31. Bundeswettbewerb Informatik, 1. Runde

Lösungshinweise und Bewertungskriterien

Allgemeines

Das Wichtigste zuerst: Wir haben uns sehr darüber gefreut, dass diesmal besonders viele sich die Mühe gemacht und die Zeit zur Bearbeitung der Aufgaben genommen haben.

Natürlich waren nicht alle Einsendungen perfekt, und einige eher äußerliche Anforderungen wurden häufiger missachtet. Im Einzelnen:

- Die Beschreibungen von Lösungsidee und Umsetzung, insbesondere aber auch ausreichend viele Beispiele (mindestens die in der Aufgabenstellung geforderten) und der Quelltext (bis auf unwichtige Teile) müssen ausgedruckt sein. Aus Zeit- und Kostengründen ist es unmöglich, vor der Bewertung erst einmal alle Einsendungen auf Vollständigkeit zu prüfen, auf den Datenträgern ggf. nach weiterem Material zu suchen und das dann auszudrucken.
- Beispiele werden immer erwartet. Zu wenige Beispiele und erst recht die Nichtbearbeitung vorgegebener Beispiele führen zu Punktabzug. Es ist auch nicht ausreichend, Beispiele nur auf CD abzugeben, ins Programm einzubauen oder den Bewertern das Erfinden und Testen von Beispielen zu überlassen. Ohne abgedruckte Beispiele ist die Bewertung einer Lösung in der knappen vorhandenen Zeit nicht möglich. Leider fehlten in vielen Einsendungen die Beispiele, was oft das Erreichen der zweiten Runde verhindert hat.
- Zu einer Einsendung gehören auch lauffähige Programme. Kompilierung von Quellcode ist während der Bewertung nicht möglich. Für die gängigsten Skript-Sprachen stehen Interpreter zur Verfügung. Ohne die Abgabe eines eigenständig ausführbaren Programms ist die Bewertung oft schwierig. Einige Entwicklungsumgebungen (z.B. BlueJ), bei denen die erstellten Programme ohne Weiteres nur innerhalb der Umgebung selbst laufen, stehen bei der Bewertung zur Verfügung, aber sicher nicht alle.

Vielleicht helfen diese Anmerkungen, wenn Sie (hoffentlich) im nächsten Jahr wieder mitmachen.

Auch die folgenden eher inhaltlichen Dinge sollten Sie beachten:

- Lösungsideen sollten Lösungsideen sein und keine Bedienungsanleitungen oder Wiederholungen der Aufgabenstellung. Es soll beschrieben werden, welches Problem hinter der Aufgabe steckt und wie dieses Problem grundsätzlich angegangen wird. Ein einfacher Grundsatz: Bezeichner von Programmelementen wie Variablen, Prozeduren etc. dürfen nicht verwendet werden – eine Lösungsidee ist nämlich unabhängig von solchen Realisierungsdetails.
- Auch ein Beispiel des Programmablaufs soll keine Bedienungsanleitung sein. Es beschreibt nicht, wie das Programm ablaufen sollte, auch nicht die zum Ablauf nötigen Interaktionen mit dem Programm, sondern protokolliert den tatsächlichen Ablauf eines Programms.
- Oben wurde schon gesagt, dass Beispiele immer dabei sein sollten. Das hat seinen Grund: An den Beispielen ist oft direkt zu sehen, ob bestimmte Punkte korrekt beachtet wurden. Viele meinen nun, wir könnten die Programme ja laufen lassen und selbst auf Beispieldaten ansetzen, und liefern keine Beispiele oder nur Beispieldaten in elektronischer Form. Das können wir aber aus Zeitmangel in der Regel nicht. Außerdem ist nicht immer sicher, dass Programme, die auf dem eigenen PC laufen, auch auf einem anderen Computer ausführbar sind. Generell muss man sich darauf einstellen, dass nur das Papiermaterial angesehen wird!
- Mit den verschiedenen Beispielen sollen wichtige Varianten des Programmablaufs gezeigt werden, also auch Sonderfälle, die die Lösung behandeln kann. Die Konstruktion solcher Testfälle ist eine ganz wesentliche Tätigkeit des Programmentwurfs.

Einige Anmerkungen noch zur Bewertung:

- Pro Aufgabe werden maximal fünf Punkte vergeben, bei Mängeln gibt es entsprechend weniger Punkte. Für die Gesamtbewertung sind die drei am besten bewerteten Aufgabenlösungen maßgeblich, es lassen sich also maximal 15 Punkte erreichen. Einen 1. Preis erreichen Sie mit 14 oder 15 Punkten, einen 2. Preis mit 12 oder 13 Punkten und eine Anerkennung mit 9 bis 11 Punkten. Die Preisträger sind für die zweite Runde qualifiziert.
- In der Juniorliga wird ein 1. Preis für 9 oder 10 Punkte, ein 2. Preis für 7 oder 8 Punkte und eine Anerkennung für 5 oder 6 Punkte vergeben. Leider gibt es in der Juniorliga (noch) keine 2. Runde.
- Im Gegensatz zur Ankündigung sind eine Reihe von Einsendungen sowohl in der Juniorliga als auch im normalen Wettbewerb (auch als „Leistungsliga“ bezeichnet) gewertet worden. Das ist der Fall, wenn alle Gruppenmitglieder die Altersbedingung für Junioraufgaben erfüllen (damit kommt die Einsendung für die Juniorliga in Frage) und in der Einsendung sowohl Junioraufgaben als auch andere Aufgaben bearbeitet wurden.

- Auf den Bewertungsbögen bedeutet ein Kreuz in einer Zeile, dass die (meist negative) Aussage in dieser Zeile auf die Einsendung zutrifft. Damit verbunden ist dann in der Regel der Abzug eines oder mehrerer Punkte. Eine Wellenlinie bedeutet „na ja, hätte besser sein können“, führt aber alleine nicht zu Punktabzug. Mehrere Wellenlinien können sich aber zu einem Punktabzug addieren. Gelegentlich sind lobende Anmerkungen der Bewerter mit einem ’+’ versehen.
- Wellenlinien wurden übrigens häufig für die Dokumentation – also Lösungsidee, Umsetzung, Beispiele und Quelltext – verteilt, obwohl Punktabzug auch gerechtfertigt gewesen wäre.
- Aber auch so ließ sich nicht verhindern, dass etliche Teilnehmer nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden. Das ist ziemlich riskant, da Fehler sich leicht einschleichen.

Zum Schluss:

- Sollte der Name auf der Urkunde falsch geschrieben sein, kann gerne eine neue angefordert werden.
- Es ist verständlich, wenn jemand, der nicht weitergekommen ist, über eine Reklamation nachdenkt. Kritische Fälle, insbesondere die mit 11 Punkten, haben wir allerdings schon genau und mit Wohlwollen geprüft.

Danksagung

An der Erstellung der Lösungsideen haben mitgewirkt: Johannes Pieper (Junioraufgabe 1), Matthias Kemper (Junioraufgabe 2), Robert Czechowski (Aufgabe 1), Martin Thoma (Aufgabe 2), Benito van der Zander (Aufgabe 3), Thomas Leineweber (Aufgabe 4) und Oliver Siebert (Aufgabe 5).

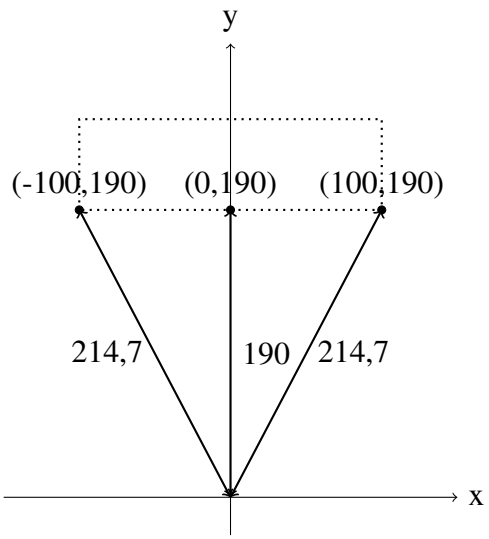
Die Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt, und zwar aus Vorschlägen von Wolfgang Pohl (Junioraufgabe 1 und Aufgabe 3), Torben Hagerup (Junioraufgabe 2 und Aufgabe 1), Michael Gamer und Arno Pasternak (Aufgabe 2), Jens Gallenbacher (Aufgabe 4) und Peter Rossmanith (Aufgabe 5).

Junioraufgabe 1: Skyline

J1.1 Lösungsidee

Die minimale Entfernung eines Gebäudes vom zentralen Bau der Needle ist entscheidend für die maximal zulässige Gebäudehöhe. Um die zulässige Gebäudehöhe bestimmen zu können, reicht es für die meisten Gebäude aus, von den vier Eck-Koordinaten des Grundrisses die Entfernung zur Needle mit Hilfe des Pythagoras-Satzes zu berechnen und die davon die kleinste Entfernung auszuwählen. Dabei wird die Rechnung dadurch vereinfacht, dass die Needle im Ursprung $(0,0)$ des Koordinatensystems liegt.

Ist die kleinste Entfernung bestimmt worden, kann daraus die maximale Höhe berechnet werden. Dazu wird die Entfernung durch 100 dividiert und das Ergebnis abgerundet. Dazu wird 100 addiert und man erhält die maximale Höhe.



Die kleinste Entfernung muss allerdings anders berechnet werden, wenn der Grundriss, wie in der vorstehenden Abbildung dargestellt, auf einer der Koordinatenachsen und damit in zwei Quadranten des Koordinatensystems liegt. In diesem Fall hat einer der beiden Schnittpunkte mit der Achse die geringste Entfernung, nämlich der mit der betragsmäßig kleineren Achsenkoordinate. Um diesen Fall zu entdecken, müssen für alle benachbarten Eckpunkte des Grundrisses deren Koordinaten verglichen werden. Der Fall liegt vor, wenn die y-Koordinaten gleich sind und die x-Koordinaten unterschiedliche Vorzeichen haben – oder umgekehrt.

