27. Bundeswettbewerb Informatik

Lösungen der 1. Runde

Sebastian Morr

16. November 2008

Inhaltsverzeichnis

Allgemeine Hinweise **1**

Aufgabe 3: Alle Alpen **2**

Aufgabe 2: Tankomatik **5**

Aufgabe 5: Bürgerampel **12**

Allgemeine Hinweise

In dieser Arbeit werden die Aufgaben 2, 3 und 5 gelöst.

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

Die Programme haben - bis auf Ruby selbst - keine externen Abhängigkeiten und können mit dem Befehl `ruby [programm]` ausgeführt werden. Entwickelt und auf Funktionsfähigkeit überprüft wurde mit dem Standardinterpreter[1] in Version 1.8.7, auf einer x86-Architektur unter GNU/Linux, Kernel 2.6.27.

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 3: Alle Alpen

Teilaufgabe 1

Ein Gebirgszug ist, wie in der Aufgabenstellung angegeben, durch eine Folge von Zahlen hinreichend definiert. Eine Datenstruktur, die einen Gebirgszug darstellt, muss also in der Lage sein, beliebig viele ganze Zahlen in einer festen Reienfolge zu speichern.

Ruby besitzt eine Klasse `Array`, die diese Anforderungen erfüllt und zudem Methoden mitliefert, um die Folge zu verlängern und auszuwerten. Zur Bearbeitung der Aufgaben waren insbesondere die Methoden `size`, `+[]`, `max` und `last` hilfreich.

Ich entschloss mich dagegen, eine Klasse `Gebirgszug` zu schreiben, da diese nur eine Art „Wrapper“ um die Array-Klasse wäre und (in diesem Rahmen) keine Vorteile bieten würde. Für ihr Spiel sollte die Schülergruppe allerdings eine eigene Klasse schreiben, die die Gültigkeit des Gebirgszuges sicherstellt.

Der in der Aufgabe als Beispiel angegebene Gebirgszug würde in Ruby wie folgt erzeugt:

```
gebirgszug = [0, 1, 1, 2, 3, 2, 2, 1, 0]
```

Teilaufgabe 2

Meine Zeichenprozedur nimmt einen Gebirgszug entgegen und stellt ihn unter Zuhilfenahme von ASCII-Zeichen dar. Ich entschied mich für diese Darstellungsweise, weil das Programm so keine externen Abhängigkeiten benötigt, um etwa eine Grafik zu zeichnen, die Ausgabe äußerst schnell erzeugt wird, in Textdateien gespeichert werden kann und in meinen Augen eine anschauliche und übersichtliche Möglichkeit bietet, sich einen Eindruck vom Verlauf des Gebirgszuges zu machen. Die Prozedur findet sich im Quelltext am Ende dieses Abschnitts.

```
zeichne.call [0, 1, 0, 1, 2, 3, 3, 4, 3, 3, 2, 2, 1, 2, 3, 3, 4, 5,
   5, 4, 4, 4, 5, 4, 3, 2, 2, 2, 2, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1,
   2, 2, 3, 4, 3, 4, 3, 3, 3, 4, 3, 3, 4, 5, 4, 3, 3, 2, 3, 2, 2, 1,
   0, 1, 2, 1, 2, 3, 3, 2, 1, 2, 1, 2, 3, 2, 3, 2, 3, 4, 5, 5, 5, 6,
   5, 6, 6, 6, 6, 5, 5, 5, 5, 5, 4, 4, 4, 3, 2, 1, 0]
```

erzeugt folgende Ausgabe:

```
                                                                                     ___
                 _                                                              __/\/   \____
                / \__/\                             /\                         /             \__
     _/\_     _/       \                  /\/\__/\_/  \_           _          /                 \
    /    \_  /          \___            _/              \/\_      / \    /\/\/                   \
   /       \/               \_        _/                    \  /\/   \/\/                         \
/\/                           \_/\___/                       \/                                    \
```

Teilaufgabe 3

Lösungsidee

Aus der Definition für einen Gebirgszug lassen sich folgende Regeln für den Aufbau ableiten:

- Ein Höhenwert h_x kann entweder um 1 größer, um 1 kleiner oder gleich h_{x-1} sein.
- Die Höhe darf sich (im Vergleich zum vorherigen Wert) verringern, solange sie dabei nicht unter 0 fällt.
- Außerdem muss das Ende des Zugs sich wieder auf der Höhe 0 befinden, daher darf zu keinem Zeitpunkt $h_{N-x} > x$ gelten, andernfalls kann die Bedingung $h_N = 0$ auch bei durchgehendem Gefälle nicht mehr erfüllt werden.

