TECHNISCHE UNIVERSITÄT CAROLO-WILHELMINA ZU BRAUNSCHWEIG

informatiCup 2012

Aufgabe 3: Geo Crosswords

Sandra Hesse, Felix Geilert, Sebastian Morr

15. Januar 2012

Inhaltsverzeichnis

1 Technische Dokumentation

1.1 Vorbemerkungen

Wir haben die dritte Aufgabe des diesjährigen Informaticups gelöst und ein Android-Spiel geschrieben, bei dem man zu (virtuell) in der Stadt verteilten Fragen läuft und so ein Kreuzworträtsel löst.

In diesem Dokument möchten wir einerseits darlegen, auf welchen theoretischen Überlegungen unsere Applikation basiert, andererseits, wie wir diese Ideen umgesetzt haben.

1.2 Theorieteil

Wenn man die im Folgenden beschriebenen Schritte als Ganzes betrachtet, dann ergibt sich ein Algorithmus, der als Input die Koordinatengrenzen des Kartenausschnitts sowie eine Liste von Wörtern besitzt, als Output eine Menge von Wörtern, deren einzelne Buchstaben ebenfalls Koordinaten zugeordnet sind.

Kurze Zusammenfassung unserer Vorgehensweise: OpenStreetMap-Ausschnitt herunterladen, parsen, die Straßen heraussieben, den Graphen verschönern, und schließlich einen genetischen Algorithmus zum Befüllen verwenden.

1.2.1 Von OSM-XML zum 2D-Grid

Warum OpenStreetMap?

Wir haben uns dafür entschieden das Kartenmaterial des OpenStreetMap-Projektes zu verwenden. Diese Daten unterliegen einer Creative Commons-Lizenz, sind außerdem über eine bequeme HTTP-API abrufbar und sehr einfach aufgebaut. Im Gegensatz zu Google Maps sind (auch in Zukunft) keine Nutzungsgebühren zu entrichten. Außerdem ist die Kartenqualität gefühlt sogar höher, insbesondere was den Detailgrad angeht, der für die hier benötigte Fußgängernavigation wichtig ist.

Das Kartenformat

Die API, über die wir die Kartendaten beziehen, erfordert es lediglich, auf folgende URL ein HTTP-GET auszuführen:

```
http://api.osm.org/api/0.6/map?bbox=minLat,minLon,maxLat,maxLon
```

(wobei die min- und max-Werte die Koordinaten eines WGS 84-Gitters bilden). Rückgabe ist eine XML-Datei mit dem Root-Tag `<osm>`, die ein `<bounds>`-Tag mit den durch

uns gegebenen Grenzen des Karten-Ausschnitts, sowie `<node>`-,`<way>`- sowie `<relation>`-Tags enthält.

Ein `<node>` beschreibt einen Punkt auf der Karte, er besteht aus zwei Float-Werten für Längen- und Breitengrad (wiederum im WGS 84).

Ein `<way>` enthält eine Liste von Knoten, aus denen er sich zusammensetzt.

Eine `<relation>` schließlich beschreibt die Beziehungen mehrerer Knoten oder Wege zueinander. Diesen Datentyp benötigen wir jedoch nicht.

Die drei Objekttypen besitzen außerdem alle einen für ihren Typ eindeutige ID, über die sie sich untereinander referenzieren können sowie *Tags*, welche die Objekte mit Eigenschaften versehen. Desweiteren enthalten sie noch Angaben über die Version, Erstelldatum, Sichtbarkeit und Zeitpunkt sowie Autor der letzten Änderung, die uns aber allesamt nicht interessieren (von der API werden nur sichtbare Objekte zurückgegeben).

Datenstrukturen, Definitionen

Ein *Knoten* unseres Kreuzworträtsels ist eine 2D-Koordinate, die eine ganzzahlige ID und einen Buchstaben speichert.

Das *Grid* ist bei uns das „Skelett“ eines Kreuzworträtsels, und im Grunde nur ein ungerichteter Graph, der aus diesen Knoten sowie Kanten besteht.

Vorgehensweise beim Parsen der XML-Datei

Wir iterieren über die Knoten und legen sie unter ihrer ID in einer Hash-Tabelle ab. Anschließend prüfen wir für jeden Weg, ob er begehbar ist (denn auch Flüsse werden beispielsweise als Weg gespeichert), dies lässt sich daran erkennen, dass der Weg einen „highway“-Tag besitzt. Ist er begehbar, fügen wir seine Knoten dem Graphen hinzu, wobei wir darauf achten, dass mindestens einer der beiden Knoten, aus denen sich ein Wegsegment zusammensetzt, innerhalb der geforderten Grenzen liegt.

