehler/CHConstructor. (Zitiert auf Seite 8)

[DEF] DEFLATE. https://tools.ietf.org/html/rfc1951. (Zitiert auf den Seiten 7 und 28)

[Dij59] E. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959. URL http://www.bibsonomy.org/bibtex/2a0cbd6f680048146f2898942717a9a5e/wvdaalst. (Zitiert auf Seite 5)

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In C. C. McGeoch, Herausgeber, *WEA*, Band 5038 von *Lecture Notes in Computer Science*, S. 319–333. Springer, 2008. (Zitiert auf den Seiten 6 und 13)

[JNI] Java Native Interface Specification. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html. (Zitiert auf Seite 7)

[LZM] Lempel–Ziv–Markov chain algorithm. http://www.7-zip.org/sdk.html. (Zitiert auf den Seiten 7 und 28)

[MFOa] mapsforge. https://code.google.com/p/mapsforge/. (Zitiert auf Seite 5)

[MFOb] Modifiziertes mapsforge-Plugin für [TOU]. https://github.com/tourenplaner/mapsforge-fork. (Zitiert auf Seite 7)

[OIF] Androidanwendung OI File Manager. https://play.google.com/store/apps/details?id=org.openintents.filemanager. (Zitiert auf Seite 41)

[OSMa] OpenStreetMap. http://www.openstreetmap.org/. (Zitiert auf Seite 6)

[OSMb] osmfind: Offline OpenStreetMap text search. http://cgit.funroll-loops.de/osmfind/. (Zitiert auf den Seiten 5, 7 und 41)

[Sch13] N. Schnelle. Distributed Shortest-Path Computation. Bachelorarbeit: Universität Stuttgart, Institut für Formale Methoden der Informatik, Algorithmik, 2013. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BCLR-0020&engl=0. (Zitiert auf Seite 15)

[Sø] Søren Sandmann Pedersen. The Radix Heap. http://ssp.impulsetrain.com/2013-05-25_The_Radix_Heap.html. (Zitiert auf Seite 16)

[TOU] ToureNPlaner-Androidanwendung. https://play.google.com/store/apps/details?id=de.uni.stuttgart.informatik.ToureNPlaner. (Zitiert auf den Seiten 8 und 51)

[XZJ] xz-jni Implementierung. https://github.com/stbuehler/xz-jni. (Zitiert auf Seite 7)

Alle URLs wurden zuletzt am 6. 8. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift