28. Bundeswettbewerb Informatik

Sebastian Morr

16. November 2009

Inhaltsverzeichnis

Allgemeine Hinweise 1

Aufgabe 2: Handytasten 2

Aufgabe 3: Wegfehler 5

Aufgabe 4: EU-WAN 10

Allgemeine Hinweise

In dieser Arbeit werden die Aufgaben 2, 3 und 4 gelöst.

Zur Umsetzung der Programmieraufgaben benutze ich die objektorientierte Skriptsprache Ruby, da sie eine schnelle Softwareentwicklung ermöglicht, leicht schreib- und lesbar sowie erweiterbar ist und (wichtig!) einer Open-Source-Lizenz unterliegt.

In den letzten beiden Aufgaben benutze ich zur Bildmanipulation die RMagick-Bibliothek, ein Ruby-Interface zu dem weitverbreiteten ImageMagick. Es befindet sich aufgrund von verzweigten Abhängigkeiten nicht auf der CD, lässt sich aber leicht als Gem oder über den Paketmanager einer Linux-Distribution installieren.

Entwickelt und auf Funktionsfähigkeit überprüft wurde mit dem Standardinterpreter[1] in Version 1.8.7, auf einer x86-64-Architektur unter GNU/Linux, Kernel 2.6.31.

Um Redundanzen zu vermeiden und die Arbeit leichter lesbar zu machen, wird die Programmdokumentation teilweise im Quelltext selbst vorgenommen, ein Überblick über das Gesamtkonzept hingegen findet sich im Fließtext.

[1] http://www.ruby-lang.org/de/

Aufgabe 2: Handytasten

Ich benutze zur Bearbeitung der Teilaufgaben den Begriff der „Ebene“. Die Ebene beschreibt, wie viele Tastendrücke man braucht, um einen bestimmten Buchstaben zu schreiben. Das „D“ liegt bei der konventionellen Belegung beispielsweise in Ebene 1, das „E“ hingegen in Ebene 2, weil man zweimal auf die Taste „3“ drücken muss.

Teilaufgabe 1

Je weniger Tasten der Benutzer zur Eingabe eines bestimmten Textes drücken muss, desto besser. Die Buchstaben, die in der betrachteten Sprache eine hohe Häufigkeit haben, sollten daher in der ersten Ebene liegen. Ich formalisiere das Ganze nun etwas:

Jedem Buchstaben b_i eines Alphabets der Länge n mit $1 \leq i \leq n$ sei eine Wahrscheinlichkeit $P(b_i)$ zugeordnet, zu der er in einem Text der betrachteten Sprache auftaucht. Ihm sei weiterhin zugeordnet eine Ebene $E(b_i)$, die angibt, mit wie vielen Tastendrücken man den Buchstaben b_i erreichen kann.

Je öfter der Buchstabe in der betrachteten Sprache vorkommt, desto „schlimmer“ ist es, wenn er eine Ebene weiter unten liegt, die Kosten eines einzelnen Buchstabens kann mal also mit $P(b_i) \cdot E(b_i)$ beschreiben.

Die „Kosten“ der gesamten Belegung definiere ich hiermit zu $\sum_{i=1}^{n} P(b_i) \cdot E(b_i)$, der Summe der Kosten der einzelnen Buchstaben. Je höher dieser Wert ist, desto ungünstiger ist die Belegung, weil der Benutzer durchschnittlich mehr Tastendrücke braucht, um einen Text einzugeben.

Teilaufgabe 2

Zunächst musste eine Datenstruktur her, die eine Belegung repräsentiert. Ich entschied mich für ein Array, das pro Buchstabentaste ein Element enthält. Es enthält Zahlen, die die Menge der auf dieser Taste befindlichen Buchstaben angibt. Die herkömmliche Belegung wird in diesem Format als `[3,3,3,3,3,4,3,4]` dargestellt. Da die Reihenfolge und die Anzahl der Buchstaben unveränderlich ist, ist es nicht notwendig, die Buchstaben selbst in der Datenstruktur aufzuführen.

Die Bedingungen für eine Verteilung von b Buchstaben auf t Tasten sind folgende:

- Das Array muss aus t Elementen bestehen, um alle Tasten abzubilden.
- Jedes Element muss mindestens den Wert 1 haben. Eine Belegung mit leeren Tasten ist sicherlich nicht die optimale.
- Die Summe der Werte muss b betragen, sonst sind nicht alle Buchstaben auf der Tastatur.

Mein Programmcode stellt sicher, dass diese Bedingungen für jede generierte Belegung erfüllt sind.

Implementation

Ein Wort vorneweg: Ich verzichte im gesamten Programm auf Fehlerabfragen, um den Code übersichtlicher zu machen. Die Eingabe muss den Vorgaben entsprechen, damit das Programm funktioniert.

Das Programm erwartet als Eingabe die Häufigkeiten der einzelnen Buchstaben von „A“ bis „Z“, die durch Zeilenumbrüche getrennt sind. Dabei ist es egal, ob die Anzahl eine absolute

Häufigkeit (etwa die tatsächliche Anzahl der Buchstaben in einem gegebenen Text) oder eine relative Häufigkeit (in Prozent) ist. Die Herunterrechnung auf Anteile an der Gesamtbuchstabenzahl im ersten Fall ist nicht nötig, denn dies würde alle Kosten um den gleichen Faktor verringern, was bei der Ermittelung der optimalen Belegung keine Rolle spielt.

Das Programm generiert alle gültigen Belegungen mithilfe einer rekursiven Funktion und gibt diejenige aus, die die niedrigsten Kosten hat.

Listing 1: handytasten.rb

```
# Erweitere die Array-Klasse um eine Methode, die alle Elemente
    addiert.

class Array
    def sum
        inject(0){|sum, x| sum+x}
    end
end

# Algorithmus, der alle Möglichkeiten generiert, b Buchstaben
# aufsteigend auf t Tasten anzuordnen.
# 'vor' enthält die bisherige Belegung, anfangs ist diese leer.
# Wenn nur noch eine Taste frei ist, landen alle verbleibenden
# Buchstaben auf dieser Taste und das Resultat wird der Funktion
# 'fertig' übergeben.
# Ansonsten werden alle Möglichkeiten für die erste Taste gebildet
# (in der Variable 'i') und für die restlichen Tasten die Funktion
# für die verbleibenden Buchstaben- und Tastenanzahl aufgerufen.

def generiere b, t, vor=[]
    (fertig vor + [b]; return) if t == 1
    (b-t+1).downto(1) do |i|
        generiere  b-i, t-1, vor + [i]
    end
end

# Berechne die Kosten einer Belegung.  Wandle dazu die Datenstruktur
# zunächst in ein Format um, das pro Buchstabe ein Element enthält,
# dessen Wert angibt, in welcher Ebene es sich befindet.
# Multipliziere dann jeweils die Häufigkeit und die Ebene des
# Buchstabens und summiere alle Ergebnisse.

def kosten belegung
    ebenen = belegung.inject([]) do |vor, g|
        vor + (1..g).to_a
    end
    $haeufigkeiten.zip(ebenen).collect{|b| b[0]*b[1]}.sum
end

# Diese Funktion wird aus obenstehendem Algorithmus aufgerufen, wenn
# eine Belegung fertig ist. Sie vergleicht mit dem bisherigen Rekord
# und ersetzt diesen, falls keiner existiert oder die Kosten für die
# übergebene Belegung niedriger sind.

def fertig belegung
    if not defined?($beste_belegung) or kosten(belegung) < kosten(
       $beste_belegung)
```

```
        $beste_belegung = belegung
    end
end

# Funktion, die die Buchstabengruppen einer Belegung ausgibt.
# Das Alphabet wird dazu in Gruppen geschnitten, deren Größe durch
# die Belegung vorgegeben wird.

def zeige_gruppen belegung
    alphabet = ("A".."Z").to_a.join
    p belegung.collect{|buchstaben| alphabet.slice! 0,buchstaben}
end

# Lies die Häufigkeitsverteilung aus der Standardeingabe, generiere
# dann alle Möglichkeiten, 26 Buchstaben auf 8 Tasten zu verteilen
# und zeite die beste Belegung an.

$haeufigkeiten = $stdin.readlines.collect{|line| line.to_i}[0..25]
generiere 26, 8
zeige_gruppen $beste_belegung
```

Teilaufgabe 3

Lässt man das Programm die deutsche Buchstabenverteilung einlesen, erzeugt sie folgende Ausgabe:

```
$ ruby handytasten < deutsch.txt
["AB", "CD", "EFG", "HIJK", "LM", "NOPQ", "RS", "TUVWXYZ"]
```

Auffallend ist die Position des „E“ auf erster Ebene. Das „E“ ist der im Deutschen am häufigsten vorkommende Buchstabe (die Beispieldatei ordnet ihm eine Häufigkeit von 17,4

```
$ ruby handytasten.rb < englisch.txt
["AB", "CD", "EFG", "HIJK", "LM", "NOPQ", "RS", "TUVWXYZ"]
```

Aus der englischen Quelldatei wird exakt die selbe Ausgabe erzeugt. Die Häufigkeiten der Buchstaben stimmen auch ansatzweise überein, denn bei beiden Sprachen handelt es sich um westgermanische Sprachen.

```
$ ruby handytasten.rb < finnisch.txt
["ABCD", "EFGH", "IJ", "KLM", "NOPQ", "RS", "TUVWX", "YZ"]
```

Die finnische Sprache enthält viele „I“-s und „Y“-s und wenige „C“-s, daher liegen die Grenzen hier deutlich anders.

```
$ ruby handytasten.rb < franzoesisch.txt
["AB", "CD", "EFGH", "IJK", "LM", "NOPQ", "RS", "TUVWXYZ"]

$ ruby handytasten.rb < italienisch.txt
["AB", "CD", "EFGH", "IJK", "LM", "NOPQ", "RS", "TUVWXYZ"]
```

Im Französischen kommt das „H“ selten vor und landet auf einer höheren Ebene, im Italienischen ist das „I“ sehr wichtig. Diese Kriterien führen dazu, dass beide Sprachen die selbe Belegung erhalten.

```
$ ruby handytasten.rb < polnisch.txt
["ABCD", "EFGH", "IJ", "KLM", "NOPQ", "RSTUV", "WX", "YZ"]
```

Polnisch enthält vergleichsweise viele „Y“-s und „W“-s, daher ergibt sich diese ungewöhnliche Belegung.

```
$ ruby handytasten.rb < spanisch.txt
["AB", "CD", "EFGH", "IJK", "LM", "NOPQ", "RS", "TUVWXYZ"]
```

Es ist auffällig, dass die Sprachen Spanisch, Französisch und Italienisch die selbe Belegung zugewiesen bekommen. Da es sich bei diesen dreien um romanische Sprachen handelt und sie somit einen ähnlichen Wortschatz aufweisen ist dies jedoch nicht weiter verwunderlich.

Aufgabe 3: Wegfehler

Teilaufgabe 1

Lösungsidee

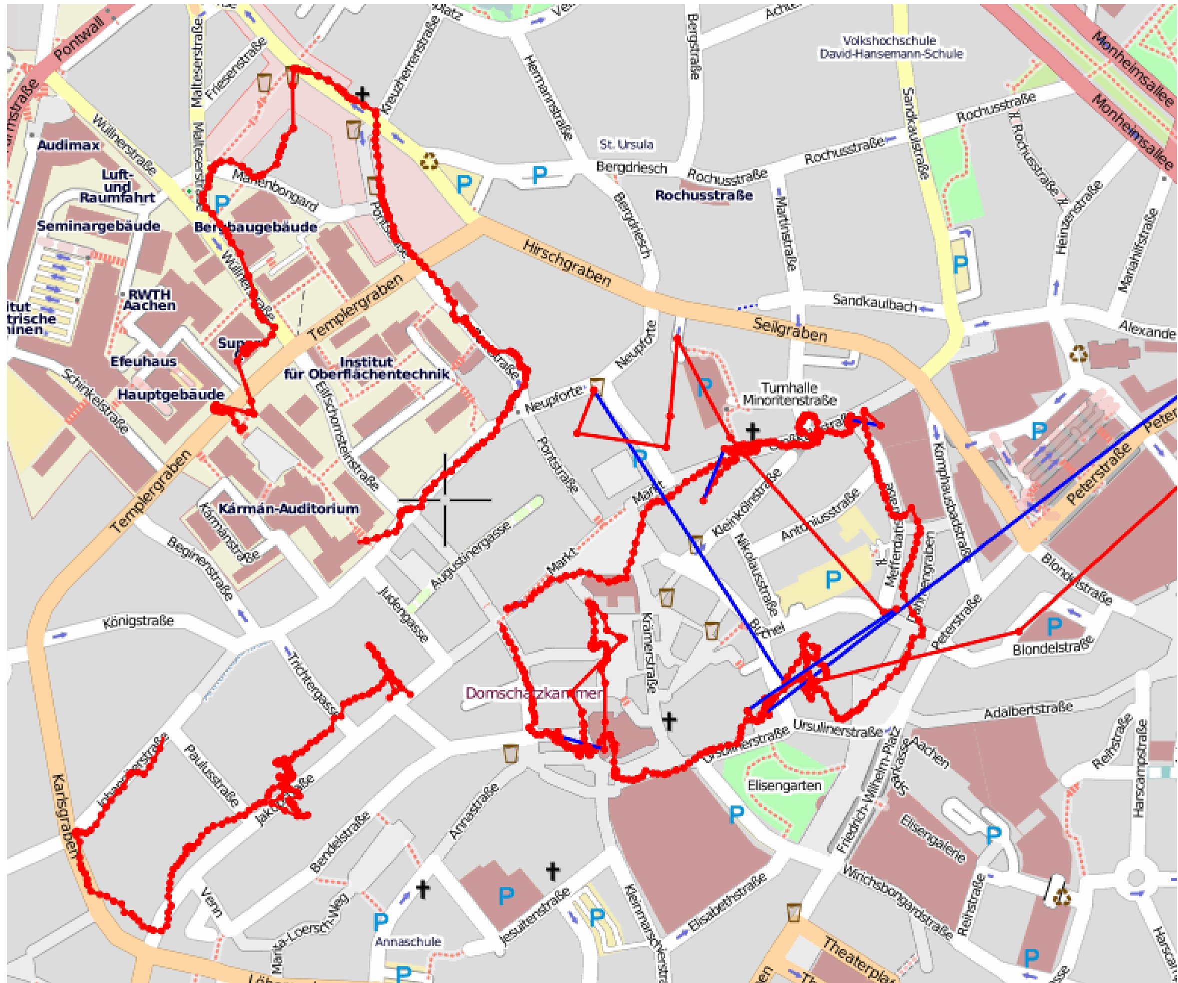
Zur Darstellung der GPS-Koordinaten auf der Karte ist es notwendig, sie in Pixelkoordinaten umzurechnen. Dazu werden die zur Verfügung gestellten Eck-Koordinaten der Karte genutzt: Die Punkte werden in die (relative) Entfernung von der linken oberen Ecke umgerechnet und mit dem errechneten „Pixel pro Grad“-Wert multipliziert. Das Resultat sind x- und y-Werte, die direkt auf der Karte eingezeichnet werden können.

Implementation

Das Programm liest GPS-Daten in dem gegebenen Format `Zeit,Breite,Länge` ein, wobei Breite und Länge dezimal und in Grad angegeben werden. Die Werte werden als Elemente der Klasse `Messpunkt` in der Klasse `Log` abgelegt. Schließlich werden diese Punkte von der Klasse `Karte` grafisch angezeigt. Einzelne Messwerte werden als Punkt dargestellt, aufeinanderfolgende Werte sind durch eine Linie verbunden. Dies macht das Erkennen von „Außreißern“ im nächsten Aufgabenteil einfacher. Punkte, die zeitlich mehr als 6 Sekunden[2] auseinanderliegen, werden mit blauen Linien verbunden, um zu verdeutlichen, dass die kontinuierliche Aufzeichnung hier unterbrochen wurde. Die resultierene Grafik sieht wie folgt aus:

[2] Die Logs 2 und 3 enhalten anfangs einen 6-Sekunden-Abstand. Das führe ich auf Effekte beim Start der Aufzeichnung zurück (höhere Prozessorlast?) und werte ich nicht als Fehler.

Abbildung 1: Die unkorrigierten GPS-Logs



Teilaufgabe 2

Lösungsidee

Zunächst ist zu klären, was als Fehler gewertet wird. Ich gehe davon aus, dass Dominic zu Fuß unterwegs ist, denn eine mittlere Fortbewegungsgeschwindigkeit von etwa 1 m/sec deutet darauf hin. Die Höchstgeschwindigkeit zu Fuß setze ich (sehr vorsichtig) auf den aktuellen Weltrekord: 100 Meter in 9,58 Sekunden. Wer weiß, ob Dominic mit Nachnamen „Bolt“ heißt. Geben zwei aufeinander folgende GPS-Daten eine schnellere Bewegung an, sind sie offensichtlich falsch.

Da für die Ausreißer eine Rekonstruktion unter keinen Umständen möglich ist (sie können, wie ich aus eigener Erfahrung mit GPS-Geräten weiß, in alle Richtungen gestreut sein), werden sie aus dem Log gelöscht. Auf der Karte wird die Verbindung zwischen den verbleibenden gültigen Punkten dann als gerade Linie dargestellt, was den tatsächlichen Weg auf jeden Fall akkurater darstellt als mit Ausreißer.

In den Daten treten Pausen auf, die länger sind als der sonstige 5-Sekunden-Abstand - teilweise von mehreren Minuten. Die folgenden Messpunkte sind oft weit entfernt, liegen aber wegen des großen zeitlichen Abstandes im Bereich des Möglichen. Ich entschied mich trotzdem dagegen, diese Punkte zuzulassen, denn auf dem Weg zu diesen Punkten hätte das GPS-Gerät in der Regel weitere Punkte aufzeichnen müssen. Ich erkläre mir diese Ausreißer

so, dass Dominic an einem Ort mit sehr schlechtem GPS-Empfang war (Gebäude, Tunnel), und das Gerät sehr selten dennoch Punkte aufzeichnete - die durch die Umgebung jedoch völlig verfälscht waren.

In Log 1 und 3 fand mein Verfahren keine Fehler. Das zweite gegebene Log enthält deutliche Ausreißer, die durch die Korrektur beseitigt werden konnten:

Abbildung 2: Die korrigierte log2.csv

Das Ablaufprotokoll zeigt die gefundenen Fehler:

```
$ ruby wegfehler.rb
Fehler gefunden: Um 09:07 Uhr 61 Meter in 120 Sekunden!
Fehler gefunden: Um 09:18 Uhr 50 Meter in 245 Sekunden!
Fehler gefunden: Um 09:30 Uhr 1856 Meter in 240 Sekunden!
Fehler gefunden: Um 09:30 Uhr 1463 Meter in 5 Sekunden!
Fehler gefunden: Um 09:30 Uhr 321 Meter in 5 Sekunden!
Fehler gefunden: Um 09:41 Uhr 417 Meter in 435 Sekunden!
Fehler gefunden: Um 09:41 Uhr 141 Meter in 5 Sekunden!
Fehler gefunden: Um 09:41 Uhr 78 Meter in 5 Sekunden!
Fehler gefunden: Um 09:41 Uhr 129 Meter in 5 Sekunden!
Fehler gefunden: Um 09:41 Uhr 296 Meter in 5 Sekunden!
Fehler gefunden: Um 09:43 Uhr 255 Meter in 85 Sekunden!
Fehler gefunden: Um 09:49 Uhr 77 Meter in 20 Sekunden!
Fehler gefunden: Um 09:51 Uhr 100 Meter in 5 Sekunden!
```

Implementation

Eine zusätzliche Methode `korrigiere!` der Klasse `Log` überprüft für alle Datenpaare, ob die oben beschriebenen Kriterien erfüllt sind. Ist dies nicht der Fall, werden die Werte als falsch markiert und anschließend aus dem Log entfernt.

Quelltext

Listing 2: wegfehler.rb

```
require "RMagick"
include Magick
require "time"
include Math

# Erweitere die Klasse Array um eine Methode, die für alle paarweise
# aufeinander folgenden Elemente einen Block aufruft.
class Array
    def each_pair
        0.upto(size-2) do |i|
            yield self[i], self[i+1]
        end
    end
end

# Erweitere die Klasse Float um eine Methode, die von Grad in Radian
# umrechnet.
class Float
    def to_rad
        self*PI/180.0
    end
end

# Klasse, die ein GPS-Log repräsentiert.
class Log < Array

    # Lies eine CSV-Datei ein und füge ihre einzelnen Punkte hinzu.
    def initialize datei
        IO.readlines(datei)[1..-1].each do |line|
            time,y,x = line.split(",")
            self << Messpunkt.new(x,y,time)
        end
    end

    # Erzeuge für alle aufeinanderfolgenden Punkte ein Objekt der
    # Klasse Verbindung und rufe den Block auf.
    def alle_verbindungen
        each_pair do |p1, p2|
            yield Verbindung.new(p1,p2)
        end
    end

    # Korrigiert das Log, indem Werte gelöscht werden, die zeitlich
    # oder räumlich zu weit auseinanderliegen.
    def korrigiere!
        mps = 100/9.58
        alle_verbindungen do |v|
            if v.zeit > 6 or v.strecke > v.zeit*mps
                v.p1.falsch!
                v.p2.falsch!
                puts "Fehler gefunden: Um #{v.p2.zeit.strftime("%H:%M
                   ")} Uhr #{v.strecke.round} Meter in #{v.zeit.to_i}
                    Sekunden!"
            end
        end
```

```
        reject!{|p| p.falsch}
    end
end

# Klasse, die einen einzelnen Messpunkt darstellt.
class Messpunkt
    attr_reader :x, :y, :zeit, :falsch

    # Wandle die Daten in die richtigen Dateitypen um und markiere
    # den Punkt zunächst als 'richtig'.
    def initialize x, y, zeit
        @x, @y = x.to_f, y.to_f
        @zeit = Time.parse(zeit)
        @falsch = false
    end

    # Berechne die Entfernung zu einem anderen Punkt in Metern.
    # Verwende hierzu die Formel der 'Great-circle distance'.
    # 'r' ist der mittlere Erdradius.
    def entfernung_zu messpunkt
        r = 6371
        x1 = self.x.to_rad
        y1 = self.y.to_rad
        x2 = messpunkt.x.to_rad
        y2 = messpunkt.y.to_rad
        acos(sin(x1)*sin(x2)+cos(x1)*cos(x2)*cos(y2-y1))*r*1000;
    end

    # Markiere diesen Messpunkt als falsch.
    def falsch!
        @falsch = true
    end
end

# Klasse, die GPS-Logs auf einer Karte darstellt.
class Karte

    # Setze die Eckdaten sowie Breite und Höhe der Karte.
    def initialize
        @min_y, @min_x = 50.7820, 6.0711
        @max_y, @max_x = 50.7716, 6.0918
        @breite, @hoehe = 968, 767
    end

    # Zeichne die 'logs' auf der Karte ein und gib diese aus.
    def zeichne logs
        karte = ImageList.new("karte.png")
        weg = Draw.new
        weg.stroke_width(2)
        weg.stroke("red")
        weg.fill("red")

        logs.each do |log|
            log.alle_verbindungen do |v|
                if v.zeit > 6
                    weg.stroke("blue")
```

```
                else
                    weg.stroke("red")
                end

                x1, y1 = rechne_um(v.p1.x, v.p1.y)
                x2, y2 = rechne_um(v.p2.x, v.p2.y)

                weg.line(x1,y1,x2,y2)
                weg.stroke("red")
                weg.circle(x1,y1,x1+1,y1)
            end
            weg.draw(karte)
        end

        karte.display
    end

    # Rechne die Gradangaben in Pixelkoordinaten um.
    def rechne_um x,y
        [(x - @min_x) * @breite / (@max_x-@min_x),
            (y - @min_y) * @hoehe / (@max_y-@min_y)]
    end
end

# Klasse, die die Verbindung zwischen zwei Messwerten repräsentiert.
class Verbindung
    attr_reader :strecke, :zeit, :p1, :p2
    def initialize p1, p2
        @strecke = p1.entfernung_zu(p2)
        @zeit = p2.zeit - p1.zeit
        @p1 = p1
        @p2 = p2
    end
end

# Lade die drei GPS-Logs, korrigiere sie und gib sie aus.

logs = ["log1.csv", "log2.csv", "log3.csv"]
logs.collect! do |log|
    log = Log.new(log)
    log.korrigiere!
end

karte = Karte.new
karte.zeichne(logs)
```

Aufgabe 4: EU-WAN

Teilaufgabe 1

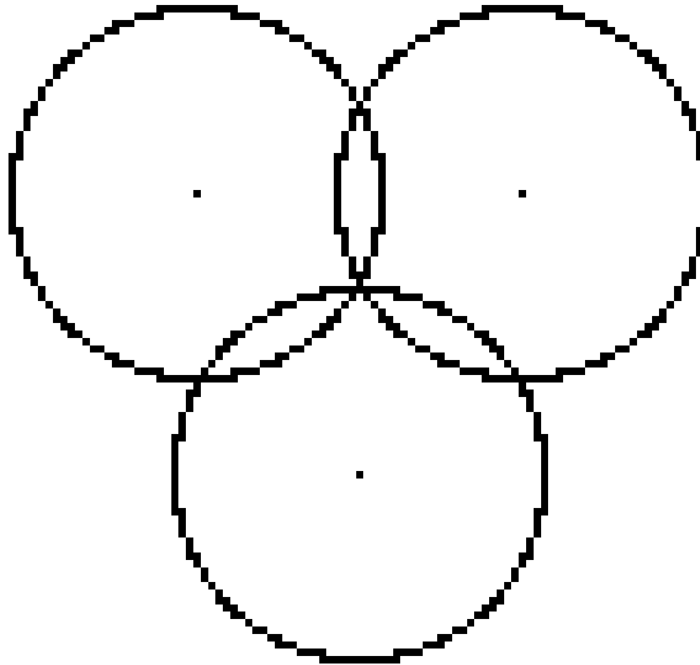
Das vorgegebene Sendegebiet hat einen Durchmesser von 51 Pixeln, meine Ansatz lässt aber beliebig große (und beliebig geformte) Sendebereiche zu.

Ich kombiniere zur Lösung dieser Aufgabe zwei Methoden:

Zuerst bedecke ich die gesamte Fläche mit einem regelmäßigen Muster von Sendemasten, die

sich in einem Abstand von 43 Pixeln zueinander befinden (siehe Abbildung). Ein Sendemast ist von sechs anderen umgeben (wie bei Bienenwaben), so ergeben sich möglichst geringe Überschneidungen bei vollständiger Abdeckung der Fläche. Die Masten, die sich im Wasser befinden, werden gelöscht.

Abbildung 3: Abstand der Sendemasten



Nun sind immer noch Stellen auf der Karte, die nicht im Sendebereich eines Masts liegen. Um auch die Bewohner dieser Gegenden in den Genuss eines kabellosen Internetanschlusses zu bringen, wende ich eine Zufallsmethode an: Ein Landpixel wird ausgesucht, der noch nicht besendet wird. Das Programm berechnet die Anzahl der Pixel, die zusätzlich besendet würden, wenn man an dieser Stelle einen Mast platzierte. Dieser Vorgang wird x-mal wiederholt (eine Anzahl von etwa 50 hat sich als günstig herausgestellt) und an der besten Stelle wird schließlich ein Mast gesetzt. Dieses zufällige Füllen wird solange wiederholt, bis die gesamte Landfläche bedeckt ist.

Ich entschied mich für diese Vorgehensweise, weil ein Algorithmus, der jeden möglichen Standort durchgeht, viel zu zeitaufwendig wäre.

Da mein Algorithmus ganz wesentlich auf den Zufall vertraut, findet er nicht die optimale Lösung. Man sollte das Programm daher wiederholt laufen lassen, bis es eine Anzahl an Sendemasten liefert, die im finanziellen Rahmen liegt.

Eine mögliche Platzierung mit 76 Sendemasten, die ich mit meinem Programm gefunden habe, sieht wie folgt aus:

Abbildung 4: Die EU ist online

Die Positionen der Sendemasten befindet sich in der Datei `masten.txt`.

Programmdokumentation

Das Programm liest die Karte im PBM-Format ein und wandelt sie in ein zweidimensionales Array um. Mit dem Sendebereich wird genauso verfahren.

Folgend werden die zwei beschriebenen Methoden hintereinander ausgeführt, die eine in der Funktion `muster`, die zweite in der Funktion `rest`. Dabei wird laufend die Anzahl der verbleibenden Landpixel verfolgt, in der Funktion `land` werden diese in ein eigenes Array gesteckt. Erreicht die Anzahl der unbesendeten Pixel Null, werden die Positionen und die Anzahl der Sendemasten ausgegeben und zur Veranschaulichung auf der Karte eingezeichnet.

Teilaufgabe 2

Durch die Beschaffenheit meines Algorithmus spielt es keine Rolle, ob Sendemasten außerhalb der EU zugelassen werden, denn ich wähle im zweiten Durchlauf zufällig aus den Landpixeln aus. Eine Auswahl aus allen Pixeln wäre an dieser Stelle zu ineffizient.

Eine komplette Bedeckung der Karte mit dem hexagonalen Muster ergibt in der Regel eine höhere Anzahl von Sendemasten als mein Ansatz mit den zwei Durchläufen.

Quellcode

Listing 3: eu-wan.rb

```
require "RMagick"
include Magick

class Map
    attr_reader :masten

    # Lies die Karte aus der pbm-Datei in ein zweidimensionales Array
    # ein und setze Höhe und Breite.
    def initialize
        type,comment,size,*data=IO.read("map.pbm").split("\n")
        @width, @height = size.split
        @width, @height = @width.to_i, @height.to_i
        @map = []
        data = data.join
        @height.times do |line|
            @map << data.slice!(0,@width).split("").collect{|x| x=="1
               "}
        end
        @masten = []
        lies_bereich
    end

    # Lies den Sendebereich aus einer anderen Datei. Wandle das
    # resultierende zweidimensionale Array in ein eindimensionales
    # um, um es später leichter benutzbar zu machen.
    def lies_bereich
        type,size,*data=IO.read("circle.pbm").split("\n")
        @w, @h = size.split
        @w = @w.to_i
        @h = @h.to_i
        bereich = []
        data = data.join
        @h.times do |line|
            bereich << data.slice!(0,@w).split("").collect{|x| x=="1"
               }
        end
        @sendebereich=[]
        @h.times do |y|
            @w.times do |x|
                @sendebereich << [x-25,y-25] if bereich[x][y]
            end
        end
    end

    # Bedecke die Karte in einem regelmäßigen, hexagonalen Muster mit
    # Sendemasten. 'vertikal' und 'horizontal' beschreiben den
    # Abstand der Masten zueinander, die horizontale 'verschiebung'
    # wird auf jede zweite Reihe angewandt. Zusätzlich wird das
    # gesamte Muster noch zufällig verschoben.
    def muster
        horizontal=40
        vertikal=43
        verschiebung=22
```

```
        versatz=false
        dx=-rand(50)
        dy=-rand(50)
        (@width/horizontal+2).to_i.times do |mx|
            (@height/vertikal+2).to_i.times do |my|
                x=horizontal*mx+dx
                y=vertikal*my+dy
                y+=25
                y+=verschiebung if versatz
                if land?(x,y) and x>=0 and y>=0
                    mast! x,y
                end
            end
            versatz = !versatz
        end
    end

    # Fülle die verbleibende Fläche zufällig mit Sendemasten, solange
    # bis jeder Pixel besendet wird.
    def rest
        l = land
        until l==0
            zufalls_turm
            l=land
            puts "Noch #{l} Pixel"
        end
    end

    # Such einen Landpixel zufällig aus und zähle die Anzahl der
    # besendeten Pixel, wenn dort ein Mast stehen würde.
    # Wiederhole das x Mal und führe den besten Vorschlag aus.
    def zufalls_turm
        vorschlaege = []
        50.times do
            punkt = @land[rand(@land.size)]
            land = zaehle punkt[0],punkt[1]
            vorschlaege << [land, punkt]
        end
        punkt = vorschlaege.sort_by{|v| v[0]}.last
        mast! punkt[1][0], punkt[1][1]
    end

    # Zeichne die Sendegebiete auf der Karte ein.
    def draw
        karte = ImageList.new("map.pbm")
        kreise = Draw.new
        kreise.stroke_width(1)
        kreise.stroke("red")
        kreise.fill("transparent")

        @masten.each do |t|
            kreise.circle t[0], t[1], t[0]+25, t[1]
        end
        kreise.draw(karte)
        karte.display
    end
```

```
# Platziere einen Sendemasten an der Position x,y
def mast! x,y
    @masten << [x,y]
    sende(x,y) do |sx,sy|
        set(sx,sy,false) if land? sx,sy
    end
end

# Ermittle die Anzahl der Landpixel, die ein Sendemast an
# Position x,y abdecken würde.
def zaehle x,y
    l=0.0
    sende(x,y) do |sx,sy|
        if land? sx,sy
            l+=1
        end
    end
    l
end

# Setzte den Wert an Stelle x,y auf 'value'.
def set x,y,value
    return false if x<0 or y<0 or x>=@width or y>=@height
    @map[y][x]=value
end

# Befindet sich an der Stelle x,y unbesendetes Land?
def land? x,y
    return false if x<0 or y<0 or x>=@width or y>=@height
    @map[y][x]
end

# Füge die Landpixel dem Array '@land' hinzu gib die Anzahl
   zurück.
def land
    @land = []
    each_map do |x,y|
            @land << [x,y] if land? x,y
    end
    @land.size
end

# Rufe für jeden besendeten Punkt vom Sender an der Stelle x,y
# den Block auf.
def sende x,y
    @sendebereich.each do |p|
        yield x+p[0], y+p[1]
    end
end

# Rufe für jeden Pixel der Karte den Block auf.
def each_map
    @width.times do |x|
        @height.times do |y|
            yield x,y
```

```
            end
        end
    end
end

m = Map.new

m.muster
m.rest

puts "Positionen der Masten:"
p m.masten
puts "Benötigte Masten: #{m.masten.size}"
m.draw
```