Institut für Formale Methoden der Informatik
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2335

Effiziente Textsuche in OpenStreetMap-Daten auf mobilen Geräten

Daniel Bahrdt

Studiengang:	Informatik
Prüfer:	Prof. Stefan Funke
Betreuer:	Prof. Stefan Funke
begonnen am:	12. Mai 2011
beendet am:	11. November 2011
CR-Klassifikation:	H.3.3

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

In dieser Arbeit sollen mehrere Möglichkeiten zur effizienten Textsuche in OpenStreetMap-Daten vorgestellt werden. Das Ziel ist ein Androidprogramm, mit welchem nach Zeichenketten in OpenStreetMap-Daten gesucht werden kann. Zur effizienten Suche bieten sich hier Hash-Verfahren und vor allem Baumstrukturen, wie Patrica- oder HAT-Tries, an. Für die Suche auf Mobilgeräten kann eine Datenstruktur verwendet werden, die nur Lesezugriffe, jedoch keine Veränderungen ermöglicht. Neben einem auf minimalen Hash-Funktionen basierenden Verfahren wurde ein Patricia-Trie in serialisierter Form in einem Byte-Feld abgelegt. Die Datenstruktur ermöglicht so eine Suche nach Präfixen in $O(k)$, mit k der Länge des gesuchten Präfixes, bei guter Cache-Nutzung. Die Baumstruktur erwies sich dabei dem Hash-Verfahren überlegen. Eine potentielle Erweiterung, die Schnittoperationen und die Suche nach mehreren Zeichenketten ermöglicht, soll im Abschnitt Ausblick kurz umrissen werden.

2 OpenStreetMap

In diesem Kapitel soll zunächst grob geklärt werden, wie Geoinformationen in OpenStreetMap abgebildet werden. Neben rein geometrischen Informationen wie der Position bedarf es auch Informationen der Topologie der Punkte sowie weiterer, die Semantik beeinflussender Informationen. Geometrische Informationen werden als Gleitkommazahl im WGS84-Geo-Referenzsystem abgespeichert. Dieses bildet die Erde auf Basis eines Referenzellipsoiden in einen zweidimensionalen Raum ab. Eine genauere Beschreibung findet sich in [IA00].

2.1 Datentypen

In OpenStreetMap sind derzeit 3 Grunddatentypen definiert: Knoten, Wege und Relationen. Diese können zusätzliche Informationen (tags) besitzen, die teilweise die Semantik des Grunddatums ändern. Der für diese Arbeit vorerst wichtigste Tag ist der Name-Tag, welcher Knoten, Wegen oder Relationen einen Namen zuordnet. Knoten (node) bilden das kleinste Datum. Sie enthalten die geometrische Information, repräsentieren also einen Punkt auf dem WGS84 Ellipsoid. Points-of-Interest (“interessanter Ort”, PoI) werden oft mit einem einzelnen Knoten und zugehörigen Tags (Name, PoI-Typ) abgespeichert. Ein Weg (way) stellt einen Kantenzug dar und besteht somit aus mehreren Knoten und einer impliziten Verbindungsrelation über alle im Kantenzug enthaltenen Knoten. Flächen werden durch geschlossene Wege mit einem entsprechenden Tag abgebildet. Relationen (relations) gruppieren mehrere Daten unterschiedlicher Typen zu einer Gruppe, welche wiederum Tags besitzen kann. Beispielsweise besteht der Neckartalweg aus vielen einzelnen Wegen welche zur Relation Neckartalweg zusammengefasst werden. Auch große Flächen z.B. Länder können durch Zusammenfassung entsprechender Wege dargestellt werden. Entsprechende Tags sorgen für die Semantik der Relation.

2.2 Dateiformate

Diese formalen Datentypen können auf unterschiedliche Weise für den Rechner abgebildet werden. Eine menschen- und maschinenlesbare Variante stellt dabei die Extensible Markup Language (XML) dar. Neben dieser Darstellung wird hauptsächlich noch ein Binärformat, welches mit google protocol buffers [goo] umgesetzt wurde, genutzt. Im folgenden soll kurz das XML-Format dargestellt werden, da dieses für den Einstieg am einfachsten umzusetzen ist, für sehr große Daten auf Grund des hohen Overheads jedoch ungeeignet ist. Zeichenketten, für den Menschen interessante Informationen oder aber auch die Semantik der Topologie

verändernde Informationen werden durch das `tag`-Element gespeichert. So werden Name, PoI-Typen oder z.B. Zugangsbeschränkungen durch `tag`-Elemente angegeben. Knoten werden mit einem `node`-Element abgebildet. Dieses kann ein `tag`-Element als Kind-Element besitzen. Zusätzlich werden Position in WGS84 und eine ID abgespeichert. Wege werden durch ein `way`-Element abgebildet. Um einem Weg einen Knoten zuzuordnen, wird das `nd`-Element, welches einen Knoten durch seine ID referenziert, als Kind-Element hinzugefügt. Zusätzliche Tags können dabei die Semantik des Weges beeinflussen und diesen z.B. als Fläche definieren. Relationen werden mit dem `relation`-Element abgebildet. Um andere Datentypen zu referenzieren, die dieser Relation angehören sollen, wird das `member`-Element genutzt. Dieses referenziert die gewollten Elemente über ihre ID. Auch Relationen erhalten Tags, die ihre Semantik beeinflussen.

```
<osm version="0.6">
    <bound box="-90,-180,90,180" />
    <node id="270387" lat="48.482063" lon="9.367616" >
         <tag k="name" v="Wasserfall Bad Urach" />
    </node>
    <way id="99947113">
        <nd ref="1155236845" />
        <nd ref="1155223179" />
        <nd ref="1155236845" />
        <tag k="building" v="yes" />
        <tag k="wall" v="no" />
    </way>
    <relation id="1430044">
        <member type="way" ref="27549584" role="street"/>
        <member type="node" ref="1155672139" role="house"/>
        <member type="node" ref="1155672022" role="house"/>
        <member type="node" ref="1155672148" role="house"/>
        <tag k="name" v="Rue Geoffroy-Drouet" />
        <tag k="type" v="associatedStreet" />
    </relation>
</osm>
```

2.3 Verarbeitung

Ziel ist ein Programm, mit welchem Zeichenketten in den Daten gefunden werden können. Die benötigten Informationen müssen zunächst aus den OpenStreetMap-Dateien extrahiert werden. Hierzu wurde ein Programm geschrieben, um das XML-Dateiformat nutzen zu können. Theoretisch wäre es möglich, ein plug-in für `osmosis` [osm], ein in Java implementierter OpenStreetMap-Parser, zu schreiben. Da osmosis nach [osm] sehr viel Arbeitsspeicher benötigt und Interesse an der Implementierung bestand, wurde ein XML-Parser in C++ mit Hilfe der Apache Bibliothek xerces-c geschrieben. Um möglichst wenige zusätzliche

Bibliotheken zu nutzen, wurde lediglich die Standard Template Library (STL) von C++ verwendet. Für die unterschiedlichen XML-Elemente können Filter definiert werden. So können z.B. nur Wege gespeichert werden. Da Wege, Relationen und Knoten unsortiert vorliegen können, müssen die Dateien mehrfach geparst werden, sofern nicht sämtliche Knoten, Wege und Relationen im Speicher abgelegt werden sollen, auch wenn sie später gar nicht benötigt werden. Um den aktuellen Deutschlandausschnitt der Geofabrik zu parsen benötigt das Programm mit 3 Durchläufen auf einem Core i7 930 mit 2.93 Ghz ca. 90 Minuten. Der Festplattendurchsatz stellte dabei nicht den Flaschenhals dar, da die ca. 15 GB große Datei mit nur rund 7 Megabyte pro Sekunde geparst wurde.

3 Datenstrukturen

Um die extrahierten Daten schnell durchsuchen zu können, bedarf es einer angepassten Datenstruktur. Die in dieser Arbeit untersuchten Datenstrukturen sollen in diesem Abschnitt vorgestellt werden.

3.1 Präfix-Baum

Zunächst wurde ein Präfixbaum implementiert. Dieser war eine vollständig im Speicher abgelegte Datenstruktur. Dabei stellte sich heraus, dass viele Knoten nur ein Kind besitzen und somit die Zeiger eines Knotens überproportional viel Speicher belegen. Knoten mit nur einem Kind können jedoch mit ihrem Kind zusammengefasst werden. Die Metadaten wie Punkte oder Straßen werden im Knoten in einer Liste gespeichert. Abbildung 3.1 zeigt einen Baum mit den Worten Java, Rad, Rand, Rau, Raum, und Rose.

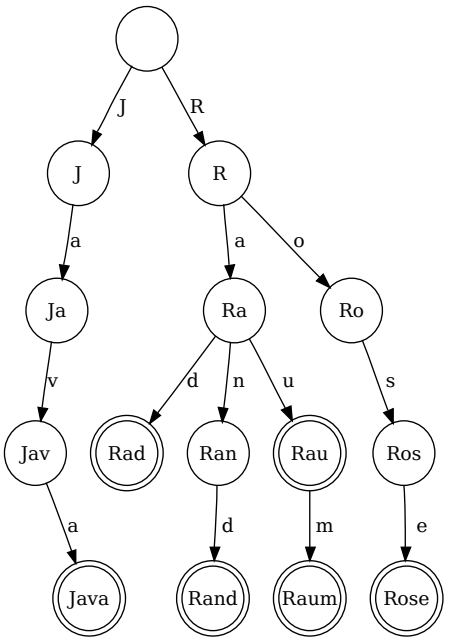

Abbildung 3.1: Beispielhafter Präfixbaum nach [pre]. Zweifach umrundete Knoten sind Zeichenkettenenden

3.2 Patricia-Trie

Der Patricia-Trie, kurz für *Practical Algorithm to Retrieve Information Coded in Alphanumeric*, auch Radix-Trie genannt, unterscheidet sich vom Präfix-Baum nur darin, dass Knoten mit nur einem Kind zusammengefasst werden. Ein Knoten stellt somit nicht nur ein Zeichen, sondern eine Zeichenkette dar. Siehe Abbildung 3.2

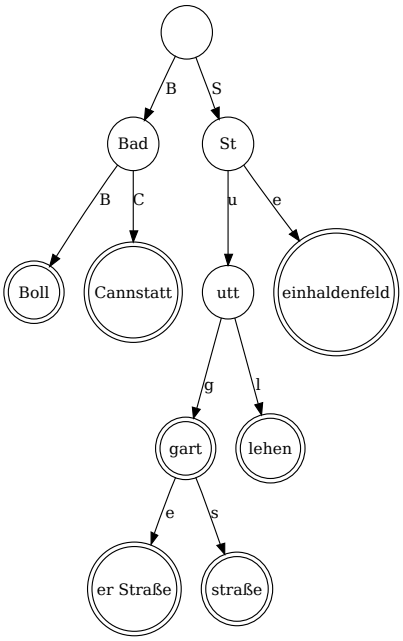

Abbildung 3.2: Beispielhafter Patricia-Trie. Zweifach umrundete Knoten sind Zeichenkettenenden

3.3 Statischer Patricia-Trie

Der Patricia-Trie ermöglicht neben der Suche nach Präfixen auch das Einfügen von neuen Daten. Letzteres ist jedoch auf dem Mobilgerät nicht nötig. Daher kann auch eine Datenstruktur genutzt werden, die kein Einfügen erlaubt, dafür der Look-Up jedoch schneller ist. Hierzu wird der Patricia-Trie in inorder Reihenfolge in einem `uint8_t` Feld abgespeichert. Kindknoten werden in sortierter Reihenfolge abgelegt. Dies ermöglicht die Verwendung einer binären Suche. Dieser Array kann als Datei abgespeichert werden und anschließend per `mmap` in den Arbeitsspeicher eingeblendet werden. Hierdurch erhält man ein sehr effizientes Caching, da das Betriebssystem des Mobilgerätes die Verwaltung übernimmt. Der Overhead einer eigenen Cache-Implementierung entfällt. Speichert man sämtliche Name-Tags des Deutschlandauschnittes in einer Textdatei ab, so hat diese eine Größe von 31 MiB. Der statische Patricia-Trie benötigt für die gleichen Daten ohne jede Form von Kompression lediglich ca. 19 MiB. Zeiger auf eventuelle Metadaten sind hierbei bereits enthalten.

3.4 Statischer Patricia-Trie mit Metadatenspeicherung in einem invertierten Index

Um alle Komplettierungszeichenketten zu erhalten, muss der statische Patricia-Trie in inorder Reihenfolge durchlaufen werden. Eine Komplettierungszeichenkette ist dabei eine Zeichenkette, die die Eingabezeichenkette als Präfix hat. Ein zufälliger Zugriff auf eine Komplettierungszeichenkette ist nicht möglich. Um dieses Problem zu umgehen, werden die Metadaten bzw. die Zeiger zu diesen nicht mehr im Baum abgelegt. Stattdessen gibt es eine weitere Datenstruktur, den Index, welche im Prinzip ein Feld aus Zeigern auf die Metadaten ist. Diese sind in inorder-Reihenfolge abgespeichert. Im Knoten werden nun die Zeiger auf die beiden Elemente im Index gespeichert, die die Zeiger auf die erste und letzte Komplettierungszeichenkette enthalten. Die Komplettierungszeichenketten werden mit ihren Metadaten in einer extra Datenstruktur serialisiert abgespeichert. Eine Komplettierung benötigt somit im schlechtesten Fall so viele Schritte, wie die Präfixzeichenkette lang ist. Durch die doppelte Verzeigerung ist zudem ein zufälliger Zugriff auf die Metadaten möglich. Hierdurch können bei entsprechender Programmoberfläche nur jene Daten geladen werden, die gerade zur Anzeige benötigt werden. Verwendet man `mmapp` um die Metadatendatei zu lesen, sorgt die serielle Anordnung zudem für eine hohe Cache-Effizienz, da Linux die Datei in Blöcken einliest und in den Hauptspeicher einblendet.

3.4.1 Algorithmus

Groß/Klein-Schreibung beachtend

Eingabe Präfixzeichenkette PZ, Wurzelknoten → **Start(**PZ, Wurzelknoten**)**

Start Prüfe ob PZ und Knoten gemeinsames Präfix haben

- Falls PZ Präfix der Knotenzeichenkette, gehe zu **Index**
- Falls Knotenzeichenkette Präfix von PZ, gehe zu **Nächster Knoten**
- Falls PZ und Knotenzeichenkette keinen Präfix haben, gehe zu **Fehler**

Nächster Knoten Finde nächsten Knoten

1. Entferne Knotenzeichenkette von PZ
2. Finde Knoten, der mit PZ[0] beginnt
   - Falls Kind existiert, gehe zu **Start(**PZ, Kindknoten**)**
   - Falls kein Kind existiert: gehe zu **Fehler**

Index Aktueller Knoten enthält Start- und Endzeiger in den Index

Fehler PZ ist kein Präfix einer Zeichenkette des Baumes

Groß/Klein-Schreibung nicht beachtend

Eine Möglichkeit, die Groß/Klein-Schreibung nicht zu beachten, wäre den gesamten Trie nur mit Klein- oder Großbuchstaben zu füllen. Eine Groß/Klein-Schreibung-sensitive Suche wäre dann aber nicht mehr möglich. Bei allen getesteten Daten ist eine rekursive Suche mit keiner großen Laufzeitvergrößerung verbunden. Daher wurde zunächst diese Variante implementiert, wodurch derselbe Trie für beide Sucharten genutzt werden kann.

Eingabe Präfixzeichenkette PZ, Wurzelknoten $\rightarrow$ **Start(**PZ, Wurzelknoten**)**

Ausgabe Liste von (Start, Ende)-Tupeln

Start Prüfe case-insensitiv ob PZ und Knoten gemeinsames Präfix

- Falls PZ Präfix der Knotenzeichenkette, gehe zu **Index**
- Falls Knotenzeichenkette Präfix von PZ, gehe zu **Nächster Knoten**
- Falls PZ und Knotenzeichenkette kein Präfix haben, gehe zu **Fehler**

Nächster Knoten Finde nächsten Knoten

1. Entferne Knotenzeichenkette von PZ
2. Finde Knoten, der case-insensitiv mit PZ$[0]$ beginnt
   - Falls Kinder existieren, gehe für jedes Kind zu **Start(**PZ, Kindknoten**)**
   - Falls keine Kinder existieren: gehe zu **Fehler**

Index Aktueller Knoten enthält Start- und Endzeiger in den Index, füge diese zur Ausgabeliste hinzu

Fehler PZ ist kein Präfix einer Zeichenkette des aktuellen Unterbaumes. Füge nichts zur Ausgabeliste hinzu.

3.4.2 Serialisierung für OpenStreetMap

In diesem Abschnitt soll die verwendete Serialisierung besprochen werden. Zunächst wird die Serialisierung der Baumdatei vorgestellt. Der Header ist wie folgt aufgebaut:

Semantik	v	t	LS	D	SC	NC	CRCTRIE	SUM	T*
Bytezahl	1	1	2	2	4	4	4	12	*

Abkürzung	Erklärung
v	Versionsnummer
t	Baumtyp
T	Serialisierter Baum
LS	Länge des längsten Strings
D	Tiefe des Baumes
NC	Anzahl der Knoten im Baum
SC	Anzahl unterschiedlicher Zeichenketten im Baum
SUM	Prüfsummen

Die Knoten werden in inorder-Reihenfolge in folgender Form abgelegt. Die Zeichenketten werden in UTF-8-Kodierung abgelegt. Die vorgestellte Serialisierung beschränkt sich dabei auf Zeichen der im Unicode-Standard definierten Basic Multilingual Plane (BMP) [TA11]. Der Unicode-Standard definiert unter anderem die Abbildung zwischen Schriftzeichen und ihrer Kodierung für den Computer. Die BMP beinhaltet die Zeichen für nahezu alle modernen Sprachen und wird in die Zahlen von `0x0000` bis `0xFFFF` abgebildet.

Semantik	a	l	L	L	SL	S	C	P	ST	EN
Bitzahl	1	1	6	8	8	SL	(8-16)*L	32*L	32	32
Bytezahl	1			1	1	SL	(1-2)*L	4*L	4	4

Abkürzung	Erklärung
a	Bit, das die Größe eines Zeichens angibt (0 $\Rightarrow$ 1 byte, 1 $\Rightarrow$ 2 bytes)
l	Bit, das angibt, ob ein weiteres Längen-Byte folgt
L	6 Bits, die die Länge des Zeichenarrays angeben
SL	Länge der Zeichenkette des Knotens
S	Zeichenkette
C	Zeichen/Buchstabe enkodiert als Unicode-Codepoint
P	Offset innerhalb des Baumes zum Kindknoten des entsprechenden Zeichens
ST	Start-Element für eine Komplettierung
EN	End-Element für eine Komplettierung

Die Index-Datei besteht lediglich aus 32bit-Integer serialisiert in Netzwerk-Byte-Reihenfolge. Diese stellen den Offset innerhalb der Metadaten-Datei dar. Die OpenStreetMap-Metadaten werden aktuell in folgendem Format abgelegt:

Semantik	L	S	ELMCOUNT	T	C	LoC	LT	LAT/LON
Bitzahl	8	*	(8-32)	2	1	5	8	2*24
Bytezahl	1	*	(1-4)	1			1	2*3

Abkürzung	Erklärung
L	Länge des Strings
S	String enkodiert als UTF-8
ELMCOUNT	Anzahl der Geoelemente (variable Länge)
T	Daten-Typ (Knoten, Weg, Relation)
C	Gibt an, ob ein LT-Byte folgt
LoC	Länge der Anzahl der Positionen
LAT	Breitengrad enkodiert als 24bit-Integer
LON	Längegrad enkodiert als 24bit-Integer

Abbildung 3.4 zeigt einen Beispielbaum mit allen vorgestellten Datenstrukturen.

3.5 Minimale perfekte Hash-Funktion

Als Alternative zur Verwendung eines Patricia-Tries böte sich noch die Verwendung einer Hash-Datenstruktur an, in der alle möglichen Präfixe gehasht werden. Für die Kombination der Ausschnitte von Deutschland und Großbritannienauschnitt der Geofabrik wären dies 11317291 unterschiedliche Präfixzeichenketten. Da eine solche Hashfunktion nur für die Untermenge der Präfixzeichenketten eindeutig ist, alle anderen Zeichenketten jedoch ebenfalls in die Bildmenge abgebildet werden, müssen Zusatzinformationen gespeichert werden, um falsch-positive Werte zu erkennen. Hierzu kann auch die Metadaten-Datei des statischen Tries genutzt werden. Die Hash-Funktion bildet die Präfxzeichenketten-Menge in die natürlichen Zahlen von 0 bis zur Größe der Eingabe-Menge ab. Diese Bildmenge muss nun noch auf die Menge der Zeichenketten abgebildet werden. Für jedes Präfix liefert die Hashfunktion einen Hashwert. Da nicht alle Präfixe unterschiedliche Komplettierungsmengen liefern, lassen sich gleiche Komplettierungsmengen zusammenfassen. Dies kann man mit einer einfachen Tabelle, dem ID-Index erreichen. Dieser bildet den Hashwert in eine weitere Tabelle, den Präfixindex ab. In diesem stehen die Start- und Endpositionen im Metadaten-Index des Patricia-Tries. Siehe Abbildung 3.3 für eine Visualisierung.

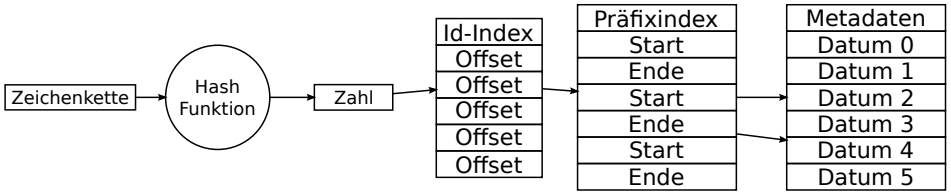

Abbildung 3.3: Datenstrukturen bei Verwendung einer minimalen Hashfunktion.

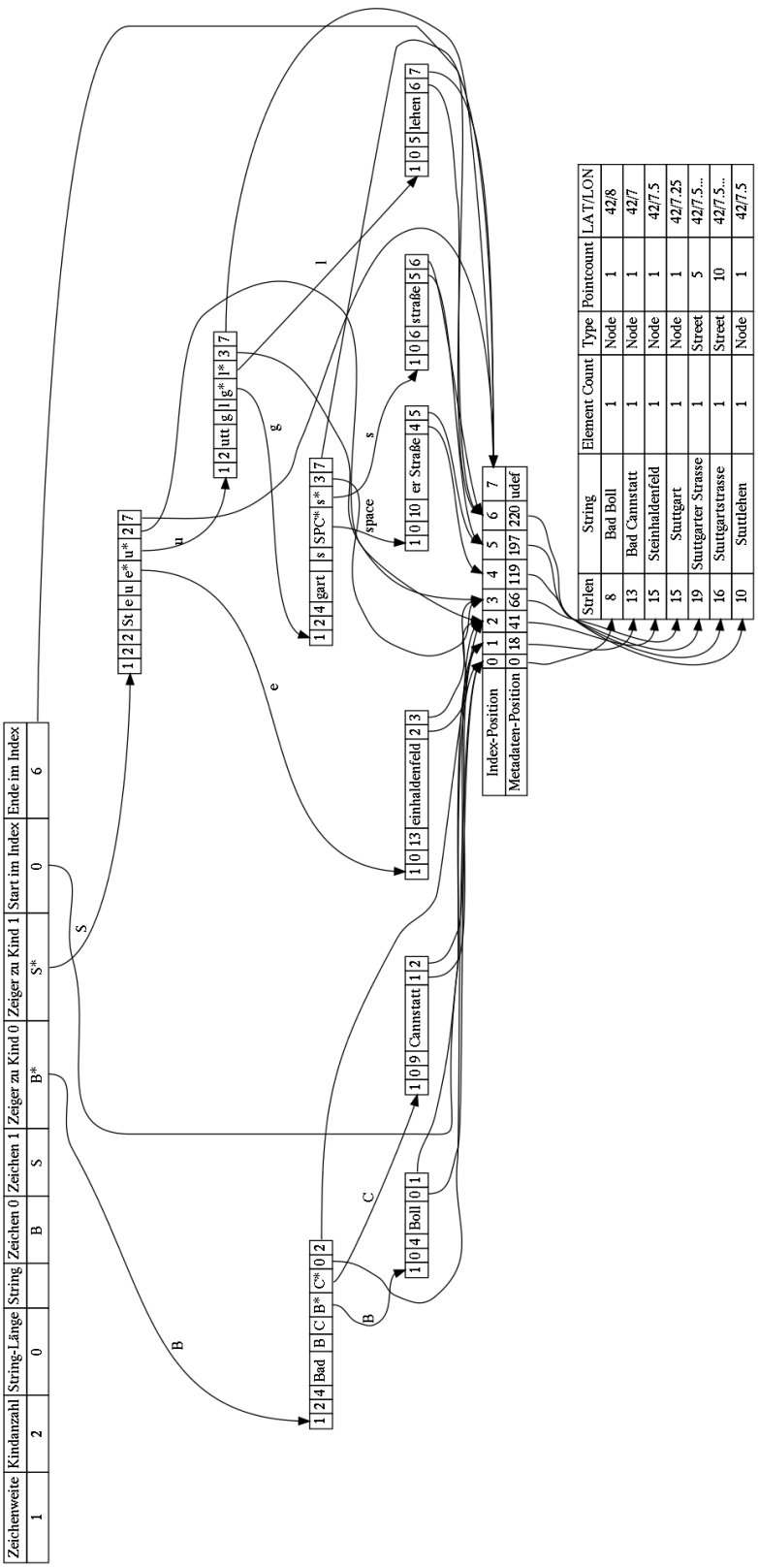

Abbildung 3.4: Beispielhafter statischer Patricia-Trie mit Index und Metadaten.

4 Implementierung

In diesem Abschnitt soll ein kurzer Überblick über die Implementierung der oben beschriebenen Datenstrukturen gegeben werden. Zunächst soll der Teil beschrieben werden, der zur Erzeugung des statischen Patricia-Trie genutzt wird und anschließend wird die Android-Applikation erläutert.

osmfind-create

Um die OpenStreetMap-Daten nutzen zu können, wurde zunächst ein XML-Parser geschrieben, welcher in Kapitel 2 vorgestellt wurde. Dieser wird genutzt, um die benötigten Informationen zu extrahieren. Aktuell werden nur diejenigen Openstreet-Map-Elemente genutzt, die einen name-Tag besitzen. Der name-Tag wird mit seinen Metadaten (Element-Typ und Geopunkte) in den Patricia-Trie eingefügt. Anschließend wird dieser serialisiert. Die Serialisierung selbst dauert für den Großbritannien-Deutschlanddatensatz auf einem Core i7 930 mit 12 GB Ram nur wenige Sekunden. Das mehrfache Einlesen der XML-Datei beansprucht dagegen 90 Minuten.

create-cmphcompletion

Der serialisierte Patricia-Trie kann dazu verwendet werden, eine minimale perfekte Hashfunktion zu erzeugen. Zusätzlich werden auch die benötigten Look-Up-Tabellen erzeugt. Hierzu werden alle möglichen Präfixe als Urbildmenge der Hashfunktion erzeugt. Anschließend müssen die Hashwerte den korrekten Knoten des Baumes und somit den entsprechenden Komplettierungsmengen zugeordnet werden. Diese Informationen werden in 2 Dateien serialisiert in Netzwerkbyteordnung gespeichert. Die Hash-Funktionen stellt dabei die Bibliothek `cmph` von [cmp] bereit. Diese ermöglicht es zu einer gegebenen Menge von Zeichenketten eine minimale perfekte Hash-Funktion zu erzeugen.

osmfind

Das Ziel der Arbeit ist die effiziente Textsuche auf mobilen Geräten. Als Plattform wurde Android gewählt, da es eine hohe Nutzerbasis hat und Android-Geräte zur Verfügung standen. Zunächst wurde ein Konsolenprogramm in C++ geschrieben, um die Datenstruktur lokal testen zu können. Um das Programm auf einem Android-Gerät ausführen zu können,

muss es lediglich entsprechend übersetzt werden. Da eine Konsolenapplikation nicht angenehm zu bedienen ist und zudem nicht offiziell von Android unterstützt wird, wurde noch eine grafische Oberfläche als Android-Applikation in Java implementiert. Diese kann mit Hilfe des Java Native Interface die C++-Implementierung nutzen. Zusätzlich wurde die Suche auch in reinem Java-Code implementiert. Hierdurch konnten Informationen über den Geschwindigkeitsunterschied zwischen nativer Ausführung und Ausführung mit dem Java-Interpreter gewonnen werden. Durch die Nutzung eines Tries ist es trivial, zu einer gegebenen Komplettierungszeichenkette, mögliche Fortsetzungen zu erhalten. Diese werden in der GUI angezeigt, sodass eine Suche sowohl über die Eingabe der Zeichenkette über die Tastatur als auch über Buttons, die die nächsten Buchstaben anzeigen, möglich ist. Die gefundenen Geodaten können mit `mapsforge`, einer Opensource-Kartendarstellungsbibliothek, visualisiert werden. In der Kartendarstellung kann dann durch einen Klick auf ein Geodatum z.B. ein Navigationsprogramm gestartet werden.

5 Android

Android ist ein Betriebssystem, das von Google entwickelt wird. Im Kern basiert es auf einem modifizierten Linux-Kernel. Android wird derzeit hauptsächlich auf Mobiltelefonen und Tablets eingesetzt. Eine Umsetzung für die x86-Architektur befindet sich derzeit in Entwicklung. Anwendungsprogramme werden bis auf wenige Ausnahmen in Java implementiert und auf einer von Google programmierten Java Virtual Machine ausgeführt. Für Programme, die Zugriff auf native Bibliotheken oder die höhere Geschwindigkeit einer nativen Ausführung benötigen, kann das Java Native Interface genutzt werden. Dieses stellt eine Schnittstelle zwischen Java und C-Code her. Falls das Gerät root-Zugriff ermöglicht, können auch beliebige Linux-Programme, die für die jeweils eingesetzte Architektur kompiliert wurden, installiert werden. Dieser Weg wird von Google jedoch nicht offiziell unterstützt. Rudimentäre Linux-Kenntnisse sind daher hilfreich.

5.1 Cross-Kompilierung

In diesem Abschnitt soll beschrieben werden, wie Linux-Programme mit Hilfe von `cmake`, für andere Architekturen, kompiliert werden können.

5.1.1 Gentoo

Falls die Linux-Distribution Gentoo zur Verfügung steht, lässt sich leicht eine Toolchain zur Cross-Kompilierung installieren:

```
crossdev arm-none-linux-gnueabi
```

Falls noch weitere Bibliotheken installiert werden müssen, kann dies mit `xmerge`, einem `emerge` Wrapper geschehen. Siehe hierzu die Gentoo-Dokumentation [AT]. Anschliessend lässt sich ein `cmake`-Projekt leicht mit dieser Toolchain installieren:

```
mkdir build && cd build
cmake -DCMAKE_TOOLCHAIN_FILE=../../cmake/toolchain-arm-none-linux-gnueabi.cmake ../
make
```

Die Toolchain-Datei basiert auf [cmaa] und [cmab]. Falls das Programm Bibliotheken enthält, die Google nicht anbietet, müssen diese möglicherweise statisch mit einkompiliert werden. Das Programm kann nun auf das Handy kopiert werden. Um es auszuführen, muss es jedoch noch auf die entsprechende Partition kopiert werden, wofür möglicherweise root-Rechte benötigt werden.

```
adb push programm /sdcard
adb shell
su
busybox ash
busybox mount -o rw,remount /system
cp /sdcard/programm /system/usr/
busybox mount -o ro,remount /system
cd /system/usr/
./programm
```

5.1.2 Android-ndk

Falls Gentoo-Linux nicht zur Verfügung steht, kann auch eine vorkompilierte Toolchain von Google genutzt werden. Es handelt sich dabei um das Native Development Kit, kurz ndk. Dieses kann auf der entsprechenden Internetseite von Google [andb] heruntergeladen werden. Nun muss es noch als stand-alone Variante installiert werden: [anda]

```
export NDK=/pfad/zum/ndk/ordner/
export ANDROID_NDK_TOOLCHAIN_ROOT=/pfad/zur/toolchain/android-5/
export ANDROID_NDK_DESTINATION_DIR=/pfad/zum/ziel/ordner/
$NDK/build/tools/make-standalone-toolchain.sh --platform=android-5 \
    --install-dir=$ANDROID_NDK_DESTINATION_DIR
```

Anschliessend lässt sich dieses mit einer weiteren Toolchain-Datei von [anda] nutzen.

```
export ANDROID_NDK_TOOLCHAIN_ROOT=$HOME/android-toolchainfile
mkdir build && cd build
cmake -DCMAKE_TOOLCHAIN_FILE=../../cmake/android.toolchain.cmake ../
make
```

5.2 Android-Applikationsentwicklung

Zunächst sollte das Android Software Development Kit von [andc] heruntergeladen und installiert werden. Falls möglich, erfolgt dies über die Softwareverwaltung der eingesetzten Distribution. Das SDK stellt auch eine virtuelle Mobiltelefonumgebung bereit, mit welcher Programme getestet und debuggt werden können. Weitere Informationen hierzu findet man in [andd].

5.3 Eclipse

Eclipse ist eine integrierte Entwicklungsumgebung für Java und weitere Sprachen. Google bietet hierfür ein plug-in zur einfachen Verwaltung von Android-Projekten an. Eine Installationsanleitung findet sich unter [ecl].

5.4 Programmierung

Offiziell können Android-Applikationen nur in Java entwickelt werden. Es ist aber auch möglich nativen Code auszuführen. Dieser kann sowohl unabhängig von der Java Virtual Machine als eigenständiges natives Programm als auch als native Bibliothek innerhalb einer Java-basierten Applikation ausgeführt werden. Ein eigenständig laufendes Programm kann, wie im Abschnitt 5.1 beschrieben, für Android kompiliert werden. Im Folgenden soll dagegen auf die Möglichkeit der Einbettung einer nativen Bibliothek in Java eingegangen werden.

5.4.1 Java Native Interface

Java bietet von Haus aus die Möglichkeit, nativen Code auszuführen. Hierzu wurde das Java Native Interface (JNI) definiert. Dieses stellt eine Verbindung zwischen Java-Code und C/C++ bereit. Java-Applikationen können so Funktionen einer nativen Bibliothek aufrufen. Im Gegenzug kann im C/C++-Code auch auf Funktionen der Java-VM zugegiffen werden. Im Folgenden soll auf die wichtigsten Punkte eingegangen werden.

Es wird davon ausgegangen, dass eine in C++ geschriebene Klasse in Java genutzt werden soll. Zunächst wird aus einer Java-Klasse ein C-Wrapper erzeugt, der die Verbindung zur C++-Klasse bereit stellt. Um aus einer Java-Klasse den Wrapper zu erzeugen, müssen die Funktionen, die in C implementiert werden sollen, mit dem Schlüsselwort **native** gekennzeichnet werden.

```
public class Example {
    public native int function(int value);
    static {
        System.loadLibrary("libname");
    }
}
```

Anschliessend muss der Java-Code in Java-Byte-Code übersetzt werden:

```
javac Example.java
```

Nun muss noch eine Header-Datei für die native Programmiersprache erstellt werden.

```
javah Example
```

Dabei erstellt `javah` eine Datei namens `Example.h`, welche C-Funktionen enthält, die implementiert werden müssen. Falls der Paketname geändert wird, muss der Header entweder neu erstellt oder entsprechend angepasst werden.

jniglobalvar

Die einfachste Variante das Java Native Interface zu nutzen ist `jniglobalvar`. Dies bietet sich vor allem dann an, wenn entweder nur statische Funktionen oder aber nur eine Instanz der C++-Klasse, genutzt werden soll.

jnipointer

Falls von einer C++-Klasse mehrere Instanzen benötigt werden, die in Java ebenfalls mehrere Instanzen darstellen, so lässt sich dies darüber lösen, dass jeder nativen Funktion der Zeiger zur Instanz übergeben wird:

```
public class Example {
    public native int function(long pointer, int value);
}
```

Im C/C++-Teil muss dieser nun noch entsprechend konvertiert werden:

```
jint Example_function(jlong pointer, jint value) {
    CppKlasse * cppKlasse = reinterpret_cast<CppKlasse*>(pointer);
}
```

jnicheckedpointer

Da eine einfache Konvertierung eines Zeigers in einen Java-Long und wieder zurück zu Problemen führen kann, ist es ratsamer, eine extra Datenstruktur zu nutzen, um aus einer Zahl die zugehörige Klasseninstanz zu bekommen. Hierzu bietet sich etwa ein Hash an.

```
jint Example_function(jlong pointer, jint value) {
    CppKlasse * cppKlasse = hash_lookup(pointer);
}
```

5.4.2 SWIG

SWIG ist ein Akronym für Simplified Wrapper and Interface Generator und eine Möglichkeit um C/C++ Funktionen und Klassen aus anderen Sprachen heraus einfach anzusprechen. SWIG erzeugt dabei mit Hilfe einer Interface-Definition selbstständig die entsprechenden Java- und C-Wrapper-Funktionen. Leider implementiert Androids Java Virtual Machine noch nicht alle benötigten Features. Dies soll sich in Zukunft ändern, wodurch die Entwicklung von Java-Applikationen, die Zugriff auf nativen Code benötigen, wesentlich vereinfacht werden wird.

6 Statistiken

6.1 Datensätze

Da die Erzeugung des Patricia-Tries derzeit noch relativ viel Arbeitsspeicher benötigt, war es leider nicht möglich einen sehr großen Datensatz zur Ermittelung der Geschwindigkeit auf Mobilgeräten zu nutzen. Der größte erzeugbare Baum basierte auf dem Deutschland- und Großbritannienausschnitt der Geofabrik. Dieser beinhaltete zum Testzeitpunkt ca 1,5 Millionen unterschiedliche Zeichenketten mit ca. 4,6 Millionen Geoelementen und 30 Millionen Geopunkten. Der Baum besteht aus 2 Millionen Knoten mit 11 Millionen UTF-8-Zeichen. Würde man alle Zeichenketten komplett abspeichern, wären dies 24 Millionen Zeichen. Da der Baum nicht nur Zeichenketten speichert, erreicht er eine Gesamtgröße von ca. 41 MiB. Für den Hash müssen 11 Millionen unterschiedliche Präfixstrings gespeichert werden. Der Id-Index ist daher ca. 44 MiB groß, der Präfix-Index dagegen hat eine Größe von 14 MiB. Die Hash-Funktion selbst ist unterschiedlich groß. `cmph` erzeugt mit dem CHD-Algorithmus [BBD09] eine 5,4 Megabyte große Datei. Neben diesem wurden noch weitere Datensätze ausgewertet. Diese sind in Tabelle 6.1 dargestellt.

6.2 Benchmark

Der zuvor vorgestellte Datensatz wurde auf einem Motorola Defy mit einer 800 Mhz CPU mit 512 MiB Arbeitsspeicher getestet. Hierzu wurden zufällige Zeichenketten mit einer mittleren Länge von 10 Zeichen im Baum komplettiert. Die vorgegebene Zeichenkettenlänge folgte einer Gleichverteilung. Die Zeichenketten entstanden, indem zufällig ein Kind des aktuellen Knotens ausgewählt wurde. Falls die aktuelle Zeichenkette innerhalb einer Knotenzeichenkette endete, wurde diese bis zum Erreichen der maximalen Zeichenkettenlänge fortgesetzt. Es wurden 100 Dateien zu je 1000 Zeichenketten erstellt. Tabelle 6.2 gibt die mittlere Komplettierungsdauer für einige Datensätze an. Die Baum-basierte Implementierung erwies sich als schnell genug um vom Benutzer als Echtzeit-Anwendung wahrgenommen zu werden.

	DE+GB	DE	GB	NL	I	JP	AT	CH	B
Unterschiedliche Zeichenketten	1.494.190	890.985	607.200	176.399	273.151	292.054	151.650	120.843	91.886
Geoelemente	4.586.232	2.944.186	1.642.046	1.034.489	690.572	889.504	390.875	263.580	219.148
Geopunkte	30.036.223	19.038.872	1.099.7351	4.991.874	7.858.616	16.557.800	4.354.008	2.056.845	1.698.008
Knoten	2.043.559	1.212.876	840.061	242.503	386.418	354.381	206.509	166.503	128.254
UTF-8 Zeichen im Baum	11.545.079	7.071.257	4.559.811	1.195.945	1.973.558	4.140.602	1.201.306	908.328	783.531
UTF-8 Zeichen im Metadatenindex	24.634.360	1.4867.871	9.794.314	2.444.102	4.638.446	7.004.302	2.334.612	1.842.893	1.470.738
Größe der Baumdatei in KB	41.220	24.680	16.760	4.724	7.592	9.572	4.200	3.328	2.648
Größe der Indexdatei in KB	5.840	3.484	2.372	692	1.068	1.144	596	476	360

Tabelle 6.1: Statistische Daten einiger Länderdatensätze

			DE+GB	DE	GB	NL	I	JP	AT	CH	B
Case sensitive	Trie	∅	112	95	88	75	72	191	73	67	69
		σ	148.9	99.7	77.2	48.2	36.9	97.5	40.0	29.2	27.2
	mphf	∅	1557	1604	301	106	130	291	101	85	80
		σ	891.1	727.3	629.1	323.8	401.5	617.9	280.0	181.4	150.4
Case insensitive	Trie	∅	202	179	197	160	161	136	169	167	161
		σ	209.1	137.4	231.2	9.5	10.4	10.6	15.9	14.8	10.2
	mphf	∅	14291	22873	6085	3580	7702	239	4306	3181	5408
		σ	7116.6	18092.4	6381.1	1719.3	2337.5	447.8	2129.7	4159.4	12841.0

Tabelle 6.2: Durchschnittliche Komplettierungszeiten in Mikrosekunden, getestet mit 100.000 Wörter

7 Ausblick

7.1 Multidimensionale Suche

Eine interessante Erweiterung wären Schnittoperationen für mehrere unterschiedliche Eingabezeichenketten. Ein OpenStreetMap-Datum beinhaltet dabei mehrere Zeichenketten. Eine Speicherung und Suche nach Points-of-Interests wäre ebenfalls möglich. So würde beispielsweise eine Suche nach (Bäcker, Stuttgart, Vaihingen) alle Bäcker in Stuttgart-Vaihingen liefern. Eine mögliche Datenstruktur soll nun kurz umrissen werden.

Die Semantik des Patricia-Trie ändert sich dabei nicht grundlegend, lediglich die Zeiger in den Index ändern sich. Dieser beinhaltet nun für jeden Knoten sämtliche OpenStreetMap-Daten, die die Knotenpräfixzeichenkette als Präfix haben. Durch die Serialisierung der Metadaten bekommt jedes Datum einen eindeutigen Offset innerhalb der Metadaten-Datei. Die Offsets zu den Metadaten des Knotens werden in aufsteigender Reihenfolge sortiert abgelegt. Die Schnittoperation kann dann mit dem Merge-Alogrithmus in linearer Zeit durchgeführt werden. Eine andere Möglichkeit wäre die rekursive Nutzung von Patricia-Tries, indem zu jedem Knoten ein weiterer Patricia-Trie gespeichert wird, der für jedes Datum die restlichen Zeichenketten enthält, für die die aktuelle Komplettierungszeichenkette kein Präfix darstellt.

Da viele Metadaten gleiche Zeichenketten enthalten, bietet sich eine komprimierte Speicherung dieser Zeichenketten an. Hierzu könnten die Zeichenketten direkt in der Metadaten-Datei komprimiert abgelegt werden. Eine andere Möglichkeit wäre die Speicherung in einer getrennten Datei.

7.1.1 Teilzeichenketten

Die Suche nach Teilzeichenketten kann ähnlich wie in einem Suffix-Baum erfolgen. In den Patricia-Trie werden für jedes OpenStreetMap-Datum hierzu alle möglichen Suffixe eingefügt.

7.2 Geometrische Einschränkungen

Neben einer Suche nach Metadaten, die bestimmte Zeichenketten enthalten, wäre es wünschenswert, alle Metadaten zu einem festlegbaren Gebiet zu erhalten. Die Suche nach Zeichenketten sollte dann nur Metadaten innerhalb des vorgegebenen Gebietes liefern.

Literaturverzeichnis

[anda] Android-CMake Projekt. URL http://code.google.com/p/android-cmake/. (Zitiert auf Seite 22)

[andb] Android Native Development Kit. URL http://developer.android.com/sdk/ndk/index.html. (Zitiert auf Seite 22)

[andc] Android Software Development Kit. URL http://developer.android.com/sdk/index.html. (Zitiert auf Seite 22)

[andd] Android Virtual Devices. URL http://developer.android.com/guide/developing/devices/index.html. (Zitiert auf Seite 22)

[AT] M. F. Alex Tarkovsky. Gentoo Cross Development Guide. URL http://www.gentoo.org/proj/en/base/embedded/cross-development.xml. (Zitiert auf Seite 21)

[BBD09] D. Belazzougui, F. C. Botelho, M. Dietzfelbinger. Hash, Displace, and Compress. In *ESA'09*, pp. 682–693. 2009. (Zitiert auf Seite 27)

[cmaa] CMake Dokumentation. URL http://www.cmake.org/cmake/help/documentation.html. (Zitiert auf Seite 22)

[cmab] CMake Toolchain-Datei für Arm-Prozessoren. URL http://www.ros.org/doc/api/eros_toolchains/html/crossdev_2i686-pc-mingw32_8cmake_source.html. (Zitiert auf Seite 22)

[cmp] CMPH - C Minimal Perfect Hashing Library. URL http://cmph.sourceforge.net/. (Zitiert auf Seite 19)

[ecl] Android Development Tools für Eclipse. URL http://developer.android.com/sdk/eclipse-adt.html. (Zitiert auf Seite 23)

[goo] Google Protocol Buffers. URL http://code.google.com/p/protobuf/. (Zitiert auf Seite 7)

[IA00] N. N. Imagery, M. Agency. Department of Defense World Geodetic System 1984. Technical report, 2000. URL http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf. (Zitiert auf Seite 7)

[osm] Osmosis - Kommandozeilenprogramm um OpenStreetMap-Daten zu Verarbeiten. URL http://wiki.openstreetmap.org/wiki/Osmosis. (Zitiert auf Seite 8)

[pre] Bild eines Präfixbaumes. URL http://upload.wikimedia.org/wikipedia/commons/e/e2/Trie.svg. (Zitiert auf den Seiten 4 und 11)

[TA11] The Unicode Consortium, J. Allen. *The Unicode Standard, Version 6.0*. Sixth edition, 2011. URL http://www.unicode.org/versions/Unicode6.0.0/. (Zitiert auf Seite 15)

Alle URLs wurden zuletzt am 30.10.2011 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Daniel Bahrdt)