Der Kern des Algorithmus muss also alle möglichen direkten Folgeelemente für eine gegebene Höhe ermitteln und dabei diese drei Regeln beachten. Die benötigten Werte für diesen Schritt sind:

- Der „momentane“ Höhenwert, von dem aus der Gebirgszug fortgesetzt werden soll.
- Die verbleibende Anzahl von Werten, bis die Höhe des Zuges wieder 0 betragen soll.

Um den Algorithmus einfach zu halten, plante ich ihn rekursiv: Seine Parameter bestimmen die Länge N, die der zu erzeugende Gebirgszug haben soll, die Prozedur P, für jeden kompletten Zug aufgerufen wird, sowie die bisher generierten Werte in einer Liste. Zu Beginn gibt es nur einen generierten Wert, die festgelegte Ausgangshöhe 0.

Mein Algorithmus prüft für den übergebenen unfertigen Gebirgszug, ob der folgende Höhenwert größer, kleiner, oder gleich dem letzten sein darf und ruft sich für alle verbleibenden Möglichkeiten rekursiv wieder auf. Ist die Restlänge Null, wurde ein kompletter Gebirgszug generiert und die Prozedur P wird aufgerufen.

Programmdokumentation

Der eigentliche Algorithmus wurde als Methode, die Zeichen- und Zählfunktionen als `Proc`-Objekte realisiert. Diese verhalten sich wie gewöhnliche Objekte, enthalten aber ausführbaren Code und können bei Bedarf mit dem fertigen Gebirgszug als Parameter aufgerufen werden.

Quelltext

Listing 1: alle_alpen.rb

```
# Algorithmus, der für alle Gebirgszüge der Länge n die Prozedur
# p aufruft. 'vor' enthält den bisherigen Gebirgszug, wird
# anfangs auf Null gesetzt. Der letze Wert dieses Arrays ist die
# Höhe, von der aus der Gebirgszug vollendet werden soll. Wenn
# die Länge 0 beträgt, ist der Gebirgszug fertig, rufe Prozedur p
# auf und verlasse die Funktion.
def alle_alpen n, p, vor = [0]
    h = vor.last
    (p.call vor; return) if n == 0
    alle_alpen n-1, p, vor + [h-1] if h > 0
    alle_alpen n-1, p, vor + [h] if h < n
    alle_alpen n-1, p, vor + [h+1] if h < n-1
end

# Prozedur, die "gebirge" mit ASCII-Zeichen darstellt.
# Sie legt zunächst ein Array leerer Strings an, die so groß sind
# ist, dass der Gebirgszug später komplett hinheinpasst. Dann
# wird das Gebirge Wert für Wert durchlaufen und ein ASCII-
# Zeichen an der entsprechenden Stelle in die Strings
# geschrieben.
zeichne = Proc.new do |gebirge|
    ausgabe=Array.new(gebirge.max + 1){" " * (gebirge.size - 1)}
    (gebirge.size - 1).times do |i|
        h = gebirge[i]
        case gebirge[i+1] - h
        when 1: ausgabe[h][i] = "/"
        when 0: ausgabe[h][i] = "_"
        when -1: ausgabe[h-1][i] = "\\"
```

```
        end
    end
    puts ausgabe.reverse
end

# Prozedur zum Zählen, die bei jedem Aufruf $anzahl um 1 erhöht.
zaehle = Proc.new do |gebirge|
    $anzahl += 1
end

puts "Alle Gebirgszüge der Länge 6:"
alle_alpen(6, zeichne)

$anzahl = 0
alle_alpen(16, zaehle)
puts "Es gibt #{$anzahl} Gebirgszüge der Länge 16."
```

Ablaufprotokoll

Das Programm erwartet keine Eingabe. Die Ausgabe stellt die Lösung auf die Teilaufgaben 3a) und 3b) dar.

```
Alle Gebirgszüge
    der Länge 6:
------

____/\

___/\_
    _
___/ \

__/\__

__/\/\
   _
__/ \_
  __
__/  \

   /\
__/  \

_/\___

_/\_/\

_/\/\_
    _
_/\/ \
  _
_/ \__
  -
_/ \/\
   __
_/    \
  ___
_/   \
  _/\
_/   \
  /\
_/  \_
  /\_
_/   \
   _
  / \
_/   \
/\____

/\__/\

/\_/\_
   _
/\_/ \

/\/\__

/\/\/\
   _
/\/ \_
  --
/    \
 _
/ \___
 _
/ \_/\
 _
/ \/\_
 _ _
/ \/ \
 __
/  \__
 __
/  \/\
 ___
/    \_
 ____
/    \

 __/\
/    \

 _/\
/    \_

 _/\_
/    \

  _
 _/ \
/    \
```

```
/\/  \

   /\
/\/  \
 _
/ \___
 _
/ \_/\
 _
/ \/\_
 _ _
/ \/ \
 __
/  \__
 __
/  \/\
 ___
/    \_

 __/\
/    \

 _/\
/    \_

 _/\_
/    \

  _
 _/ \
/    \
```

```

  /\
/  \__

  /\
/  \/\

  /\_
/    \_

  /\__
/     \

  /\/\
/     \
  _
  / \
/    \_
  _
  / \_
/     \
  __
  /  \
/     \

  /\
  /  \
/     \
Es gibt 853467
   Gebirgszüge
   der Länge 16.
```

Aufgabe 2: Tankomatik

Teilaufgabe 1

Lösungsidee

Eine wirklich „realistische“ Preisentwicklung zu simulieren, dürfte extrem schwierig sein. Es spielen viele Faktoren eine Rolle (und eine „Börsensimulation“ erwarte ich eher in Runde 2!). Davon abgesehen spielt der Kaufpreis für das Kundenverhalten letztendlich nur eine untergeordnete Rolle, da die Tankstellen den Preis jeweils zum gleichen Zeitpunkt abfragen, wesentlich ist also der relative Preisunterschied. Nach einer Analyse der Entwicklung des Benzinpreises der letzten Jahrzehnte glaube ich jedoch gewisse wiederkehrende Muster entdeckt zu haben:

Grundlage bildet der aktuelle Durschnittspreis von 144,2 ct[2]. Für die Preisentwicklung setze ich eine leichte Inflationsrate von 2,5 Prozent voraus, die Schwankungen simuliere ich durch mehrere Sinusfunktionen. Eine bildet den Trend ab, dass Benzin im Frühjahr und im Herbst besonders teuer ist, muss also eine Periode von einem halben Jahr haben. Zwei weitere sorgen für Preisschwankungen in einem größeren Rahmen (mit einer Periodenlänge von vier Jahren) und in kleinem Rahmen (Periodenlänge von einer Woche). Schließlich gab ich der Funktion noch eine Zufallskomponente, denn nicht alle Faktoren, die den Preis beeinflussen, sind zyklisch und vorhersehbar, etwa Geschehnisse auf dem Weltmarkt oder der Politik, Preiskämpfe zwischen verschiedenen Ölmärkten oder einfach das Wetter. Die resultierende Gesamtfunktion kann im Quelltext eingesehen werden.

In dem Fall, dass die Preise beider Tankstellen exakt übereinstimmen sollten, entscheidet sich der Kunde zufällig, wie er es im echten Leben wohl auch tun würde. Anfangs setze ich den Preis beider Tankstellen auf 5 ct über dem Einkaufspreis, um sie vom gleichen Wert her starten zu lassen.

Die Simulation ist zudem dahingehend abstrahiert, dass jeder Kunde gleich viel tankt (60 Liter). Bei einer derartig hohen Anzahl an Kunden ist es überflüssig, verschiedene Tankgrößen zu betrachten, da sich die Ergebnisse sowieso auf den Erwartungswert einpendeln würden.

Programmdokumentation

Bei der Implementierung ging es in dieser Aufgabe vor allem um eine exakte Umsetzung der Vorgaben. Ich wählte einen objektorientierten Ansatz, hauptsächlich um den Quelltext so lesbar wie möglich zu halten. Die drei Partner, der Ölmarkt sowie die beiden Tankstellen, haben jeweils eine eigene Klasse erhalten. Wesentlich sind jeweils die `tick`-Methoden, die von der Simulation später in jeder Sekunde der Simulation aufgerufen werden, und die `tue_alle`-Methode, die einen Codeblock in gegebenen Zeitabständen wiederholt aufruft.

Das Programm erwartet keine Eingabe. Auch wenn verschiedene Faktoren, die die Simulation beeinflussen würden, aus der Standardeingabe oder einer Konfigurationsdatei hätten gelesen werden können, erschien mir dies zur Beantwortung der Fragen unnötig.

Quelltext

Listing 2: tankomatik.rb

```
include Math

# Hilfsmethode, um eine Kommazahl auf eine bestimmte Anzahl von
    Stellen zu runden.
```

[2] http://www.aral.de/toolserver/retaileurope/annualstatement.do

```
class Float
    def runden stellen=0
        faktor = (10**stellen).to_f
        (self*faktor).round/faktor
    end
end

# Eine Stunde hat 3600 Sekunden, ein Tag 86400, ein Jahr 31536000.
$h=60*60.0
$d=$h*24
$a=$d*365

# Klasse für den Spotmarkt Öl.
class Oelmarkt
    attr_reader :preis

    # Zur Initialisierung bestimme den Preis für die Zeit 0.
    def initialize
        tick 0
    end

    # Bei jedem Aufruf wird der Ölpreis neu bestimmt, die
       Preisschwankungen werden dabei durch Sinus- und
       Zufallsfunktionen simuliert.
    def tick zeit
        # Durchschnittspreis für Benzin in Cent
        @preis = 144.2 +
            # Inflation pro Jahr
            2.5*zeit/$a +
            # Im Frühjahr und Herbst steigen die Preise
            10*sin(2*PI*zeit/($a/2)) +
            # Und etwa alle vier Jahre gibt es einen Preisabfall
            20*sin(2*PI*zeit/(4*$a)) +
            # Preisschwankung pro Woche und pro Tag
            2*sin(2*PI*zeit/(7*$d)) +
            sin(2*PI*zeit/$d) +
            # ...und zu guter letzt ein wenig Zufall.
            rand
    end
end

# Elternklasse der beiden Tankstellen.
class Tankstelle
    attr_reader :preis, :kunden, :gewinn

    # Setze statistische Werte auf Null und frage das erste Mal den
       Preis ab.
    def initialize oelmarkt
        @oelmarkt = oelmarkt
        preis_abfragen
        @preis = @kaufpreis + 5
        @kunden = @gewinn = 0
        @aktionen = {}
    end

    def preis_abfragen
```

```
        @kaufpreis = @oelmarkt.preis
    end

    # Ein Kunde kommt! Erhöhe den Zähler und verkaufe 60 Liter zum
      aktuellen Preis.
    def kunde!
        @kunden += 1
        @gewinn += (60*@preis).to_i
    end

    # Prüfe in jeder Sekunde, ob es Aktionen gibt, deren Intervall
      verstrichen ist, und führe diese aus.
    def tick zeit
        @aktionen.each do |intervall, block|
            block.call if zeit%intervall == 0
        end
    end

    # Füge eine neue Aktion für ein gervall hinzu.
    def tue_alle intervall, &block
        @aktionen[intervall] = block
    end
end

class Asso < Tankstelle
    # Tankstelle Asso benötigt einen Zähler für die Kunden in der
      letzten Stunde, sie fragt alle 6h den Preis ab und legt ihn
      jede Stunde neu fest.
    def initialize oelmarkt
        super
        @kunden_zwischenstand = 0
        tue_alle($h){ preis_festlegen }
        tue_alle(6*$h){ preis_abfragen }
    end

    # Bestimme die Anzahl der Kunden seit dem letzten Aufruf und
      verändere den Preis gemäß der Vorschriften.
    def preis_festlegen
        kunden_differenz = @kunden - @kunden_zwischenstand
        @kunden_zwischenstand = @kunden
        @preis += 1 if kunden_differenz > 35
        @preis -= 1 if kunden_differenz < 25
        @preis = @kaufpreis if @preis < @kaufpreis
    end
end

class Scholl < Tankstelle
    # Tankstelle Scholl fragt alle 6h den Preis ab und legt ihn dann
      gleich neu fest.
    def initialize oelmarkt
        super
        tue_alle(6*$h){ preis_abfragen; preis_festlegen }
    end

    def preis_festlegen
        @preis = @kaufpreis + 5
```

```
    end
end

class Simulation
    # Erzeuge alle benötigten Objekte.
    def initialize
        @spotmarkt = Oelmarkt.new
        @asso = Asso.new @spotmarkt
        @scholl = Scholl.new @spotmarkt
        @zeit = 0
    end

    # Lass jede Sekunde ein Auto kommen, informiere dann die drei
       Beteiligten.
    # Gib außerdem jede Stunde die Zeit und die aktuellen Preise aus.
    def starten
        puts "# Zeit in s, Spotmarkt, Asso, Scholl (Preis in ct)"
        loop do
            auto_kommt
            [@spotmarkt, @asso, @scholl].each{|i| i.tick(@zeit)}
            puts "#{@zeit} #{@spotmarkt.preis.runden(1)} #{@asso.
               preis.runden(1)} #{@scholl.preis.runden(1)}" if @zeit
               % $h==0
            @zeit += 1
        end
    end

    # Simuliere das Verhalten eines Kunden gemäß den Vorschriften.
    def auto_kommt
        if rand(60) == 0
            # Ein Auto will tanken!
            case rand(5)
            when 0..2
```

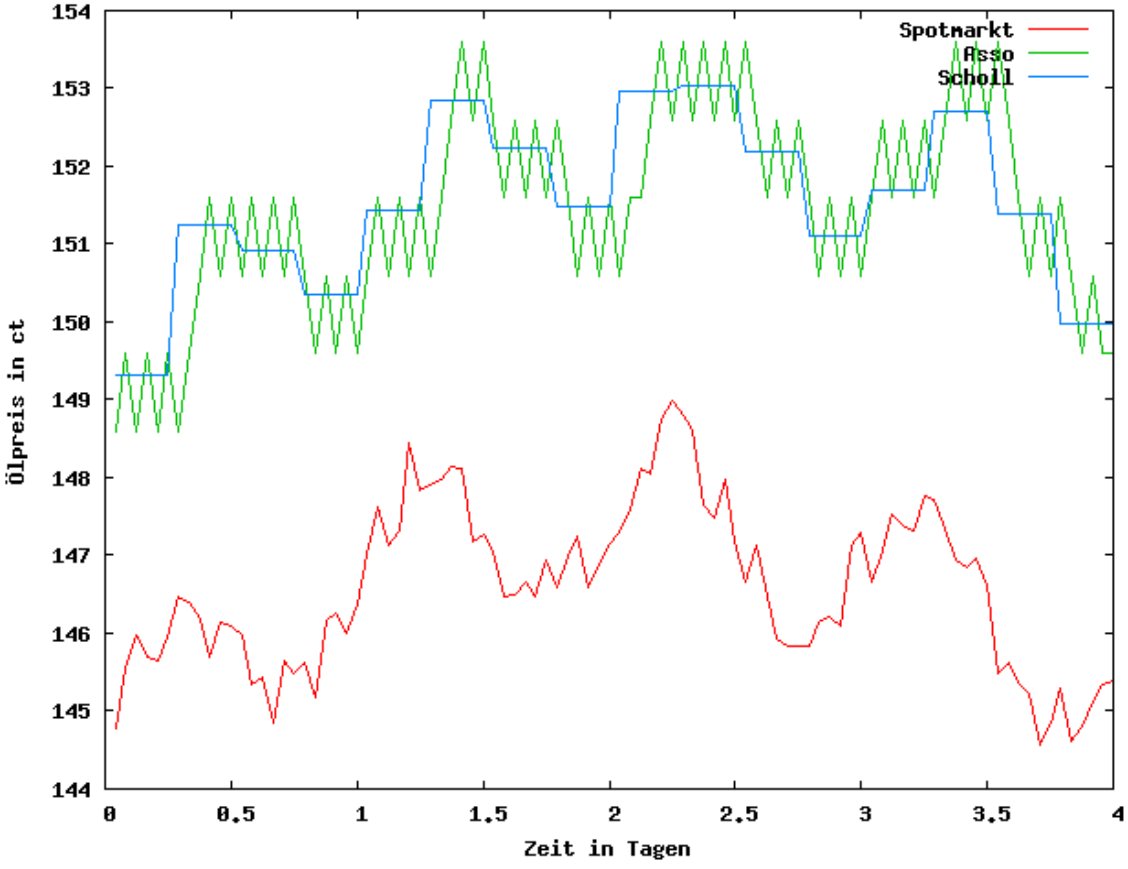
Ablaufprotokoll

Ein beispielhafter Ablauf:

Listing 3: ruby tankomatik.rb

```
# Zeit in s Spotmarkt    Asso         Scholl (Preise in ct)
0           144.2         148.3        149.2
3600        145.1         149.3        149.2
7200        145.9         148.3        149.2
10800       145.7         149.3        149.2
14400       145.7         148.3        149.2
18000       146.6         149.3        149.2
21600       146.4         148.3        151.4
25200       146.3         149.3        151.4
28800       146.4         150.3        151.4
32400       146.2         151.3        151.4
36000       146.4         152.3        151.4
39600       146.1         151.3        151.4
43200       145.7         152.3        150.7
46800       145.3         151.3        150.7
50400       145.3         150.3        150.7
54000       144.9         151.3        150.7
57600       144.8         150.3        150.7
61200       145.3         151.3        150.7
64800       145.8         150.3        150.8
```

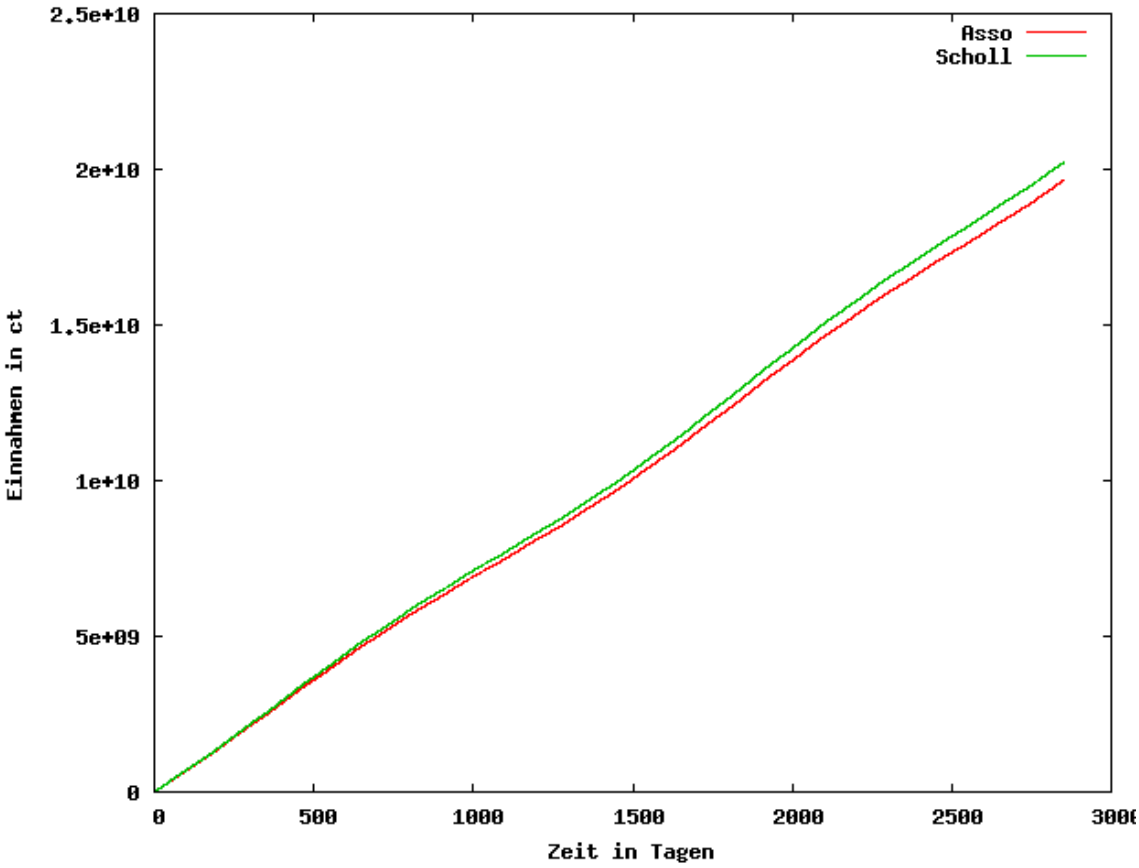
Man sieht sehr schön, wie Asso sich mit häufigen Preissteigerungen- und sekungen um Scholls Preis herumbewegt. Die Ausgabe ist geeignet, um mit GNUPLOT grafisch dargestellt zu werden, hier ein etwas längeres Beispiel:



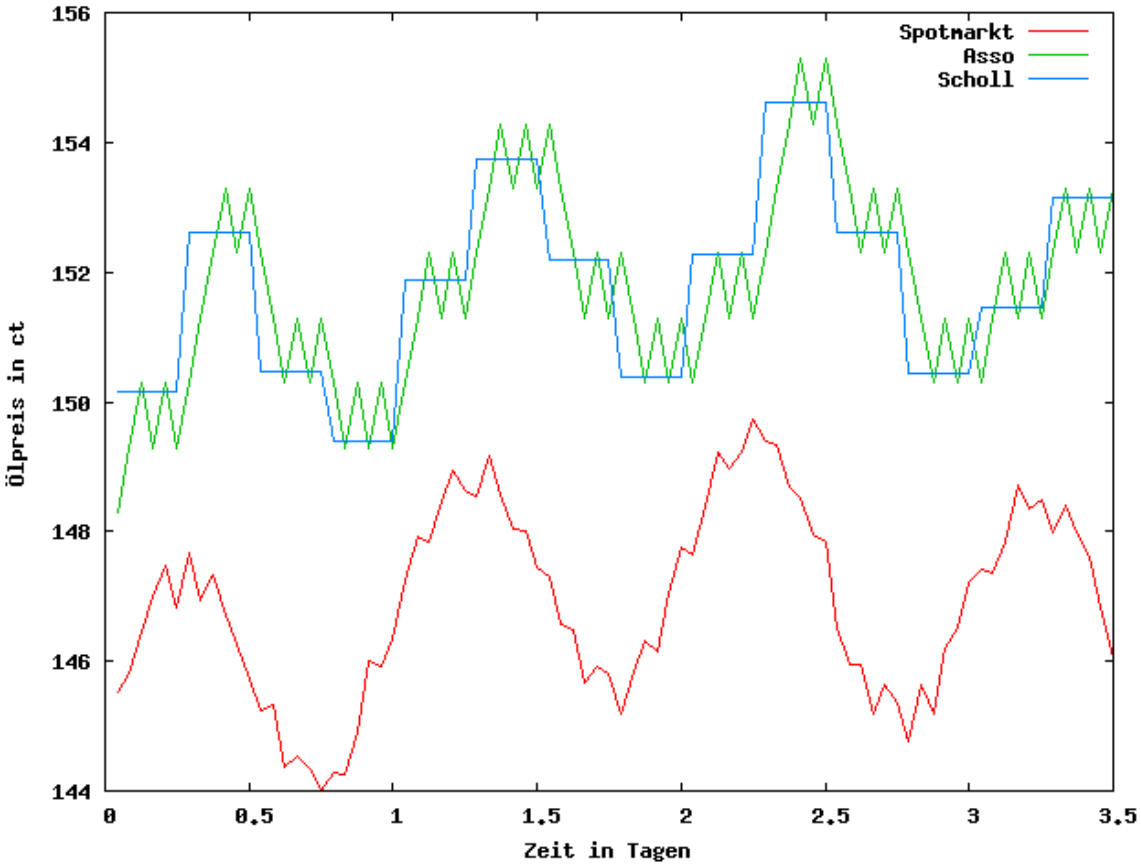
Teilaufgabe 2

Da – in diesem Modell genauso wie in der Wirklichkeit – der Zufall eine große Rolle spielt, fand ich die Antwort auf diese Aufgabe eher empirisch.

Nach vielen Testläufen kam ich zu dem Ergebnis, dass Scholl über einen längeren Zeitraum gesehen mehr Gewinn macht. Da beide Tankstellen ihr Benzin zum selben Zeitpunkt und zum selben Preis kaufen und somit gleich hohe Ausgaben haben, reicht es aus, die Einnahmen zu vergleichen. Folgende Grafik zeigt die Entwicklung der Einnahmen über einige Jahre:



Auf der Suche nach Erklärungsansätzen stieß ich auf folgendes, wiederkehrendes Muster: Wenn der Spotmarkt den Preis stark senkt, kann sich Scholl unmittelbar anpassen. Asso hingegen hat eine begrenzte „Preissenkgeschwindigkeit“ von einem ct pro Stunde, wodurch sein Preis nun für längeren Zeitraum über dem Scholls liegt, in der folgenden Grafik ist dies sichtbar:



Zu erkennen ist auch, dass der Preis von Asso insgesamt öfter über dem von Scholl liegt, was ihm letztendlich weniger Gewinne einbringt.

Teilaufgabe 3

Die Betreiberin von Asso hat mehrere Möglichkeiten, mehr Gewinne zu machen. Scholls Strategie ist Asso nicht bekannt, sie kennt jedoch ebenfalls die in der Aufgabe gegebenen Wahrscheinlichkeiten zum Verhalten der Kunden. In einer Stunde tanken insgesamt durchschnittlich 60 Autos. 20% davon (12) kann sie fest einplanen, wichtig ist es nun, die Aufteilung der 36 „unentschlossenen“ Kunden zu betrachten: Kommen weniger als die Hälfte davon zu ihr (ich beziehe mich hier stets auf die durchschnittlich zu erwartenden Kunden), scheint ihr Konkurrent einen besseren Preis zu haben, sie sollte also ihren Preis schon bei weniger als $12 + \frac{36}{2} = 30$ Kunden senken und ihn erst wieder heben, wenn die unentschlossenen Kunden fast alle zu ihr kommen.

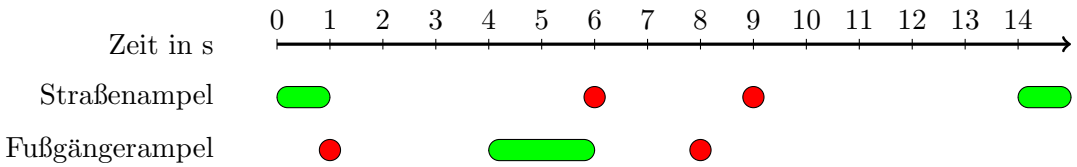
Die optimale Strategie wäre, den Preis stets auf 4 ct über dem Einkaufspreis zu setzen (oder 3 oder 4,9, auf jeden Fall weniger als 5). Assos Benzin wäre dann *zu jedem Zeitpunkt* billiger als Scholls, 80% der Kunden wären ihm sicher! Der Betreiberin ist diese Grenze von 5 ct nicht bekannt, aber auch wenn sie nur vermutet, dass Scholl eine solche Strategie anwendet, könnte sie ein neues Vorgehen entwickeln:

- Zunächst verlangt sie nur den Einkaufspreis.
- Wenn sie mehr als 35 Kunden hatte, setzt sie den Preis das nächste Mal auf auf den Einkaufspreis + 1 ct und erhöht im Folgenden ihren „Preisaufschlag“ stets um 1 ct.
- Wenn sie weniger als 25 Kunden hatte, verringert sie ihren Aufschlag um 1 ct.

Diese Strategie funktioniert sogar, wenn Scholl seinen Preisaufschlag ändert, nicht aber, wenn dieser eine andere Strategie wählt.

Aufgabe 5: Bürgerampel

Teilaufgabe 1



Die grünen Streifen bedeuten, dass die Ampel im jeweiligen Zeitraum grün ist. Der rote Kreis steht für eine Bedarfsanmeldung.

Das Diagramm demonstriert alle drei Vorschriften: In Sekunde 1 wird Schaltungsablauf 1 in Gang gesetzt, in Sekunde 6 Schaltungsablauf 2. In den Sekunden 8 und 9 werden die Abläufe jeweils durch Anmeldung eines Bedarfs unterbrochen.

Teilaufgabe 2

Um die Lesbarkeit der folgenden Teilaufgaben zu verbessern, sei folgende Sprachregelung vorgegeben: Wenn „ein Fußgänger kommt“, wird Fußgängerbedarf angemeldet und wenn „ein Auto kommt“ Straßenbedarf.

Testfälle

Folgende Tests wurden an jeder Implementierung vorgenommen, sie überprüfen die in der Aufgabenstellung geforderten Vorschriften:

1. Ein Fußgänger kommt an eine rote Ampel. Der Test ist erfolgreich, wenn die Straßenampel sofort auf rot schaltet und nach 3 Sekunden die Fußgängerampel grün wird.

2. Ein Auto kommt an eine rote Ampel. Der Test ist erfolgreich, wenn die Fußgängerampel sofort rot wird und nach 5 Sekunden die Straßenampel grünes Licht gibt.

3. Bei roter Fußgängerampel kommt ein Fußgänger und sofort danach ein Auto. Es sollte die Straßenampel rot, nach 5 Sekunden jedoch wieder grün werden.

4. Bei roter Straßenampel kommt ein Auto, unmittelbar gefolgt von einem Fußgänger. Es sollte die Fußgängerampel rot, nach 3 Sekunden jedoch wieder grün werden.

Die Testfälle sind auch dann erfolgreich, wenn eine Ampel, die zu Beginn eines Tests rot werden soll, schon vor Beginn des Tests rot ist.

Über die Vorschriften hinausgehende Funktionalität wird nicht als Fehler gewertet. Die folgenden Abschnitte beschreiben fehlgeschlagene Tests und Besonderheiten.

Methode 1

Test 1 ist erfolgreich, nach einigen Sekunden wird zusätzlich Schaltungsablauf 2 gestartet. Auch bei Test 4 wird nach einigen Sekunden Schaltungsablauf 2 ausgeführt.

Methode 2

Test 3 schlägt fehl: die Fußgängerampel wird nach 3 Sekunden kurz grün, danach beginnt Schaltungsablauf 2. Auch Test 4 schlägt fehl: Schaltungsablauf 2 wird nicht unterbrochen, sondern komplett ausgeführt, danach geschieht nicht mehr.

Methode 3

Test 4 entspricht formal den Vorschriften, zeigt jedoch die Problematik, dass die Straßenampel nun nicht mehr grün werden kann, siehe hierzu auch Teilaufgabe 3.

Methode 4

Test 1 und Test 2 sind nicht zu jedem Zeitpunkt erfolgreich: Nachdem eine Ampel grün geworden ist, kann für einige Sekunden kein Schaltungsablauf gestartet werden. Test 3 schlägt fehl, Schaltungsablauf 1 läuft weiter, erst einige Sekunden nachdem die Fußgängerampel grün ist, startet Schaltungsablauf 2. Auch Test 4 schlägt fehl, Schaltungsablauf 1 wird nicht unterbrochen.

Teilaufgabe 3

Die Verkehrbehörde von Pedes hat die Vorschriften nicht gründlich genug durchdacht. Würde die Schaltung tatsächlich wie angegeben umgesetzt, würde das für einige Probleme sorgen:

- Kommen viele Fußgänger in kurzen Zeitabständen an die Ampel, müssen die Autos unter Umständen sehr lange warten, weil Schaltungsablauf 2 ständig unterbrochen wird.
- Die Zeitspanne, in der eine Ampel grün ist, kann beliebig kurz sein. Allein um anzufahren ist aber eine Mindestdauer von mehreren Sekunden notwendig, diese wird in den Vorschriften nicht sichergestellt.
- Laut Aufgabenstellung wird nur in dem Augenblick Straßenbedarf angemeldet, in dem die Induktionsschleife überfahren wird. Wartet ein Fahrzeug vor einer roten Ampel (etwa, weil durch einen Fußgänger Schaltungsablauf 2 unterbrochen wurde), müsste es möglicherweise zurücksetzen, um erneut Bedarf anzumelden, was ihm durch hinter im stehende Fahrzeuge unmöglich gemacht werden könnte.

Methode 4 der Implementation behebt diese Probleme, indem eine Mindestdauer festgelegt wird, die eine Ampel grün sein muss. In diesem Zeitraum wird eine Bedarfsanmeldung zurückgestellt und später durchgeführt. Dies scheint (nach meinen intensiven Versuchen mit dem lokalen Ampelsystem) auch in Wirklichkeit sichergestellt zu sein.