1.2.2 Bereinigen des Grids

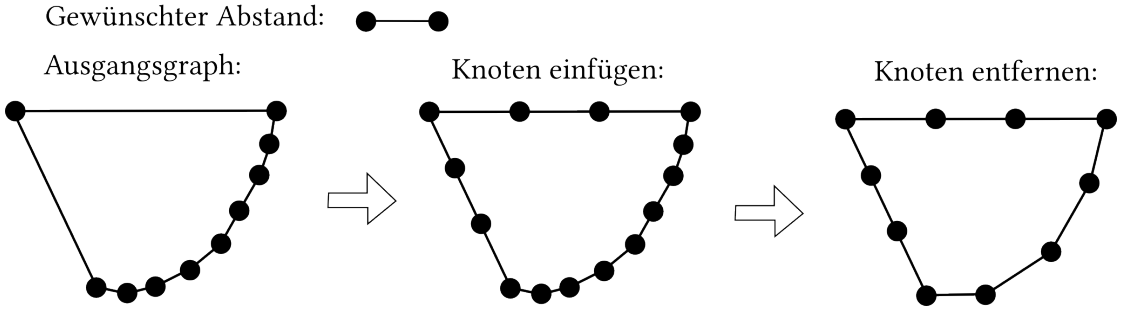
Der so gewonnene Graph bildet die begehbaren Bereiche der Stadt gut ab, ist jedoch noch nicht als Gitter für ein Kreuzworträtsel geeignet, denn Knoten können zu nah aneinander liegen oder zu weit voneinander weg sein. Erstrebenswert sind möglichst gleichmäßige Abstände zwischen den einzelnen verbundenen Knotenpaaren.

Zu große Abstände stellen kein allzu großes Problem dar: Wenn eine Kante eine Länge von l hat und wir einen Knotenabstand von a anstreben, sollten wir die Kante in $s = \text{round}(\frac{l}{a})$ Segmente teilen, das heißt, wir fügen $s - 1$ Knoten in Abständen von jeweils l/s ein.

Zu kleine Abstände sind erfordern mehr Aufwand. Wir laufen hier zunächst über die Knoten und weisen ihnen jeweils einen „Kompaktheitswert“ zu, der sich aus der Summe der inversen Abstände der (nicht notwendigerweise verbundenen) Nachbarn innerhalb eines vorgegebenen Radius' berechnet. Je höher dieser Wert ist, desto enger ist der Knoten von Nachbarn umgeben und desto dringender sollte er entfernt werden. Wir sortieren also die Knoten nach diesem Wert und fangen an, die Knoten mit der höchsten Kompaktheit

zu entfernen. Die (verbundenen) Nachbarn eines zu entfernenden Knotens verbinden wir mit demjenigen Nachbarn, der am nähesten an unserem Knoten lag, damit der Graph nicht zerfällt. Der Kompaktheitswert der umliegenden Knoten wird nun noch entsprechend verringert, da ja ein Knoten weggefallen ist.

Diese beiden Verfahren, hinereinander angewendet ergeben einen halbwegs ästhetischen Graphen. Da es trotzdem vorkommen kann, dass der Graph in mehrere unverbundene Teile zerfällt (durch die Wahl des Ausschnittes, durch Bussteige, die auch den Tag „highway“ besitzen), suchen wir am Schluss noch die größte Zusammenhangskomponente heraus und lassen alles andere weg.

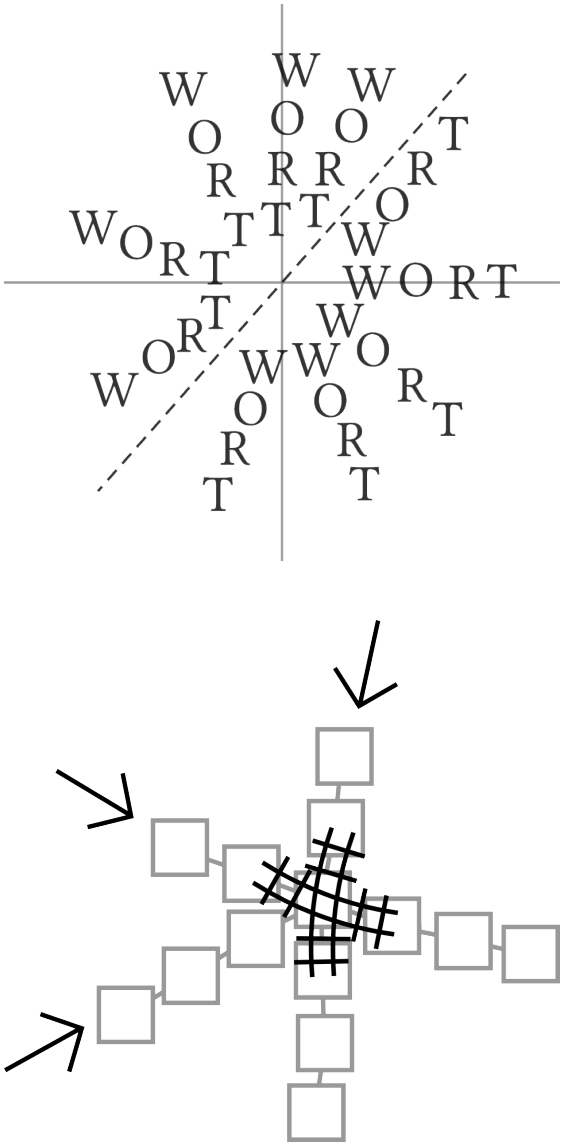


1.2.3 Vom 2D-Grid zum Metagraphen

Unsere Karte ist immer genordet, da dies eine einfachere Orientierung ermöglicht als eine sich ständig mitdrehende Karte. Wir müssen beachten, dass die einzelnen Wörter des Rätsels immer lesbar bleiben. In unserem Kulturkreis sind wir es gewohnt, Buchstaben von links nach rechts, notfalls auch von oben nach unten, aneinanderzureihen. Für jede Kante lässt sich also bestimmen, in welche Richtung sie „lesbar“ ist: Zeigt eine Kante in Richtung des rechten, unteren Halbkreises, kann sie vorwärts gelesen werden, ansonsten rückwärts (diese Aufteilung hat sich auch durch Ausprobieren ergeben und bewährt).

Es ist nun zu bestimmen, welche der Wege, die an Kreuzungen zusammenstoßen, ineinander übergehen dürfen, sprich, auf welcher Kante ein Wort weitergehen soll, das auf einer anderen Kante auf die Kreuzung stößt. Dieses Konzept erinnerte uns so sehr an Weichen, dass wir diesen Begriff im Folgenden verwenden werden als Eigenschaft eines Knotens, einer eintreffende Kante eine ausgehende Kante zuzuordnen.

Um diese Weichen/Turnouts zu bestimmen, iteriert man für jeden Knoten über sämtliche Nachbarn, und prüft jeweils, ob die entstehende Kante eine eingehende ist, also vom Nachbarn in Richtung unseres Knotens gelesen werden kann. Ist das der Fall, iteriert man über die restlichen Nachbarn und sucht denjenigen heraus, der einerseits eine ausgehende Kante ergibt, zweitens noch nicht an einer Weiche beteiligt ist und außerdem die minimale Differenz in der Steigung im Vergleich zur einfallenden Kante bildet. Findet man einen solchen zweiten Nachbarn, fügt man dem Knoten eine neue Weiche hinzu, die den ersten Nachbarn mit dem zweiten verbindet.



1.2.4 Das Befüllen des Metagraphen

Die größte Schwierigkeit beim Befüllen ist die beschränkte Rechenkapazität von mobilen Endgeräten. Deshalb verwenden wir zum generieren des Rätsels (d.h. zum Befüllen des Metagraphen) einen genetischen Algorithmus. Einen Überblick über genetische Algorithmen findet sich hier:

`http://de.wikipedia.org/wiki/Genetische_Algorithmen`

Der Algorithmus durchläuft mehrere Iterationen. Für die Iterationen gibt es mehrere Endbedingungen. Dabei haben wir zwei Ansätze implementiert, wobei sich der Zweite für unseren Fall als performanter erwiesen hat.

Klassischer Ansatz

Der Algorithmus durchläuft eine vorher festgelegte Anzahl an Iterationen oder bricht ab, wenn das Resultat eine bestimmte Qualität erreicht hat. Innerhalb der eigentlichen Iterationen durchläuft der Algorithmus mehrere Phasen mit dem Ziel eine neue Population zu erzeugen:

1. **Selektion**
   Zuerst selektieren wir aus der bisherigen Population. Dazu sortieren wir die bisher erzeugten Graphen nach bestimmten Qualitätskriterien (Anzahl der leeren/blockierenden Knoten) und übernehmen die obere Hälfte. Danach fügen wir einen neuen leeren Graphen ein, der zufällig mit Wörtern aus der Wortliste befüllt wird. Dies sorgt dafür, dass sich der Algorithmus möglichst nicht „festfährt“ (d.h. neue Werte hinzukommen).

2. **Rekombination**
   Hier werden jeweils zwei der bisher übernommenen Graphen gesucht und kombiniert („gemergt“). Bei der Kombination werden alle Worte des Graphens durchlaufen und abwechselnd versucht Worte des ersten und zweiten Elternteils einzufügen. Abschließend werden die leeren Worte im Graphen mit möglichst passenden Worten aus der Wortliste gefüllt.

3. **Mutation**
   Anschließend werden die neue erzeugten Graphen („Childs“) noch mutiert. Dazu werden zufällig Worte im Graphen ausgewählt und durch ein anderes Wort ersetzt.

Das mehrmalige Ausführen dieser Schritte durch die Iterationen führt zu einem relativ guten Ergebnis für den letztendlichen Graphen.

Mutationsansatz

Bei diesem Ansatz haben wir den bisherigen Algorithmus etwas modifiziert, sodass wir die Rekombination und Mutation kombiniert haben. Dabei hat jeder neue Graph nur noch ein Eltern-Graphen. Der Algorithmus durchläuft alle Wörter des Eltern-Graphen und fügt jedes mit einer Wahrscheinlichkeit von 75% in den Kind-Graphen ein. Die leeren Wörter im Kind-Graphen werden nun mit Wörtern aus der Wortliste aufgefüllt.
Dieser Ansatz läuft schneller als der Erste, da weniger Berechnungen nötig sind (bzw. Berechnungen kombiniert werden). Dennoch liefern beide Ansätze quasi die gleiche Qualität.

Komplexitätsanalyse

In diesem Abschnitt werden wir einen genauen Blick auf die Komplexität des Algorithmus werfen. Dabei wird der Mutationsansatz aus Abschnitt 1.2.4 zugrunde gelegt. Wir werden den Algorithmus im Folgenden in seine einzelnen Teile aufteilen. Als Basis dient uns dabei

der Code der Android-Anwendung. Wir beginnen auf der obersten Stufe, der execute-Methode des Algorithmus (aus Platzgründen haben wir für die Komplexität unerhebliche Teile weggelassen und durch Markierungen *[Code]* ersetzt):

Listing 1.2.1 Wrapper Code für die Rekursive Berechnungsfunktion

```
public synchronized CWTuple<CWGraph, Float> execute(CWWordManager mgr
    , CWGraph graph) {
    //[INIT DATA]
    for (int i=0; i<mPopulation; i++) {
        //process: generate the base population by cloning the blank
            graphs
        mGraphs.add(graph.clone());
    }
    //[INIT DATA]
    //process: start the genetic iteration
    for (int i=0; !flag && i<mIterations; i++) {
        //[SUB-INIT + GET TOP HALF OF POPULATION]
        CWGraph newChild = merge(null, null, blank, (Vector<CWWord>)
            mWords.clone());
        next.add(newChild);
        //process: create childs
        for (int j=(mPopulation/2); j<mPopulation; j++) {
            //[INIT DATA]
            //process: generates childs of the other half (using
                mutation and merging of parents)
            CWGraph child = merge(next.get((j - (mPopulation/2)) % (
                mPopulation/2)), next.get((j - (mPopulation/2) + 1) %
                (mPopulation/2)), blank, lstClone);
            //[PREPARE DATA]
        }
        //[EXIT CONDITIONS + LISTENER CODE]
    }
    //[PREPARE DATA]
    for (int i=0; i<best.countWords(); i++) {
        //[INIT DATA]
        for (int j=0; j<word.getLength(); j++) {
            //[REMOVE NODES]
        }
        for (int j=word.getLength() - 1; j>= 0; j--) {
            //[REMOVE NODES]
        }
    }
    //[RETURN]
}
```

Listing 1.2.2 Code für das Mergen der Graphen

```
private synchronized CWGraph merge(CWGraph graph1, CWGraph graph2,
    CWGraph blank, Vector<CWWord> words) {
     //[INIT DATA]
     if (graph1 != null && graph2 != null) {
          //[INIT DATA]
          //process: iterate through all words of both graphs and
              choose one word that is working
          for (int i=0; i<result.countWords(); i++) {
               //[PROPABILITY CHECK + INIT]
               slidingWindow(result, i, words, word);
          }
     }
     for (int i=0; i<result.countWords(); i++) {
          //[INIT + GET THE WORD]
          for (int j=0; j<words.size(); j++) {
               if (slidingWindow(result, i, words, ((w + j) % words.size
                   ())))
                    break;
          }
     }
     //[RETURN]
}
```

Listing 1.2.3 Code für das Einfügen der Wörter

```
private synchronized boolean slidingWindow(CWGraph graph, int
    graph_pos, Vector<CWWord> words, int word_pos) {
     for (int i=0; i<tempWords.size(); i++) {
          if (tempWords.get(i).getWord().equals(words.get(word_pos).
              getWord()) == true) {
               return false;
          }
     }
     for (int pos=0; pos<graph.getWord(graph_pos).getLength() - words.
         get(word_pos).getWord().length() + 1; pos++) {
          if (graph.getWord(graph_pos).applyWord(words.get(word_pos).
              getWord(), pos, false, words.get(word_pos).getHint())) {
               //[INIT]
               return true;
          }
     }
     //[RETURN]
}
```

Für die Analyse werden wir folgende Bezeichnungen wählen:

m = Anzahl der einfügbaren Wörter
n = Anzahl der Wörter im Graphen
i = Anzahl der Iterationen
p = Größe der Population
k = Anzahl der Knoten
Wir beginnen mit der Analyse im Code von Listing 1.2.4. Dort haben wir zuerst eine for-Schleife über alle Wörter die schon eingefügt wurden. Diese sind maximal die Anzahl der Wörter im Graphen lässt sich also mit $\mathcal{O}(n)$ abschätzen, wobei n die Anzahl der Wörter im Graphen darstellt. Der zweite Teil der Funktion geht durch alle Positionen des Wortes, wo das Wort eingefügt werden könnte. Für die Eingabedaten, die wir verwenden, ist die Komplexität dieser Schleife gegenüber der ersten vernachlässigbar.
Auch die nächste Funktion aus Listing 1.2.4 besteht aus zwei äußeren for-Schleifen. Die erste geht durch alle Wörter des Graphen und versucht Fragewörter mit der *slidingWindow* Funktion aus Listing 1.2.4 einzufügen. Die Komplexität ist daher $\mathcal{O}(m * n)$ (m für das Durchlaufen aller Wörter und n für das Einfügen über *slidingWindow*). Die zweite Schleife läuft ebenfalls durch alle Wörter und danach noch durch alle Fragewörter, um diese dann einzufügen. Entsprechend ist die Komplexität dieser Schleife $\mathcal{O}(m*n^2)$. Damit überlagert sie die erste Schleife und stellt die Komplexität für diese Funktion dar.
Letztendlich in der obersten Funktion aus Listing 1.2.4 finden sich drei äußere Schleifen. Die Erste erstellt eine neue Population. Der *clone* Vorgang hat eine Komplexität von $\mathcal{O}(n * k)$ und somit ist die gesamte Komplexität $\mathcal{O}(p * n * k)$. Die mittlere Schleife beinhaltet die eigentlichen Operationen. Im inneren wird für die Population die merge Funktion aufgerufen. Dies hat also eine Komplexität von $\mathcal{O}(p * m * n^2)$. Zusammen mit der äußeren Schleife, welche durch alle Iterationen läuft ergibt sich für diese Schleife eine Komplexität von $\mathcal{O}(i * p * m * n^2)$. Die letzte Schleife hat eine Komplexität von $\mathcal{O}(n * k)$, da für jedes Wort alle Knoten durchlaufen werden, um überflüssige Knoten zu entfernen. Folglich ist die Gesamt-Komplexität des Algorithmus $\boldsymbol{\mathcal{O}(i * p * m * n^2)}$. Wichtig ist jedoch, dass diese nur für den Worst-Case gilt. Im Durchschnitt sollte der Algorithmus deutlich schneller sein, da nie alle Listen vollständig durchlaufen werden müssen.