Theoretisch kann ein Grundriss auch den Nullpunkt enthalten, etwa wenn das Gebäude in allen vier Quadranten stehen soll. Dafür müsste die Needle aber abgerissen werden, und dieses weiß der Rat der Stadt zu verhindern. Im Programm muss dieser Fall daher nicht behandelt werden. Außerdem darf das Programm davon ausgehen, dass die eingegebenen Koordinaten wirklich ein Rechteck beschreiben.

J1.2 Umsetzung

Für das Programm ist es wichtig, dass es für mehrere Gebäude die Koordinaten einliest, für diese das Ergebnis berechnet und in geeigneter Form ausgibt. Dabei ist es zulässig, die Koordinaten nicht aus einer Datei einzulesen, sondern per Hand im Programm einzugeben.

Zur Bestimmung der geringsten Entfernung e_{min} (die in einer Variable gespeichert wird), kann man grundsätzlich so vorgehen, dass nach und nach verschiedene mögliche Entfernungen berechnet werden. Ist ein aktueller Wert geringer als der bisherige Wert von e_{min}, wird e_{min} auf den aktuellen Wert gesetzt. Am Anfang muss e_{min} mit Begründung entsprechend hoch gewählt sein oder automatisch auf die erste Entfernung gesetzt werden.

Zunächst wird die Lage des Grundrisses geprüft. Wir gehen davon aus, dass die Koordinaten der Eckpunkte reihum angegeben sind. Dann muss jedes Koordinatenpaar mit dem vorherigen verglichen und geprüft werden, ob die zugehörige Grundrissseite eine Achse schneidet. Trifft dieser Fall zu, so ist die Entfernung zu dieser Seite der Absolutwert der Komponente, die bei beiden Koordinaten gleich ist, und kann ggf. nach obigem Schema den Wert von e_{min} ersetzen.

Wurde im ersten Schritt festgestellt, dass der Grundriss nicht in zwei Quadranten liegt, wird anschließend für jede Koordinate (x,y) die Entfernung zum Ursprung des Koordinatensystems und damit zur Needle mit Hilfe des Pythagoras $e = \sqrt{x^2+y^2}$ berechnet und ggf. in e_{min} gespeichert.

Wie bereits in der Lösungsidee beschrieben wird schließlich die maximale Höhe berechnet mit $h_{max} = \text{abrunden}(e_{min}/100) + 100$. Diese muss anschließend ausgegeben werden.

J1.3 Beispiele

Als Beispieldaten können zunächst die im Aufgabenblatt angegebenen Grundrisse dienen; sie haben die folgenden Koordinaten und Lösungen:

1. (-80, 100); (-20, 100); (-20, 160); (-80, 160) → 101
2. (60, 160); (120, 160); (120, 250); (60, 250) → 101
3. (-400,200); (-300, 200); (-300, 300); (-400, 300) → 103

Auf jeden Fall müssen mindestens zwei selbst gewählte Grundrissen und die zugehörigen Ergebnisse angegeben sein, darunter einer in zwei Quadranten. Zum Testen der Funktionsweise eignen sich z. B. folgende Grundrisse:

1. (-100, 195); (100, 195); (100, 300); (-100, 300) → 101
2. (195,-100); (195,100); (300,100); (300,-100) → 101

Sollte die Überprüfung der Lage über zwei Quadranten fehlerhaft oder nicht implementiert sein, so wird als Ergebnis jeweils 102 geliefert.

J1.4 Bewertungskriterien

Die Aufgabenstellung verlangt ein Programm, welches in der Lage ist, die maximale Höhe für mehrere durch jeweils vier Koordinatenpunkte angegebene Gebäude zu berechnen.

- Ein Grundriss (ein Rechteck) wird durch zwei diagonal gegenüberliegende Eckpunkte eindeutig festgelegt. Es ist in Ordnung, wenn eine Lösung eine entsprechende Eingabe erwartet.
- Die Berechnung der maximal zulässigen Höhe ist nachvollziehbar beschrieben.
- Der Sonderfall eines Gebäudes, das in mehreren Quadranten steht, muss erkannt worden sein; das Programm kann entsprechende Grundrisse korrekt verarbeiten.
- Das Programm berechnet dabei korrekt die maximale Höhe jedes einzelnen Gebäudes. Dabei soll es nicht zu aufwändig vorgehen; die Berechnung von Entfernungen zu mehr als den Eckpunkten oder Achsenschnittpunkten des Grundrisses ist nicht nötig.
- Es sollen eigene Beispiele (und deren Ergebnisse) angegeben werden; davon mindestens eines, bei dem Eckpunkte in verschiedenen Quadranten liegen.

Junioraufgabe 2: Verben

J2.1 Lösungsidee

In den meisten Fällen ist die Aufgabe des Programms, die Konjugationsendung einer eingegebenen konjugierten Verbform zu erkennen und durch die Infinitivendung »-en« zu ersetzen. Um einen Überblick über die möglichen Endungen zu erhalten, betrachten wir einmal die verschiedenen Formen des Verbs »sagen«:

	Präsens	Präteritum
ich	sage	sagte
du	sagst	sagtest
er/sie/es	sagt	sagte
wir	sagen	sagten
ihr	sagt	sagtet
sie	sagen	sagten

Da wir wie Mehmet im Deutschunterricht gut aufgepasst haben, wissen wir, dass es auch noch verschiedene Konjunktivendungen gibt[1]:

	Konjunktiv I	Konjunktiv II
ich	sage	sagte
du	sagest	sagtest
er/sie/es	sage	sagte
wir	sagen	sagten
ihr	saget	sagtet
sie	sagen	sagten

Die anderen Zeitformen werden mit Hilfsverben gebildet, sodass wir neben diesen Formen nur noch die Partizipien »sagend« und »gesagt« sowie die Imperative »sag«/»sage« und »sagt« berücksichtigen müssen. Nicht zu vergessen ist, dass das Verb schon als Infinitiv »sagen« vorliegen könnte, der allerdings von einer Präsensform nicht zu unterscheiden ist.

Nun muss man nicht alle diese Fälle einzeln betrachten, sondern stellt fest, dass viele Endungen mehrmals auftauchen. Da wir nicht die Form des Verbs bestimmen möchten, sondern nur den Infinitiv finden wollen, können wir alle Fälle zusammenfassen und erhalten die folgende Liste von Endungen:

e test est st te tet et t ten en end

Dabei wurden die Endungen bereits so sortiert, dass Endungen, die selbst hinterer Teil einer anderen Endung sind, weiter rechts stehen. Sollte ein gegebenes Wort eine dieser Endungen

[1] Die Pflichtbeispiele aus der Aufgabenstellung sind auch ohne Konjunktiv zu erklären; deshalb ist eine Beachtung von Konjunktivformen nicht zwingend gefordert.

haben, muss immer die am weitesten links stehende passende Endung weggestrichen werden, da man sonst bspw. der Verbform »sagtest« auch die Endung »t« wegstreichen und den Infinitiv »sagtesen« zuordnen würde.

Das obige Schema funktioniert auch für viele andere Verben als »sagen«, aber ein paar weitere Fälle müssen noch berücksichtigt werden. Am Beispiel »leiten« stellt man fest, dass vor einigen dieser Endungen noch ein »e« stehen kann (vgl. »du leitetest«, »er leitete«, »sie leiteten«). Da dieses im Infinitiv gestrichen werden muss (das Wortende »-een« klingt eher holländisch), kann man ein mögliches »e« am besten getrennt behandeln und braucht nur noch folgende Endungen zu berücksichtigen:

test st te tet t ten n nd

Neben den Endungen muss noch der Wortanfang »ge-« des Partizips entfernt werden. Manchmal steht dieses nicht am Anfang (z. B. bei »weggeholt«), doch gibt es auch viele Verben, die schon im Infinitiv ein »-ge-« enthalten. Da man hier keine einfachen, computer-entscheidbaren Kriterien hat, wollen wir hier nur »ge« am Wortanfang entfernen.

J2.2 Umsetzung

Erste Möglichkeit: Fallunterscheidungen

Eine Möglichkeit zur Implementierung besteht darin, mittels Fallunterscheidungen die verschiedenen Endungen (und das »ge-« am Anfang) zu erkennen und zu entfernen. Dazu sollte eine Programmiersprache benutzt werden, die ausreichende Möglichkeiten zum Bearbeiten von Zeichenketten[2] wie das Prüfen von Anfangs- und Endstücken und Entfernen derselben zur Verfügung stellt. Bei der Überprüfung der Endungen sollte man wie in der Lösungsidee erläutert bei den längsten Endungen beginnen. Sobald festgestellt wurde, dass das gegebene Verb tatsächlich eine gegebene Endung besitzt, wird die entsprechende Teilzeichenkette entfernt und stattdessen ein »en« angehängt (bzw. ein »n«, wenn man mit der kurzen Endungsliste arbeitet und der letzte Buchstabe nach der Entfernung der Endung ein »e« ist). Ein mögliches »ge« am Anfang muss getrennt entfernt werden.

Um nicht für jede Endung eine neue `if`-Verzweigung mit fast gleichem Inhalt schreiben zu müssen, kann man die zu entfernenden Endungen in einer Liste oder einem Array von Zeichenketten verwalten. Dieses kann in einer Schleife abgearbeitet werden.

Zweite Möglichkeit: Reguläre Ausdrücke

Im Wesentlichen muss das Programm zu dieser Aufgabe nur Zeichenketten nach bestimmten Regeln bearbeiten können. Das erfordert keine komplizierte Softwarearchitektur, sondern nur eine sinnvolle Kodierung dieser Regeln. Eine elegante Möglichkeit zu deren Umsetzung

[2] Zeichenketten sind hier Folgen von Buchstaben. Im Allgemeinen können Zeichenketten auch andere Zeichen wie Ziffern, Interpunktionszeichen etc. enthalten.

bietet das genau für solche Probleme entwickelte Konzept der *regulären Ausdrücke* (regular expressions).

In dieser Lösung verwenden wir Java und die zugehörige Implementierung regulärer Ausdrücke. Für die meisten modernen Programmiersprachen findet man Bibliotheken zur Verwendung regulärer Ausdrücke, doch kann sich die Syntax der Ausrücke geringfügig unterscheiden. Eine ausführliche Einführung in reguläre Ausdrücke mit Java findet man in den Java-Tutorials unter `http://docs.oracle.com/javase/tutorial/essential/regex`.

Zur Erkennung der Verben kann man den regulären Ausdruck

```
ENDUNGEN="^(ge)?(.*?)e?(test|st|te|tet|t|ten|n|nd)?$"
```

verwenden. Dann realisiert der Java-Befehl

```
verb.replaceAll(ENDUNGEN, "$2en")
```

die in der Lösungsidee genannten Schritte: Die erste Klammer des Ausdrucks „erkennt“ ein mögliches »ge« am Anfang, der hintere Teil die oben genannten Endungen mit einem möglicherweise vorangehenden »e«. Wichtig ist in der zweiten Klammer der Ausdruck `.*?`, bei dem das Fragezeichen dafür sorgt, dass dieser Klammer ein möglichst kurzer Ausdruck zugeordnet wird, sodass auch längere Endungen (wie »test«) erkannt werden können, auch wenn bereits der hintere Teil hiervon (»t«) eine gültige Endung wäre. Durch die zweite Klammer wird also der Verbstamm identifiziert. Im Java-Befehl wählt das `$2` den der zweiten Klammer zugeordneten Text aus; der Befehl ersetzt also wie gewünscht das Verb durch den um »en« erweiterten Stamm.

J2.3 Beispiele

Die in der Aufgabenstellung vorgegebenen Beispiele müssen korrekt gelöst werden:

Tabelle 1: Beispiele aus der Aufgabenstellung

sagst	sagen	schweigend	schweigen
leitete	leiten	trag	tragen
geforscht	forschen		

Die folgenden Testfälle decken alle verbliebenen Endungen ab und sollten ebenfalls korrekt erkannt werden:

Tabelle 2: weitere Beispiele

glaubtest	glauben	tippten	tippen
meintet	meinen	holen	holen

Probleme hat diese einfache Lösung allerdings, wenn Verben schon im Infinitiv auf »-ten« enden, da das t dann weggestrichen wird (»schreite« → »schreien«), oder wenn sie mit »ge-« beginnen, ohne ein Partizip zu sein. Auch Verben wie »kichern«, die im Infinitiv nicht auf »-en« enden, können nicht richtig zugeordnet werden.

Tabelle 3: Beispiele für die Grenzen des Programms

gebet	ben	arbeite	arbeien
genehmigt	nehmigen	gehört	hören
mitgehalten	mitgehalen	kichern	kicheren

In einigen Fällen kann man hier vielleicht besseres Verhalten erreichen, doch für alle gibt es keine allgemeine Lösung (vgl. »verzehrst« mit »kicherst«). Unregelmäßige Verben scheitern sowieso. Eine allgemein gültige Lösung lässt sich nur durch Verwendung eines Wörterbuchs erzielen.

J2.4 Bewertungskriterien

- Das Verfahren soll nachvollziehbar beschrieben werden; falls reguläre Ausdrücke verwendet werden, sollten diese genauer erläutert werden.
- Es soll erkannt sein, dass es genügt, Wortendungen (plus das »ge« am Anfang eines Partizips) zu erkennen. Die vom Programm betrachteten Endungen der verschiedenen Verbformen sollen in der Dokumentation benannt werden; dabei muss auch die Problematik der Reihenfolge, in der die Endungen zu behandeln sind, angesprochen sein.
- Zumindest die Beispiele aus der Aufgabenstellung und andere Beispiele nach gleichem Muster müssen korrekt gelöst werden, möglichst auch die Beispiele aus Tabelle 2. Es ist akzeptabel, wenn ein Infinitiv oder eine Konjunktivform (einzige spezifische Endung ist »est«) nicht korrekt verarbeitet wird. Nicht akzeptabel ist, wenn die Lösung auf die Beispiele aus der Aufgabenstellung oder eigene Beispiele zugeschnitten ist und bei anderen gleichartigen Verbformen nicht funktioniert.
- Es muss erkannt werden, dass die Lösung beliebige Verbformen nicht korrekt behandeln kann (solange kein Wörterbuch verwendet wird).
- Es sollten ausreichend viele Beispiele angegeben sein, darunter gemäß Aufgabenstellung die vorgegebenen aus Tabelle 1, zusätzliche eigene und einige Gegenbeispiele wie in Tabelle 3, die das Scheitern des Programms zeigen.

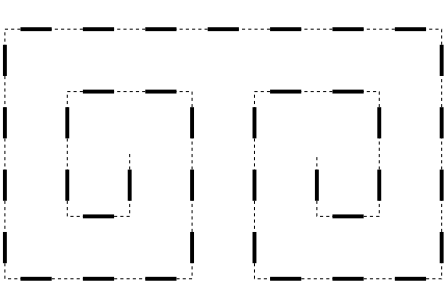
Aufgabe 1: Frühsport

1.1 Problem

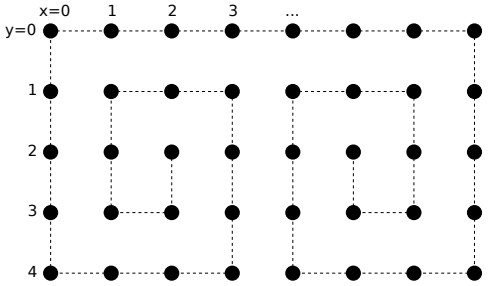
Bei dieser Aufgabe geht es darum, eine Schlange, bestehend aus Ringeln gleicher Länge, auf einem zweidimensionalen Gitter zu entknäueln. Dabei kann die Schlange jeden ihrer ursprünglich goldfarbenen Ringel zu zwei roten Ringeln strecken und umgekehrt zwei rote Ringel zu einem goldfarbenen Ringel zusammenziehen. Zusätzlich kann die Schlange zwei benachbarte verschiedenfarbige Ringel vertauschen und einfache Bewegungen auf dem Gitter ausführen.

1.2 Darstellung der Schlange

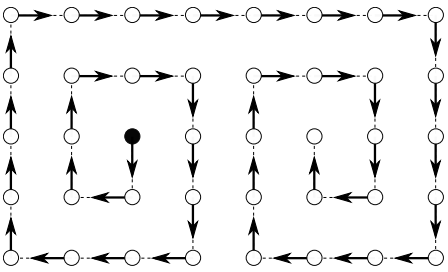
Zunächst sollte man sich die Frage stellen, wie die Schlange im Programm überhaupt dargestellt werden kann. Gleichzeitig – und verwandt damit – stellt sich die Frage, in welchem Format die Schlange eingelesen werden kann.



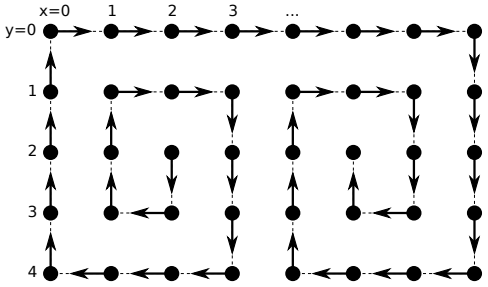
(a) Darstellung als Liste von Segmenten.



(b) Darstellung als Liste von Punkten.



(c) Darstellung als Liste von Richtungen.



(d) Kombination der Darstellungen aus Punkten und Richtungen.

Abbildung 1: Mögliche Arten der Darstellung einer verknäulten Schlange.

Da die Schlange in der Aufgabe aus Segmenten besteht, scheint eine intuitive Darstellung der Schlange einfach aus einer Liste dieser Segmente zu bestehen, wie in Abb. 1(a) dargestellt.

Allerdings stellt man schnell fest, dass diese Darstellung sehr unhandlich ist, da es zwei Arten von Segmenten gibt (horizontal und vertikal) und sich nur schwierig gute Koordinaten für beide Arten von Segmenten einführen lassen.

Sinnvoller ist dagegen eine Darstellung, die nur die Endpunkte der Segmente betrachtet. Dies ist in Abb. 1(b) illustriert. Hier beschreibt das kanonische Koordinatensystem einfach die x- und y-Koordinaten des Punktes in ganzen Zahlen (entsprechend den Segmentlängen der Schlange.)

Man kann auch nur die relativen Änderungen der Koordinaten (also “links”, “rechts”, “hoch”, “runter”) speichern (Abb. 1(c)). Definiert man noch einen Startpunkt – der Kopf der Schlange kann z.B. bei $P_0 = (0,0)$ liegen, so kann man alle anderen Koordinaten aus diesen Angaben berechnen. Alternativ kann man sich die Relativänderungen aus den absoluten Koordinaten herleiten (Abb. 1(d)), weshalb im Folgenden diese Darstellung verwendet wird.

1.3 Lösungsansätze

Wenn man versucht, verschieden verknäulte Schlangen per Hand und Intuition zu entknäueln, so stellt man schnell fest, dass dies immer irgendwie möglich ist. Allerdings wird man dabei ähnlich vorgehen wie beim Entknäueln eines Bindfadens. Das Entknäueln sieht fallabhängig sehr verschieden aus und lässt sich dadurch nur schwer in einen Algorithmus umsetzen.

Es soll also versucht werden, ein Verfahren zu finden, das sich einfach implementieren lässt und dennoch alle verknäulten Schlangen ausreichend effizient entknäueln kann.

Reduktion des Problems

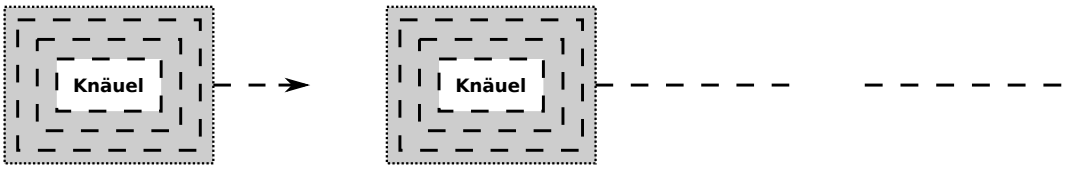
Eine Schlange der Länge $l = N$ besteht zu Beginn aus N goldfarbenen Ringeln. Jeder davon kann beim Strecken zu zwei roten Ringeln werden, die Schlange kann also aus bis zu $2N$ Ringeln bestehen. Da die roten und die goldfarbenen Ringel beliebig ausgetauscht werden können, sind Strecken und Zusammenziehen der Schlange immer an jeder Stelle möglich. Folglich kann man das Problem in die folgenden zwei Teilprobleme zerlegen:

1. Entknäule eine Schlange, die ihre Länge beliebig von $l = N$ bis $l = 2N$ Ringel variieren kann. Dabei werden die Ringelfarben zunächst außer Acht gelassen.
2. Füge das Vertauschen der Ringe im Nachhinein so zur Entknäuelanweisung hinzu, dass die Anweisung gültig wird.

Da das zweite Teilproblem relativ einfach zu lösen ist, soll zuerst die Lösung für das erste Teilproblem erklärt werden. Dabei kann man eine Fallunterscheidung danach treffen, ob ein Ende der Schlange außen liegt.

Fall 1: Ende der Schlange außen am Knäuel

Sofern ein Ende der Schlange schon zu Beginn an einer Außenseite[3] des Knäuels liegt, ist das Entknäueln recht einfach: Durch Verlängern der Schlange auf $l = 2N$ Ringel wird die Schlange am außen liegenden Ende geradlinig vom Knäuel weg gestreckt. Danach wird die Schlange wieder zusammengezogen, so dass sich aber nun das andere Ende aus dem Knäuel herauszieht. Schließlich enden wir mit einer geraden und goldfarbenen Schlange.



(a) Das eine Ende wird beim Verlängern vom Käuel weggezogen.

(b) Das andere Ende kann beim Zusammenziehen aus dem Knäuel herausgezogen werden.

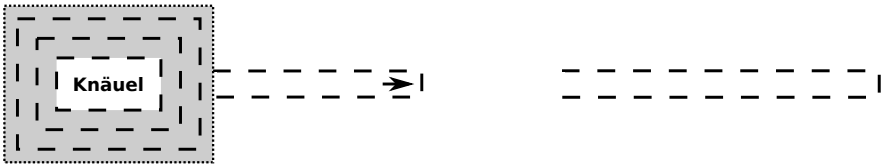
(c) Digitail ist nüchtern.

Abbildung 2: Illustration des Verfahrens im Fall 1.

Laufzeit Es werden maximal N Verlängerungen, N Verkürzungen und für das passende Vertauschen der Ringel $2 \cdot \sum_{m=0}^{N} 2m = 2N(N+1)$ Vertauschungen benötigt. Dieses Verfahren besteht also aus maximal $2N(N+2)$ Schritten.

Fall 2: Beide Enden der Schlange im Knäuel

Eine Schlange, die kein äußeres Ende hat, kann man so entkäueln: Es wird der am weitesten rechts[4] liegende Ringel gewählt. Dieser liegt senkrecht, weil er sonst ein Endstück wäre und dann der Einfachheit halber der Algorithmus von Fall 1 angewandt werden könnte. Der Ringel wird um die Länge 1 nach rechts verschoben und die Schlange damit um 2 Ringel verlängert.



(a) Ein Stück Schlange wird aus dem Knäuel gezogen.

(b) Beim Zusammenziehen werden die beiden Enden aus dem Knäuel gezogen.

Abbildung 3: Illustration des Verfahrens im Fall 2.

[3] Außenseite bedeutet hier: der Ringel liegt in eine Richtung am weitesten außen. Man kann noch verallgemeinern, dass der Ringel lediglich so liegen muss, dass es eine Richtung gibt, in die man den Ringel beliebig weit verschieben kann.

[4] Es kann natürlich auch jede andere Richtung gewählt werden.

Dies wird solange wiederholt, bis es nicht mehr möglich ist: bis die Schlange also die Länge $l = 2N - 1$ oder $l = 2N$ erreicht hat.

Schließlich wird die Schlange wieder zusammengezogen. Dazu werden je zwei Ringel an einem Ende der Schlange wieder zu einem Ringel verkürzt. Dies geschieht so lange und für beide Enden, bis das Knäuel komplett verschwunden ist.

Bildlich gesprochen wird an einer Stelle der Schlange gezogen, während sie gleichzeitig von ihrer Originallänge auf das Doppelte verlängert wird. Dann hält man sie an dieser Stelle fest und lässt sie wieder auf Originallänge zusammen schrumpfen.

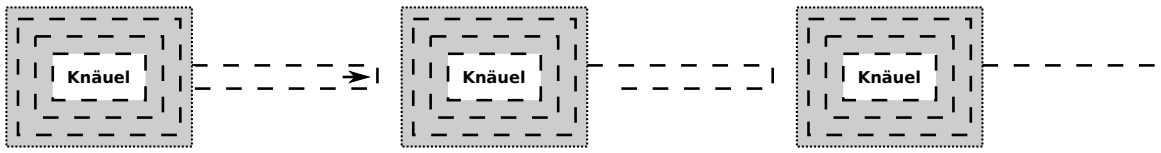
Laufzeit Hierbei werden maximal $N/2$ Verlängerungen und N Verkürzungen benötigt, und auch hier liegt die Anzahl der Vertauschungen maximal bei $2N(N+1)$. Allerdings erhält man hierbei noch keine vollkommen gerade Schlange.

Allgemeines Verfahren

Um die Aufgabe allgemein zu lösen, sollen nun die Algorithmen aus Fall 1 und Fall 2 einfach kombiniert werden. Dazu wird der Ringel bestimmt, der

- einerseits außen liegt und
- andererseits möglichst nah an einem der beiden Endstücke.

Der Abstand zu einem der Endstücke soll nun mit d bezeichnet werden. Man braucht nun den Verlängerungsschritt aus Fall 2 nur d-mal anzuwenden. Danach kann man die Schlange wieder verkürzen und erhält ein außen liegendes Ende der Schlange. Mit Fall 1 wird das Problem nun vollständig gelöst.



(a) Es wird dort gezogen, wo ein Endstück in der Nähe ist.

(b) Beim symmetrischen Zusammenziehen wird diese Konfiguration erhalten.

(c) Verkürzt man dagegen nur am kürzeren Ende, erhält man ein gerades Endstück.

Abbildung 4: Illustration des Verfahrens für Erweiterung von Fall 2.

Laufzeit Fall 1 wird d Schritte ausgeführt, das benötigt d Verlängerungen, d Verkürzungen und $d(d+2)$ Vertauschungen. Fall 2 braucht nun nur noch $N-d$ Schritte durchgeführt werden, also $N-d$ Verlängerungen, $N-d$ Verkürzungen und $2(N-d)(N-d-1)$ Vertauschungen. Insgesamt benötigen wir

$$d + d + d(d+2) + N - d + N - d + 2(N-d)(N-d-1) = 2(N-d)^2 + d(d+4)$$

Schritte. Alle Werte sind Abschätzungen nach oben.

Vertauschen der Ringel

Schließlich bleibt noch übrig, die Ringel in den Entknäuelanweisungen richtig zu vertauschen. Dazu muss zunächst die Farbe der Ringel geeignet dargestellt werden. Dann kann die Liste der Anweisungen durchgegangen werden. Vor jedem Strecken ist dafür zu sorgen, dass bei der Stelle der Verlängerung ein bzw. zwei goldfarbene Ringel liegen, vor jedem Zusammenziehen dafür, dass bei der Stelle der Verkürzung zwei bzw. vier rote Ringel liegen. Dabei kann man ohne große Effizienzverluste einen *Greedy*-Algorithmus einsetzen. Bei geeigneter Implementierung erreicht man so die oben genannte Anzahl von Schritten für das Entknäueln der gesamten Schlange.

1.4 Bewertungskriterien

- Das Entknäuelungsverfahren soll nachvollziehbar beschrieben sein. Insbesondere soll die Repräsentation der Schlange zumindest angesprochen sein. Idealerweise sollte die Lösungsidee auch auf Vor- und Nachteile der Repräsentation eingehen, dies ist aber nicht zwingend gefordert.
- Die Anzahl der verwendeten Schritte sollte nicht wesentlich größer sein als in der Musterlösung. Insbesondere sollten keine offensichtlich unnötigen Schritte auftreten.
- Die Schlange muss am Ende gerade und wieder vollständig goldfarben sein.
- Das Entknäueln muss in geeigneter Weisee visualisiert werden.
- Mindestens zwei Entknäuelvorgänge sollten in der Dokumentation protokolliert werden, darunter ein nichttrivialer Fall. Die Beispiele sollen das gewählte Verfahren anschaulich verdeutlichen.

Aufgabe 2: Geldtransporter

In der Aufgabe „Geldtransporter“ wird verlangt, dass Koffer, die jeweils einen positiven Wert und ein positives Gewicht haben, so auf einen linken und einen rechten Kofferraum verteilt werden, dass der Gewichtsunterschied so gering wie möglich ist. Dabei muss jedoch zugleich die Nebenbedingung eingehalten werden, dass der Wert aller Koffer im linken Kofferraum sich um höchstens 10 000 vom Wert der Koffer im rechten Kofferraum unterscheidet. Aufteilungen, die diese Nebenbedingung erfüllen, nennen wir eine Lösung. Wir suchen also eine optimale Lösung in der Hinsicht, dass der Betrag der Gewichtsdifferenz minimal ist.

Sei nun s der Wert aller Koffer zusammen. Die Nebenbedingung entspricht der Forderung, dass sich die Summe der Kofferwerte eines Kofferraums jeweils im Intervall $\left[\frac{s-10000}{2}, \frac{s+10000}{2}\right]$ befinden muss.

2.1 Lösungsidee(n)

Brute-Force

Abgesehen von `eingabe40.txt` sind die Beispieleingaben klein genug, dass mittels eines Brute-Force-Ansatzes eine optimale Lösung gefunden werden kann. Das bedeutet, dass alle möglichen Verteilungen der Koffer untersucht werden und für jede einzelne zuerst die Nebenbedingung geprüft und dann die Gewichtsdifferenz bestimmt werden muss. So wird sicher die Verteilung mit der geringsten Gewichtsdifferenz gefunden.

Eine Möglichkeit alle Verteilungen durchzugehen ist, binär von 0 bis $2^{\text{Kofferanzahl}-1}$ zu zählen. Dabei steht eine 0 in der Binärzahl dafür, dass der Koffer auf die linke Seite gelegt wird, und eine 1 bedeutet, dass der Koffer auf die rechte Seite kommt. Dieses Verfahren zeigt, dass ein Brute-Force-Ansatz für n Koffer 2^{n-1} Verteilungen untersuchen muss. Bei 40 Koffern dauert das zu lange.

Branch-and-Bound und Pruning

Es gibt drei wichtige Ideen, die das Finden einer Lösung wesentlich bescheunigen:

- Wenn man sich die Eingabegrößen ansieht, fällt auf, dass die Versicherung einen Großteil der Kofferverteilungen nicht zulässt. Also genügt es, nicht alle Verteilungen auszuprobieren, sondern nur jene, die versicherungstechnisch erlaubt sind (also die zu Anfang genannte Nebenbedingung erfüllen).
- Da wir außerdem ein Intervall kennen, in dem der Wert aller Koffer auf den beiden Seiten jeweils liegen muss, können wir das nutzen, um viele Verteilungen zu überspringen.
- Wurde bereits eine Lösung gefunden, dann sind alle Lösungen mit einer höheren Gewichtsdifferenz offensichtlich nicht optimal. Diese können also übersprungen werden.

Um eine Lösung zu finden, können wir einen Koffer nach dem anderen verteilen. Dabei bietet sich ein rekursiver Algorithmus an, der in jedem Rekursionsschritt einen Koffer einer Kofferraumseite zuweist. Für jeden Koffer müssen wir uns entscheiden: Ist es mit der bisherigen Kofferverteilung versicherungstechnisch möglich, den Koffer auf die linke (bzw. rechte) Seite zu legen? Kann man mit dem Gewicht der restlichen Koffer, wenn sie optimal verteilt werden könnten, überhaupt noch auf eine bessere Lösung kommen als auf die beste unter den bereits gefundenen Lösungen? Wenn beides noch möglich ist, dann müssen wir den Koffer auf die linke (bzw. rechte) Seite legen und genauso mit dem nächsten Koffer weiter machen.

Für die ersten Koffer werden immer beide Möglichkeiten in Frage kommen. Man betrachtet bei den ersten Koffern also sowohl die Möglichkeit, dass sie in den linken Kofferraum kommen, als auch die Möglichkeit, dass sie in den rechten Kofferraum kommen. Dann geht man für beide Möglichkeiten in den Rekursionsschritt und hat darin jeweils ein einfacheres Problem, da ein Koffer weniger verteilt werden muss. Diese Verzweigung in zwei (kleinere) Teilprobleme wird auch „Branch“ (engl. für verzweigen) genannt. Da rekursiv immer wieder in immer kleinere Teilprobleme verzweigt wird, entsteht ein ganzer „Rekursionsbaum“ an Bearbeitungszweigen. Sobald man die letzten Koffer verteilen muss, wird bei den gegebenen Problemstellungen aufgrund der Versicherung nur eine Verteilung möglich sein. Die Versicherung stellt also eine Schranke (engl. „Bound“) für viele Rekursionsschritte dar. Dank dieser Schranke können einige Äste des Rekursionsbaumes (also viele der kleineren Teilprobleme) ignoriert werden.

Sobald gültige Lösungen gefunden wurden, kann in vielen Situationen, in denen erst wenige Koffer verteilt wurden, bereits früh mit dem Verteilen der Koffer aufgehört werden, da es auf keinen Fall mehr möglich ist eine bessere Gewichtsverteilung zu finden. Die Koffer, die in diesen Situationen bereits im Wagen sind, sind so ungleichmäßig verteilt, dass das restliche Gewicht der noch nicht verteilten Koffer nicht ausreicht, um eine bessere Lösung zu finden. Nun ist klar, dass man diese Verteilungen nicht weiter betrachten will. Es werden also Äste im Rekursionsbaum gestutzt. Dieses Stutzen von Ästen des Rekursionsbaumes wird „Pruning“ genannt.

Partitionsproblem

Auch der obige Ansatz könnte im schlimmsten Fall zu langen Laufzeiten führen: Wenn alle Koffer zusammen nicht mehr als 10000 Euro wert sind, sind alle Verteilungen erlaubt und müssen untersucht werden. Gibt es für dieses Problem möglicherweise keine grundsätzlich bessere Lösung?

Das gestellte Problem ist dem Partitionsproblem sehr ähnlich. Das Partitionsproblem lautet: Gegeben ist eine Menge von Zahlen M. Nun soll M so in zwei Mengen geteilt werden, dass der Unterschied der Summe der beiden Mengen minimal ist.

Dieses Problem ist NP-vollständig. Das bedeutet unter anderem, dass bisher keine Möglichkeit gefunden wurde, dieses Problem effizient – also in polynomieller Laufzeit – zu lösen. Es ist nicht einmal bekannt, ob es überhaupt möglich ist, eine effiziente Lösung zu finden.

Es wurde bewiesen, dass das Partitionsproblem selbst dann noch NP-vollständig ist, wenn man fordert, dass die beiden Mengen der Partition gleich viele Elemente haben sollen. Für diese Aufgabe wäre diese Forderung erfüllt, wenn der Wert jedes Koffers gleich 10 000 wäre. In diesem Spezialfall ist unser Problem also NP-vollständig.

Wenn der Wert aller Koffer zusammen geringer als 10 000 wäre, dann würde die Versicherung bei der Verteilung der Koffer keine Rolle mehr spielen. Dann würde man nur noch nach der optimalen Aufteilung nach dem Gewicht suchen. Das wäre jedoch wieder ein Partitionsproblem, also auch NP-vollständig. Man sieht also, dass eine Programm, welches die Aufgabenstellung für alle Eingaben in polynomieller Zeit löst, nur existieren kann, wenn $\mathrm{P} = \mathrm{NP}$ gilt.

Die Beispieldatensätze entsprechen jedoch keinem der beiden beschriebenen Sonderfälle.

Dynamische Programmierung

Es gibt eine Klasse von NP-vollständigen Problemen, die „pseudo-polynomiell“ nach dem Schema der so genannten Dynamischen Programmierung (kurz: DP) gelöst werden können. Das Partitionsproblem gehört auch dazu. Auch beim hier gestellten Problem lässt sich DP einsetzen.[5]

Lineare Optimierung

Das gegebene Problem ist ein lineares, ganzzahliges Optimierungsproblem:

Sei n die Anzahl der Koffer und seien x_i mit $i = 1, \ldots, n$ die Koffer. $w(x_i)$ bezeichne das Gewicht des Koffers x_i, $v(x_i)$ bezeichne seinen Wert. Sei $c_i \in \{0, 1\}$. Minimiere

$$\sum_{i=0}^{n} w(x_i) - \sum_{i=0}^{n} c_i \cdot w(x_i)$$

unter der Nebenbedingung

$$\sum_{i=0}^{n} v(x_i) - \sum_{i=0}^{n} c_i \cdot v(x_i) \leq 10\,000.$$

Solche Probleme kann man mit dem GNU Linear Programming Kit GLPK[6] lösen.

2.2 Beispiele

Die folgende Tabelle gibt für jeden der vorgegebenen Beispieldatensätze die optimale Gewichtsdifferenz Δ, die Anzahl der möglichen Aufteilungen, die Anzahl der Lösungen und die Anzahl der optimalen Lösungen # an:

[5] `http://de.wikipedia.org/wiki/Partitionsproblem`
[6] `http://www.gnu.org/software/glpk/`

Datei	Δ	Verteilungen	Lösungen	#
`eingabe15.txt`	5	$2^{14} = 16\,384$	77	3
`eingabe20.txt`	11	$2^{19} = 524\,288$	1 323	2
`eingabe30.txt`	112	$2^{29} \approx 537 \cdot 10^6$	599 375	1
`eingabe40.txt`	5	$2^{39} \approx 550 \cdot 10^9$	$> 1\,000\,000$	≥ 1

2.3 Bewertungskriterien

- Das gewählte Verfahren muss nachvollziehbar erklärt sein.
- Das gewählte Verfahren darf nicht zu aufwändig sein; insbesondere muss es in der Lage sein, die Beispieleingabe mit 40 Koffern in überschaubarer Zeit zu bearbeiten. Ein reiner brute-force Ansatz ist nicht akzeptabel, jede effektive Verbesserung ist akzeptabel.
- Das implementierte Verfahren muss optimal sein, also insbesondere die Beispieleingaben 15, 20, 30 und 40 korrekt lösen. Ein Zufallsverfahren kann hier nicht genügen.
- Die Verwendung externer Programme wie z.B. GLPK ist erlaubt.
- Für die vorgegebenen Beispiele (oder zumindest einen relevanten Teil davon, insbesondere das große Beispiel mit 40 Koffern darf nicht fehlen) müssen die Berechnungsergebnisse angegeben sein. Weitere Beispiele können gerne hinzukommen, müssen aber nicht. Damit die Ausgabe zu einem Beispiel auch aussagekräftig ist, sollte sie nicht nur die Aufteilung der Koffer, sondern auch die Gesamtgewichte und -werte für die beiden Kofferräume und die jeweiligen Differenzen enthalten.

Aufgabe 3: Turn 90

3.1 Lösungsidee

Da der Raum für den Roboter zu Beginn unbekannt ist, kann man keines der üblichen Pfadsuchverfahren anwenden, sondern muss den Raum von dem Roboter tatsächlich »erforschen« lassen.

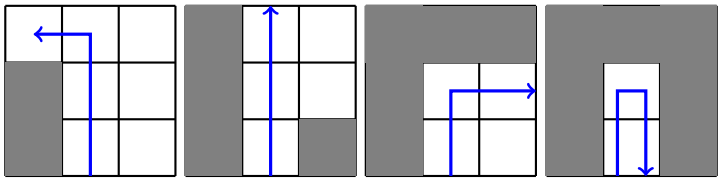
Man sieht leicht, dass der Roboter, wenn es keine Hindernisse gäbe, nur entlang der Wand des Raumes fahren müsste, um das Ziel zu erreichen, da es sich an der Außenwand befindet und der Roboter auf diese Weise alle äußeren Rasterquadrate besucht.

Gibt es nun Hindernisse, welche die Außenwand berühren, so kann man diese als Teil der Außenwand betrachten, da sie einen zusammenhängenden Pfad entlang der Wand bilden. Dabei muss nur darauf geachtet werden, tatsächlich an dieser Wand entlangzufahren, und sich nicht irgendwo zu »verirren«.

Hierzu kann man die so genannte Linke-Hand-Regel verwenden: Läuft man nämlich so durch ein Labyrinth, dass man mit einer (im Folgenden der linken) Hand stets die Wand des Labyrinths berührt, so erreicht man jeden Punkt entlang der Wand[7].

Für einen (handlosen) Roboter entstehen dabei vier Fälle:

- Falls das linke Rasterquadrat frei ist, muss der Roboter nach links drehen;
- ansonsten, falls möglich, vorwärts fahren;
- ansonsten, falls möglich, nach rechts drehen;
- ansonsten umkehren[8].

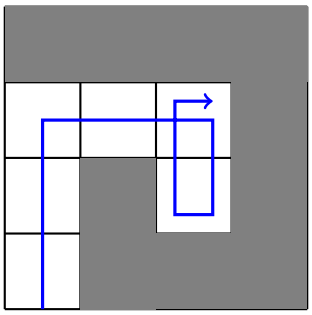


Falls der Roboter auf ein Hindernis trifft, welches nicht Bestandteil der Außenwand ist, muss er dem Hindernis ausweichen, indem er am Rande des Hindernis entlangfährt, bis er sich auf der anderen Seite befindet und sich wieder vom Hindernis lösen kann. Dazu kann die Linke-Hand-Regel nicht direkt verwendet werden, da der Roboter dann ewig am Rande des Hindernisses entlang führe und den Raum nie verließe. Stattdessen muss der Roboter nur solange an dem Hindernis entlang fahren, bis er wieder dieselbe Ausrichtung wie zuvor hat, dann kann er das Hindernis verlassen und weiterfahren. Dies wird auch als *Pledge-Algorithmus* bezeichnet.

[7] Beweis: `http://de.wikipedia.org/wiki/L\%C3\%B6sungsalgorithmen_f\%C3\%BCr_Irrg\%C3\%A4rten`

[8] Der Fall muss aber nicht berücksichtigt werden, da sich der Roboter im vorherigen Fall beim Versuch, nach rechts abzubiegen, dreht, so dass das Feld hinter dem Roboter zum neuen rechten Feld wird.

Dabei ist zu beachten, dass als Ausrichtung die absolute Rotation des Roboters gezählt wird. So ergeben zum Beispiel vier 90°-Drehungen in dieselbe Richtung eine Ausrichtung von 360° und nicht von 0°, und erst nach vier weiteren 90°-Drehungen in die Gegenrichtung würde der Roboter wieder dieselbe Ausrichtung besitzen. Ansonsten könnte der Roboter in einem Teil eines Hindernisses in einer Schleife hängenbleiben, wie das folgende Bild demonstriert:



Es ist nicht nötig, diese Sorte von Hindernissen von der Außenwand zu unterscheiden, da der Roboter sowieso nie die andere Seite der Außenwand erreichen kann, ohne dabei den Ausgang zu durchqueren.

Dieses Verfahren hat eine Speicherkomplexität von $O(1)$ und eine Fahrzeitkomplexität[9] von $O(N)$, wobei N die Zahl der freien Quadrate bezeichnet, da jedes Hindernis maximal einmal umrundet wird.

Alternativen

Eine Alternative zum Pledge-Algorithmus ist es, alle besuchten Rasterquadrate entlang des aktuellen Hindernisses zu markieren. Dann hat der Roboter ein Hindernis vollständig umrundet, wenn er ein Quadrat zum zweiten Mal erreicht. Daraufhin muss er zu dem besuchten Quadrat fahren, bei dem die größte Strecke entlang der initialen Fahrtrichtung zurückgelegt wurde, da er von dort weiterfahren kann, ohne erneut mit dem Hindernis zu kollidieren. Dies hätte eine Speicherkomplexität von $O(N)$ und eine Fahrzeitkomplexität von $O(N)$.

Eine weitere alternative, noch schlechtere Lösung ist es, die Position sämtlicher Hindernisse zu speichern, um eine Karte von dem Raum zu erstellen, die alle blockierten Rasterquadrate verzeichnet. Dann kann ein Standardalgorithmus (z.B.: Breitensuche, A*) angewandt werden und der Roboter auf dem berechneten Pfad zum Ziel fahren. Trifft er dabei auf ein bisher unbekanntes Hindernis, ist der Pfad ungültig, wird verworfen, und ein neuer Pfad auf der Karte berechnet.

Dies hätte Speicherkomplexität $O(N)$ für die Karte und Fahrzeitkomplexität $O(N^2)$, da jeder evaluierte Pfad eine Länge von $O(N)$ hat und der Roboter mehrmals dieselbe Strecke fahren könnte, aber maximal einmal für jedes freie Feld.

[9] Keine Laufzeit, weil der Roboter ja fährt.

3.2 Umsetzung

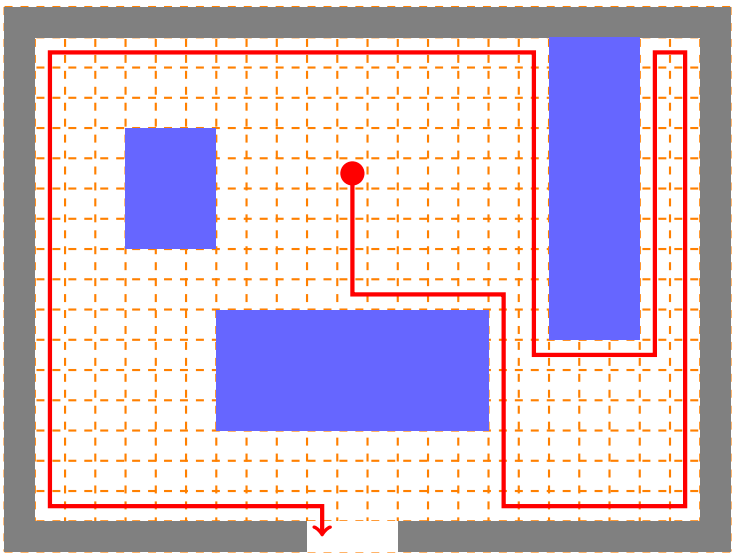
Robotersteuermodul

Der folgende Pseudocode implementiert den Pledge-Algorithmus für Roboter wie oben beschrieben, unter Verwendung der Robotersteuerfunktionen: `Aktuelle-Rotation`, `Drehe-Links`, `Drehe-Rechts`, `Drehe-Herum`, `Fahre-Vorwärts`, `Am-Ziel`:

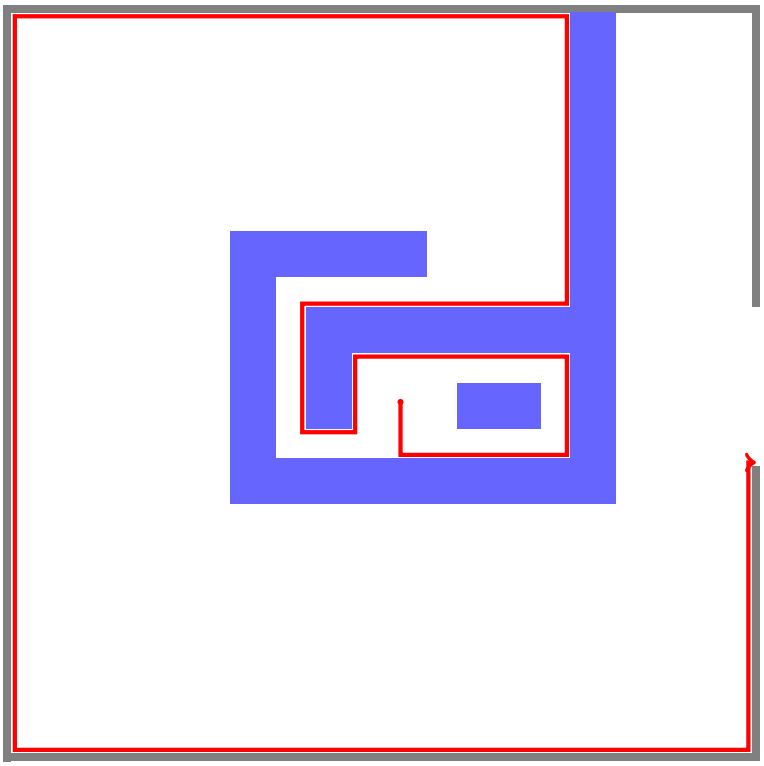
```
function GEHE-ZUM-LINKEN-RASTERQUADRAT-FALLS-FREI
    Drehe-Links
    if nicht (Fahre-Vorwärts) then
        Drehe-Rechts                          ▷ Quadrat ist nicht frei, drehe zurück
while nicht (Am-Ziel) do
                                              ▷ Vorwärts fahren, bis zu einem Hindernis:
    while nicht (Am-Ziel) ∧ Fahre-Vorwärts do
                                              ▷ Hindernis getroffen, Rotation speichern
    anfängliche-Rotation ← Aktuelle-Rotation
                                              ▷ Roboter am Hindernis ausrichten (»Hand« ausstrecken)
    Drehe-Rechts
                                              ▷ Mit dem Pledge-Algorithmus am Hindernis entlangfahren:
    while nicht (Am-Ziel) ∧ anfängliche-Rotation ≠ Aktuelle-Rotation do
        if Fahre-Vorwärts then
            Gehe-zum-linken-Rasterquadrat-falls-frei
        else
            Drehe-Rechts
```

3.3 Beispiele

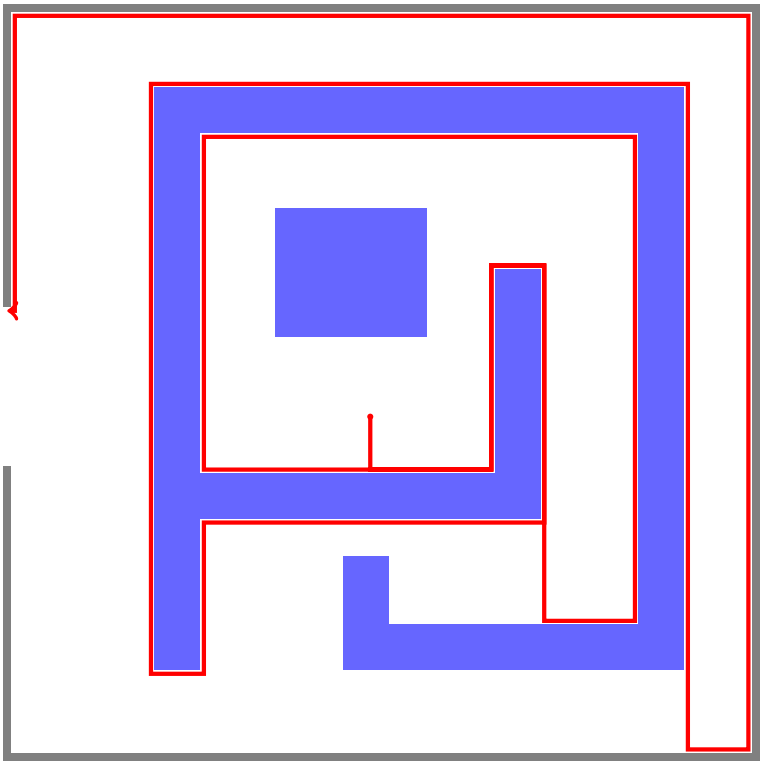
Raum 0 (Beispiel vom Aufgabenblatt)



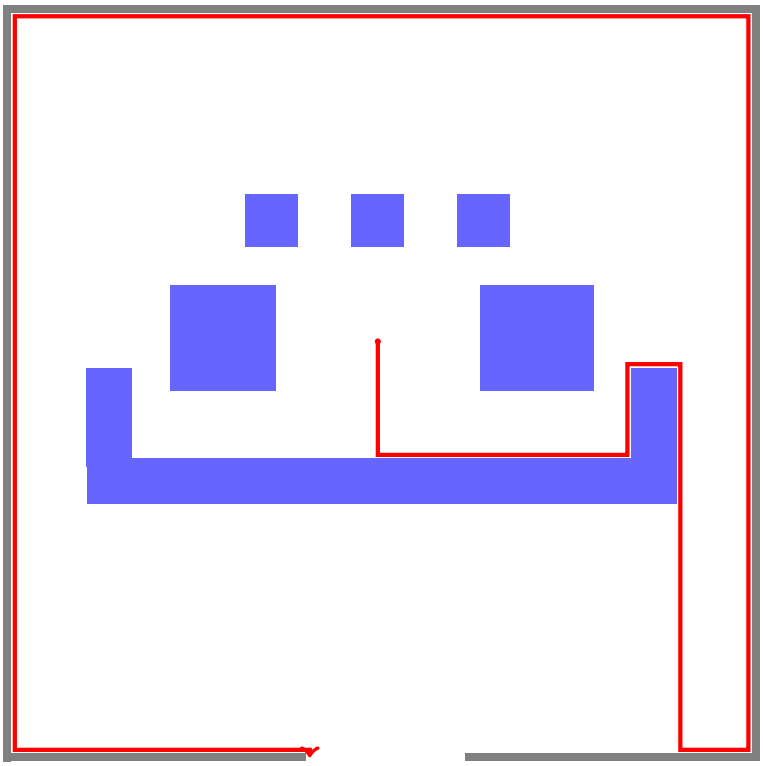
Raum 1



Raum 2



Raum 3



3.4 Bewertungskriterien

- Typischerweise wird zur Lösung dieser Aufgabe der Pledge-Algorithmus verwendet. Es genügt aber nicht, nur den Namen des Algorithmus' zu nennen; zumindest sollte kurz erklärt werden, wie er vorgeht. Eine ausführliche Begründung oder gar ein Beweis seiner Korrektheit wird nicht erwartet; bei dieser Aufgabe kann also die Beschreibung des Verfahrens relativ knapp ausfallen.
- Der Roboter muss immer den existierenden Ausgang finden. Verfahren, die dies nicht garantieren, sind nicht akzeptabel. Eine „Hand-Regel" alleine reicht nicht aus. Beim Pledge-Algorithmus muss etwa die Gesamtzahl der Drehungen berücksichtigt werden, nicht nur die resultierende Richtung.
- Der Roboter darf nur erlaubte Informationen nutzen. Insbesondere dürfen die Position der Hindernisse bzw. Wände nicht zu Beginn, sondern erst nach Betreten eines benachbarten Feldes bekannt sein.
- Die Wegsuche sollte Speicherkomplexität $O(N)$ (höchstens) und Fahrzeitkomplexität $O(N)$ haben, $O(N^2)$ ist für letzteres aber auch akzeptabel. Die Laufzeitkomplexität des Algorithmus zur Berechnung des Weges sollte nicht viel schlechter sein als die Fahrzeitkomplexität des tatsächlich gefahrenen Weges.

- Es ist nicht nötig ein Programm zu schreiben, das die Beispieldateien lädt oder die Umgebung – den Raum und die Hindernisse – simuliert. Es muss jedoch das auf dem Roboter laufende Wegsucheprogramm geschrieben werden[10].
- Die Aufgabenstellung fordert grafische Darstellungen des Roboterwegs für die vorgegebenen Räume 1 bis 3 sowie zwei weitere Räume. Abzug gibt es aber nur, wenn mindestens eines der vorgegebenen Beispiele fehlt oder wenn grundsätzlich keine grafischen Darstellungen vorliegen.

[10] Ohne Umgebungsprogramm muss das Roboterprogramm dann bei jedem Schritt den Benutzer fragen, ob das nächste Quadrat frei ist.

Aufgabe 4: SVG-Fraktale

4.1 Lösungsidee

Für vier der fünf vorgegebenen Beispielfraktale reicht es aus, immer ein einzelnes SVG-Element aus der Eingabedatei einzulesen und für dieses Element eine Menge von neuen Elementen in die Ausgabedatei zu schreiben.

Sierpinski-Teppich

Beim Sierpinski-Teppich wird ein eingelesenes schwarzes Quadrat unverändert wieder ausgegeben. Wenn ein weißes Quadrat (mit der Seitenlänge x) eingelesen wird, werden neun kleinere Quadrate (jeweils mit der Seitenlänge $x/3$) ausgegeben, von denen das mittlere Quadrat schwarz und die acht Quadrate um das schwarze Quadrat herum weiß sind.

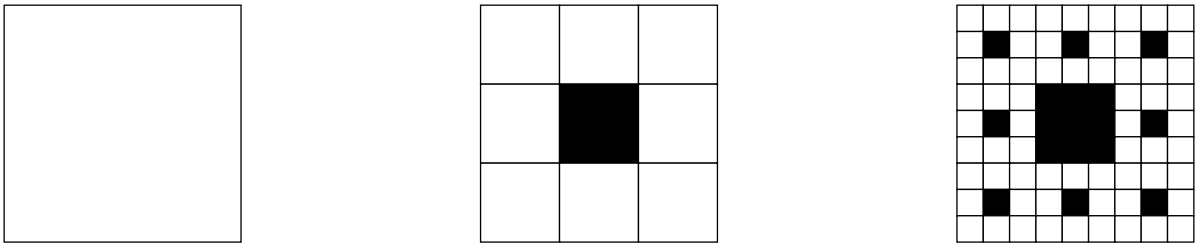


Abbildung 5: Die ersten Stufen eines Sierpinski-Teppichs. Die Ränder der Quadrate sind zur Verdeutlichung mit eingezeichnet. Schwarze Quadrate sind nicht weiter aufgeteilt.

Sierpinski-Dreieck

Beim Sierpinski-Dreieck wird ein eingelesenes weißes Dreieck unverändert wieder ausgegeben. Wenn ein schwarzes Dreieck eingelesen wird, wird für jede der drei Seiten des Dreiecks der Mittelpunkt bestimmt. Dann werden vier neue kleinere Dreiecke ausgegeben. Dabei ist das Dreieck, das von den drei Seitenmittelpunkten gebildet wird, weiß gefüllt. Die drei anderen Dreiecke sind schwarz gefüllt und werden von einem Eckpunkt des Ausgangsdreiecks und den beiden anliegenden Seitenmittelpunkten gebildet.