1.3 Praxisteil

1.3.1 Warum Android 4?

Wir haben uns dafür entschieden die Anwendung für Googles mobile Plattform zu entwickeln, da die Untermenge der Android-Begeisterten in unserem Team jede andere mögliche Untermenge in seiner Mächtigkeit übersteigt und uns außerdem mehr Geräte vorliegen, die auf dieser Plattform laufen.

Da das Update auf Version 4 unmittelbar bevorstand, beschlossen wir, von vorneherein für diese Version zu entwickeln. Gleichzeitig wurden auch zeitnahe Updates der Hersteller für zwei unserer drei Androidbasierten Testgeräte versprochen, so dass wir diese Version als beste Grundlage unseres Spiels empfanden. Außerdem sind wir sicher, dass das dem Geist dieses Wettbewerbs am meisten entspricht: Wir gehen mit der Zeit und eignen uns brandaktuelle Technologien an. Mit Android hatte jeder von uns ein *wenig* Erfahrung,

aber noch niemand hatte eine größere Anwendung dafür geschrieben.

1.3.2 Quelle der Frage/Antwort-Paare

Das freie Wörterbuch *Wiktionary* enthält nicht nur Hinweise zur Rechtschreibung und grammatikalische Eigenschaften von Wörtern, sondern auch (oft) eine kurze Angabe der Bedeutung. Umgekehrt taugt diese Beschreibung aber auch exzellent als Hinweis auf das Wort selbst, wir nutzen diese Eigenschaft aus, um uns eine Liste von Frage/Antwort-Paaren zusammenzustellen.
Ausgehend von einem Startwort bewegen wir uns wie ein Crawler mittels Breitensuche durch verlinkte Wiktionary-Artikel und speichern neu auftretende Wörter, deren Bedeutung beschrieben ist, in einer XML-Datei.

1.3.3 Aufbau der Anwendung

Die Android-Anwendung besteht hauptsächlich aus der Main-Klasse *GeoCrosswords*, der Klasse *Options* für Programmeinstellungen und der Klasse *OSMap* für die Anzeige der OpenStreetMap-Karte.
Desweiteren gibt es die Klassen *StartGame* für das Vorbereiten des Graphen zu Spielbeginn verantwortlich ist, *GameOptions* für die Einstellungen des neuen Spiels, sowie die Overlays *OSMapOverlay* für das Rätsel, *MyCustomLocationOverlay* als eigenes Standort-Overlay mit Pacman und *PopupOverlayItem* zur Anzeige der Fragestellung.
Außerdem gibt es noch drei Dialog-Klassen zur Anzeige des Eingabe-Dialogs (*EditDialog*), zur Anzeige der Glückwünsche bei vollständig gelöstem Rätsel (*SolvedDialog*) und zum Nachfragen bei Drücken des Zurück-Buttons im Spiel (*BackButtonDialog*).

GeoCrosswords.java

Diese Klasse stellt das Hauptmenü dar. Von hier aus gelangt der Spieler über zwei Buttons entweder in das Einstellungsmenü oder das Spielmenü.

Options.java

Diese Klasse stellt das Einstellungsmenü dar.

GameOptions.java

Diese Klasse stellt die Spieleinstellungen für das aktuelle Spiel mit deren Funktionalität und einen Button zum Starten des Spiels bereit.

StartGame.java

Diese Klasse stößt das Generieren des Kreuzworträtsels an. Zuerst werden mit Hilfe des Wiktionary-Crawlers Wörter gesucht, bzw. aus dem Internet heruntergeladen. Danach

wird die OpenStreetMap-Karte der Umgebung heruntergeladen und daraus ein Graph erstellt. Dieser Graph wird anschließend mit den zuvor gefundenen Wörtern befüllt. Während all dies geschieht, wird eine Fortschrittsanzeige angezeigt. Anschließend wird das eigentliche Spiel gestartet.

OSMap.java

Diese Klasse stellt die OpenStreetMap-Karte dar und ist über die Overlays für die Spieler-Interaktion verantwortlich. Es stehen im Grunde drei Overlays zur Verfügung:

- **OSMapOverlay.java**
  Das OSMapOverlay zeigt das Kreuzworträtsel an. Je nach Status der Wörter werden diese entweder als grün-, blau- oder rot-umrandete Kästchen dargestellt. Grüne Kästchen enthalten außerdem die Buchstaben des korrekten Lösungswortes, blaue Kästchen gehören zum aktuell ausgewählten Wort und rot-umrandete Kästchen wurden noch nicht gelöst. Außerdem können auch schwarz-ausgefüllte Füllkästchen dargestellt werden. Diese enthalten keine Buchstaben, sondern dienen nur dem Zusammenhang des Graphen. Beim Zeichnen der Kästchen wird zusätzlich deren Essbarkeit durch den Pacman bestimmt. Essbare Kästchen werden rund statt eckig dargestellt.
  Schließlich ist diese Klasse auch noch für die Darstellung und Entfernung der Marker am Anfang eines Wortes und die Erstellung der Popups unter Verwendung der *PopupOverlayItem.java* verantwortlich. Das PopupOverlayItem ist ein OverlayItem zur Darstellung der Popups, das Kenntnis vom aktuellen Wort hat.

- **ItemizedOverlayWithFocus**
  Das ItemizedOverlayWithFocus stellt den Hinweis zum aktuellen Wort als kleines gelbes Popup dar. Dazu benötigt es sowohl einen OnItemGestureListener als auch eine ArrayListe der darzustellenden PopupOverlayItems. Der OnItemGestureListener implementiert die Reaktion auf kurzes und längeres Antippen der Marker. Bei kurzem Antippen wird dem OSMapOverlay das neue aktuelle Wort übergeben, bei längerem Antippen öffnet sich der Dialog zur Eingabe des Lösungswortes.

- **MyCustomLocationOverlay.java**
  Dieses LocationOverlay zeigt die aktuelle Position des Spieler mit einem Pacman an. Befindet sich der Spieler in der Nähe des Anfang eines noch nicht gelösten und nicht gefressenen Wortes, so löst diese Klasse die Anzeige des Markers aus. Ist das Wort gelöst, so kann der Spieler den Pacman durch reale Bewegung über das gelöste und essbare Wort bewegen, so dass dieser das Wort "frisst". Wenn alle Wörter von der Karte verschwunden sind, wird der SolvedDialog aufgerufen.

Für die weiter Interaktion stehen dieser Klasse außerdem drei Dialog-Klassen zur Seite:

- **EditDialog.java**
  Dieser Dialog dient der Eingabe des Lösungswortes. Wird das richtige Lösungswort eingegeben, so wird das Wort als gelöst markiert und es ertönt ein lobender Sound.

Wurde das falsche Lösungswort eingegeben, ertönt ein trauriger Sound. Außerdem kann die Eingabe über einen Button auch abgebrochen werden.

- **SolvedDialog.java**
  Dieser Dialog erscheint, sobald alle Wörter vom Pacman "gefressen" wurden und zeigt das Gewinnen des Spielers an.

- **BackButtonDialog.java**
  Berührt der Spieler den Hardware-, bzw. Software-Zurück-Button seines Gerätes während des Spiels, so erscheint dieser Dialog. Er dient der Nachfrage, ob der Spieler tatsächlich zurück ins Hauptmenü möchte, damit das Spiel nicht ausversehen abgebrochen wird.

Zusätzlich besteht noch die Möglichkeit der Interaktion über die Menüleiste am oberen Rand des Displays. Dort sind zwei Buttons platziert. Einer der beiden dient dazu, die Kartenansicht wieder auf den aktuellen Standort zu zentrieren. Der andere zeigt den aktuellen Punktestand an und ermöglicht die Benutzung eines Jokers, der den Spieler allerdings 10 Punkte kostet.
Im über den Hardware-Button erreichbaren Menü besteht außerdem noch die Möglichkeit zur Aufgabe. In dem Fall wird ebenfalls ein trauriger Sound abgespielt und sämtliche Wörter werden aufgedeckt.

GCApplication.java

Die Klasse GCApplication dient der Klassenübergreifenden Speicherung von Informationen. Hier werden die Startkoordinaten, der Graph und der Punktestand gespeichert. Außerdem implementiert diese Klasse die Funktionalität des Wörter-"Fressens" für den Pacman und der Benutzung des Jokers.
Bewegt der Spieler sich, und damit den Pacman, über ein Wort, so wird dieses, begleitet von einem passenden Geräusch, vom Pacman "gefressen" und verschwindet damit von der Karte.
Den Joker kann der Spieler jederzeit einsetzen, wenn er die Lösung eines Wortes nicht weiß. Dieser Einsatz markiert das aktuelle Wort als gelöst, wodurch es aufgedeckt wird.

Genauere Informationen zu Variablen und Methoden der einzelnen Klassen können im dokumentierten Code der jeweiligen Klasse eingesehen werden.

1.3.4 Verwendete externe Bibliotheken

Wir verwenden **osmdroid**[1] zur Darstellung einer zoom- und scrollbaren OpenStreetMap-Karte mit diversen Overlays, sowie eine Abhängigkeit davon, die Logging-Bibliothek **slf4j**[2].

[1] `http://code.google.com/p/osmdroid/`
[2] `http://www.slf4j.org/`

1.4 Ausblick

Da wir aufgrund des knappen Zeitlimits nicht alle unsere Ideen umsetzen konnten, folgt hier ein kleiner Ausblick auf mögliche zukünftige Features.

- **Fragenmodus**: Auswahl zwischen verschiedenen Fragenmodi, z.B. Ortsbezogene Fragen, Fragen zu einem bestimmten Thema oder verschiedene Schwierigkeitsgrade
- **Vorgebbare Spielzeit**: Die Möglichkeit zur Vorgabe der zu erwartenden Spieldauer. In die Berechnung der Größe des Spielfeldes fließt dann diese Zahl, die durchschnittliche Gehgeschwindigkeit sowie eine ungefähre Schwierigkeit der Fragen ein.
- **Soziale Features**: "Teilen" des aktuellen Standortes und der aktuellen Frage mit Freunden auf Facebook/Twitter/G+ etc.
- **Eulerpfad**: erhöhte Schwierigkeit durch die Bedingung, dass kein Weg zweimal gegangen werden darf
- **Animation**: Pacman frisst gelöste Teile des Kreuzworträtsels auf
- **Hilfefunktion**: aus der App heraus das Internet zu befragen (z.B. Google/Wikipedia)
- **Speichern**: Speichern eines Zwischenstandes des Spiels um es später fortsetzen zu können

Außerdem gibt es natürlich noch Möglichkeiten zur Verbesserung des Akku- und Speicherverbrauchs.

2 Installationsanleitung

2.1 Installation des fertigen Programmpakets

2.1.1 Installation von Nicht-Market-Quellen zulassen

Da unser Programm nicht im Android-Market verfügbar ist, muss als erstes die Installation von Programm aus anderen Quellen erlaubt werden. Dazu navigiert man in den Einstellungen zum Punkt "Sicherheit" und macht dort unter dem Menüpunkt "Geräteverwaltung" ein Häckchen bei "Unbekannte Herkunft".

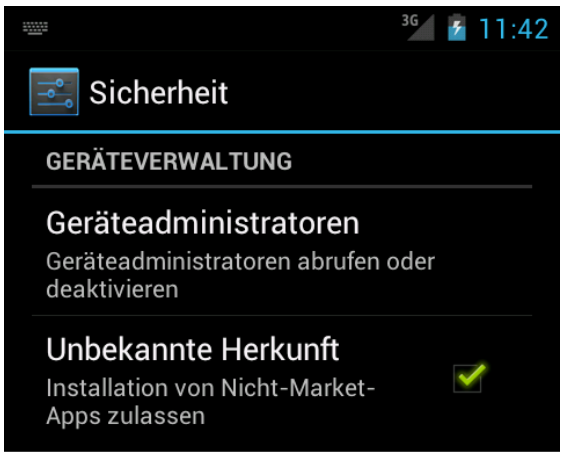


Abbildung 2.1: Geräteverwaltung von Android 4 im Emulator mit aktiviertem Häckchen bei "Unbekannte Herkunft"

2.1.2 Programmpaket dem Telefon zur Verfügung stellen

Als nächstes muss das Programmpaket mit der Endung ".apk" dem Telefon zu Verfügung gestellt werden. Dies kann auf verschiedenen Wegen geschehen:

SD-Karte

Man kann das Programmpaket auf die SD-Karte verschieben und dann dort über einen Dateimanager vom Telefon aus aufrufen.

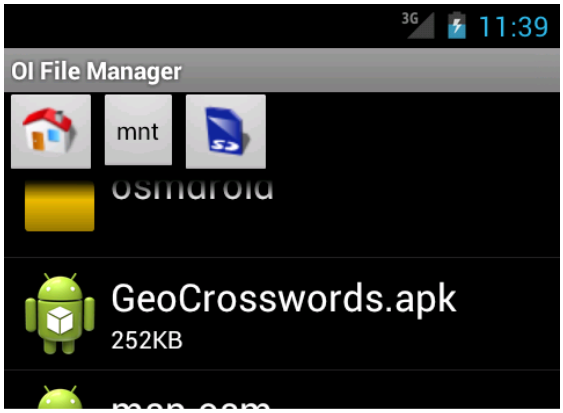


Abbildung 2.2: Ansicht der GeoCrosswords.apk im OI FileManager unter Android 4 im Emulator

Online-Speicher

Alternativ kann man das Programmpaket auch bei einem Online-Speicheranbieter wie zum Beispiel Dropbox hochladen und dann über die Android-App des Anbieters aufrufen.

2.1.3 Installation

Zur Installation muss einfach nur das Programmpaket ausgewählt werden, dann öffnet sich ein Dialog zur Installation.

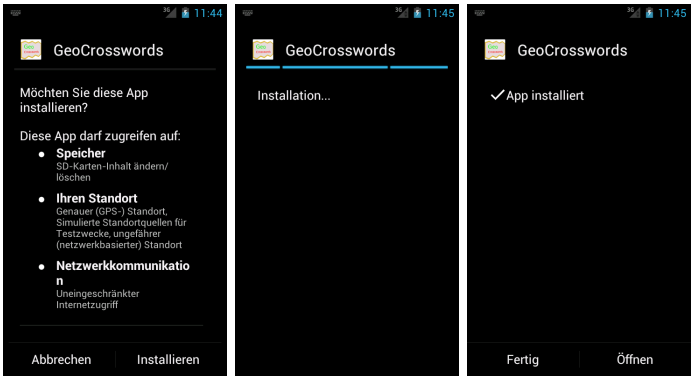


Abbildung 2.3: Installationsroutine der GeoCrosswords.apk unter Android 4 im Emulator

2.2 Anwendung selbst kompilieren

Möchte man die Anwendung selbst kompilieren, benötigt man das Build-Tool *ant*, die Android SDK Tools sowie die Android API 15. Man wechselt in den Projektordner, führt zunächst

```
android update project -p . -t android-15
```

aus, gefolgt von einem

```
ant debug install
```

bei angeschlossenem Gerät oder laufendem Emulator. Der Emulator sollte auf Deutsch gestellt werden, bei uns kam es sonst zu seltsamen Abstürzen durch den Spellchecker, die auf dem Gerät nicht auftraten.

2.3 Hinweis zu den Wortlisten

Die heruntergeladenen Wortlisten befinden sich auf der SD-Karte, im Ordner „/sdcard/-geocrosswords/words/“. Hier kann man auch eigene Wortlisten hinterlegen, eine Liste mit über 8000 Wörtern etwa befindet sich unter

`http://files.morr.cc/Wissenschaft.xml`

3 Bedienungsanleitung

Das Programm soll dem User eine einfache und intuitive Bedienoberfläche bieten. Deshalb ist das eigentliche Menü relativ einfach gehalten und bietet lediglich 2 Optionen: "Neues Spiel" und "Einstellungen".

Abbildung 3.1: Das Hauptmenü

3.1 Neues Spiel (Spieleinstellungen)

In diesem Menü kann der Benutzer einige Rahmenlinien für das Spiel festlegen.

- Der streng geheime Entwickler-Modus ermöglicht ein Testen der Anwendung, ohne tatsächlich herumlaufen zu müssen. Per Tap auf die Karte gelangt man sofort an diesen Ort.
- Die Kantenlänge legt die Größe des verwendeten Kartenausschnittes fest. (Werte, die größer als 6 sind werden aus Performancegründen geblockt).
- Außerdem kann die verwendete Wortliste festgelegt werden, aus der die Wörter zur Erzeugung des Rätsels geschöpft werden. Entweder, man wählt hier eine bereits

vorhandene Datei, oder man lässt eine neue Wortdatei herunterladen, dann ist das „Thema“ sowie die Anzahl der Wörter wählbar. Richtwert: 500 Wörter ergeben schon ein ganz ordentliches Rätsel. Hundert Wörter herunterzuladen dauert etwa eine Minute, natürlich je nach Geschwindigkeit der Internetverbindung.

3.2 Im Spiel

An der Position des Spielers befindet sich ein kleiner, gelber Wortfresser, der sich mit dem Gerät bewegt und dreht. Ziel des Spiels ist es, alle Buchstaben zu fressen! An den Anfängen derjenigen Wörter, auf denen man steht, befinden sich orangefarbene Marker, die auf einen einfachen Tap den Hinweis anzeigen, welches Wort dort einzutragen ist und auf einen langen Tap ein Fenster zur Eingabe des Wortes öffnen. Groß- und Kleinschreibung ist hier unerheblich.
Hat man ein Wort richtig gelöst, werden die Buchstabenfelder, die in keinen anderen Wörtern mehr auftauchen, zu Kreisen und sind nun fressbar. Pro Buchstabe gibt es einen Punkt. Für 10 Punkte kann man sich einen Joker kaufen, der die Buchstaben in näherer Umgebung aufdeckt.
Die Zu-mir-Aktion ist selbsterklärend. Die Menütaste eröffnet einem die Option, aufzugeben, was alle Wörter aufdeckt und den Punktestand auf Null setzt. Über den Zurück-Hardwarebutton (oder bei nativen Android 4 Geräten den Zurück-Button am linken unteren Bildschirmrand) gelangt man nach einem Bestätigungsdialog zurück ins Hauptmenü.

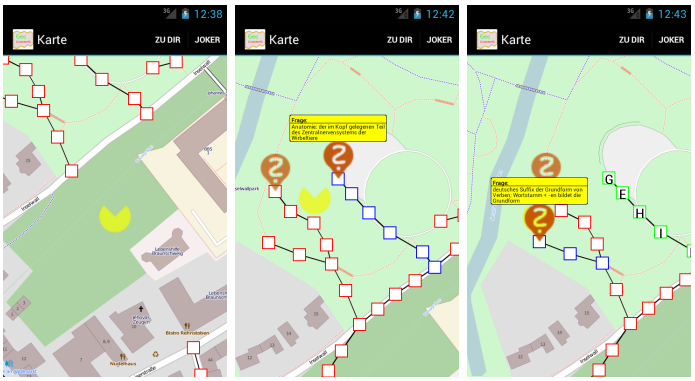


Abbildung 3.2: Screenshots des Spiel: Das linke zeigt den Spieler auf dem zu einem Rätselwort, das mittlere zeigt den Spieler neben dem aktuellen Wort samt dem dazugehörigen Hinweis und auf den rechten wurde ein Wort schon gelöst

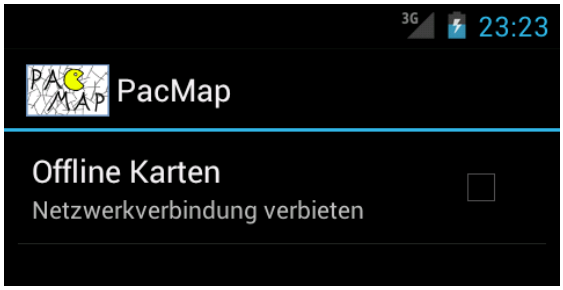


Abbildung 3.3: Das Einstellungsmenü

3.3 Einstellungen

Hier hat der User die Möglichkeit einige grundlegende Optionen einzustellen:

- Offline-Maps
  Bietet die Möglichkeit Kartenmaterial zu einem beliebigen Zeitpunkt herunterzuladen und später zu verwenden. Eine Internetverbindung wird dann nicht mehr benötigt.