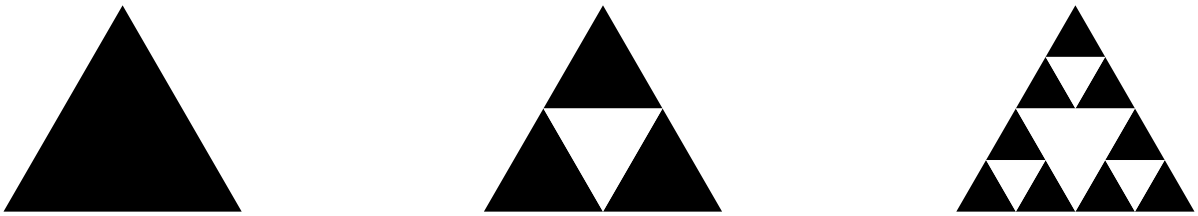


Abbildung 6: Die ersten Stufen eines Sierpinski-Dreiecks.

Koch-Kurve

Bei der Koch-Kurve werden für eine eingelesene Linie die zwei Punkte bestimmt, die die Linie dritteln. Nun werden für die eingelesene Linie vier neue Linien gezeichnet. Eine Linie geht vom Ausgangspunkt der eingelesenen Linie zum ersten Drittelpunkt und eine weitere Linie geht vom zweiten Drittelpunkt zum Endpunkt der eingelesenen Linie. Zwei weitere Linien bilden mit der (nicht gezeichneten) Linie zwischen den beiden Drittelpunkten ein gleichseitiges Dreieck. Der Punkt, an dem sich die beiden mittleren Linien treffen, ist der um 60° um den ersten Drittelpunkt gedrehte zweite Drittelpunkt.

Abbildung 7: Die ersten Stufen einer Koch-Kurve.

Koch-Flocke

Bei der Koch-Flocke kann genauso gearbeitet werden wie bei der Koch-Kurve, da hier wieder mit einzelnen Linien gearbeitet wird. Es muss nur darauf geachtet werden, dass das entstehende gleichseitige Dreieck auf der Außenseite der bisherigen Figur entsteht.

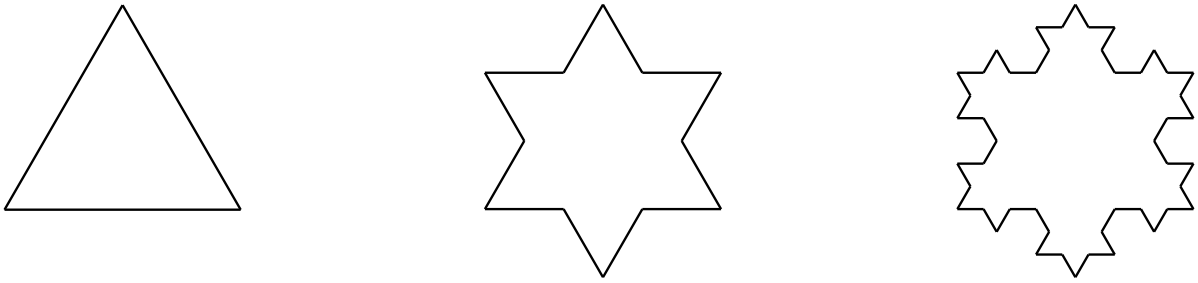


Abbildung 8: Die ersten Stufen einer Koch-Flocke.

Hilbertkurve

Bei der Hilbertkurve sieht die Regel, wie aus einer gegebenen Stufe die nächste Stufe gebildet wird, etwas anders aus. Es wird davon ausgegangen, dass Start- und Endpunkt der Hilbertkurve wie in den BwInf-Vorgaben an der oberen Seite des durch die Hilbertkurve gefüllten Quadrates liegen. Nun wird die Ausgangskurve vier Mal für die nächste Stufe kopiert. Die erste Kopie wird um 90° nach links gedreht nach oben links gesetzt. Die zweite und die dritte Kopie werden nicht gedreht und nach unten links und unten rechts gesetzt. Die vierte Kopie wird um 90° nach rechts gedreht und nach oben rechts gesetzt. Die vier Kopien werden durch drei neue Linien verbunden.

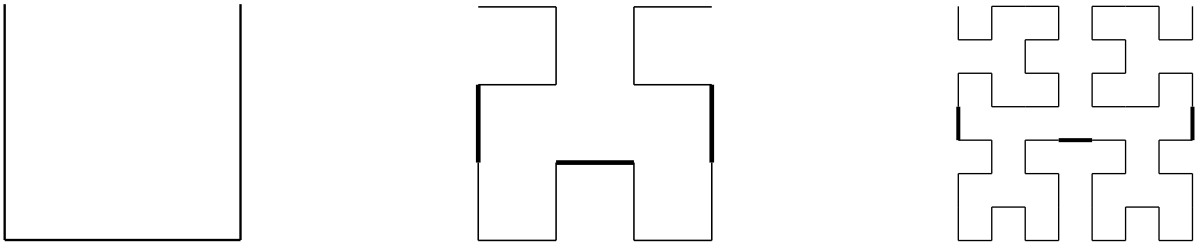


Abbildung 9: Die ersten Stufen der Hilbertkurve. In einer neuen Stufe gibt es zusätzlich zu den vier Kopien der vorigen Stufe jeweils drei (hier fetter gezeichnete) Linien, die die Kopien verbinden.

4.2 Umsetzung

Die Lösungsidee für die vier Fraktale kann einfach in ein Programm umgesetzt werden. Aus der Eingabedatei werden die einzelnen Zeichenelemente gelesen und für jedes Zeichenelement wird bestimmt, durch welche Zeichenelemente es in der Ausgabedatei ersetzt werden muss.

Das Einlesen der Eingabedatei und das Schreiben der Ausgabedatei kann mit Hilfe einer XML-Bibliothek geschehen (in Java z. B. mit JDOM[11]). Alternativ ist auch ein direktes Lesen und Schreiben der Dateien möglich; es muss aber gewährleistet sein, dass korrekte Eingabedateien gelesen werden können und dass korrekte Ausgabedateien geschrieben werden.

Die Grundstruktur eines Programms, das die Eingabedatei mit Hilfe eines XML-Parsers einliest, wird im Folgenden halbformal beschrieben:

```
Lies die Eingabedatei in den XML-Parser ein.
Gibt es auf oberster Ebene das SVG-Element?
  Wenn nein: Programm mit Fehler beenden.
Gebe XML-Header aus und startendes SVG-Element aus.
Für alle Elemente innerhalb des SVG-Elementes:
  Ist das Element bei diesem Fraktal korrekt?
    Wenn nein: dieses Element nicht bearbeiten und weitermachen.
  Bestimme die Werte des eingelesenen Elementes.
  Bestimme die hierzu auszugebenden Elemente.
  Gebe die auszugebenden Elemente aus.
Gebe beendendes SVG-Element aus.
```

Für das Fraktal Sierpinski-Teppich werden wie folgt die Werte eingelesen und die auszugebenden Elemente bestimmt:

Jedes SVG-Element im Sierpinski-Teppich muss ein `rect`-Element mit den Attributen `x` und `y` für den Ausgangspunkt, `width` und `height` für die Höhe und Breite, sowie `fill` für die Farbe sein. Diese Werte werden (hier mit Java und JDOM) direkt eingelesen:

```
float x = child.getAttribute("x").getFloatValue();
float y = child.getAttribute("y").getFloatValue();
float width=child.getAttribute("width").getFloatValue();
```

[11] http://www.jdom.org/

```
float height=child.getAttribute("height").getFloatValue();
String fill = child.getAttribute("fill").getValue();
```

Für die Ausgabe wird eine Methode `writeRect` genutzt, die genau ein `rect`-Element in die Datei `pw` ausgibt:

```
static void writeRect(PrintWriter pw,
                      float x, float y,
                      float width, float height,
                      String fill) {
  pw.println("<rect␣x=\""+x+"\"␣" +
                  "y=\""+y+"\"␣" +
                  "width=\""+width+"\"␣" +
                  "height=\""+height+"\"␣" +
                  "fill=\""+fill+"\"␣/>");
}
```

Wenn die Füllfarbe nicht weiß ist, wird das eingelesene Rechteck identisch in die Ausgabedatei `pw` geschrieben:

```
if (!"white".equals(fill)) {
  writeRect(pw, x,y,width,height, fill);
}
```

Wenn die Füllfarbe weiß ist, müssen neun neue Rechtecke geschrieben werden, die als neue Höhe und Breite nur noch ein Drittel der Höhe und Breite des eingelesenen Rechtecks haben:

```
float wNew = width/3;
float hNew = height/3;
writeRect(pw, x,        y,        wNew, hNew, "white");
writeRect(pw, x+wNew,   y,        wNew, hNew, "white");
writeRect(pw, x+2*wNew, y,        wNew, hNew, "white");
writeRect(pw, x,        y+hNew,   wNew, hNew, "white");
writeRect(pw, x+wNew,   y+hNew,   wNew, hNew, "black");
writeRect(pw, x+2*wNew, y+hNew,   wNew, hNew, "white");
writeRect(pw, x,        y+2*hNew, wNew, hNew, "white");
writeRect(pw, x+wNew,   y+2*hNew, wNew, hNew, "white");
writeRect(pw, x+2*wNew, y+2*hNew, wNew, hNew, "white");
```

Da die berechneten Koordinaten in dem Beispiel bei der Division durch 3 nicht mehr mit ganzen Zahlen darstellbar sind, wird hier mit Fließkommazahlen gearbeitet. Diese können auch in den SVG-Elementen genutzt werden. Da besonders kleine Elemente nur noch in hohen Vergrößerungsstufen sichtbar sind, ist es nicht schlimm, wenn die Berechnungen nur für Eingabeelemente einer Mindestgröße durchgeführt werden.

Auf die gleiche Art können die nächsten Stufen des Sierpinski-Dreiecks, der Koch-Kurve und der Koch-Flocke bestimmt werden.

Andere Umsetzungen

Es gibt weitere Möglichkeiten, eine neue Stufe der Fraktale zu generieren. Z. B. kann, wie in der Aufgabenstellung angedeutet, die Ersetzung mit der Hilfe von XSLT passieren.

Der Lösungsansatz für die Hilbertkurve ist auch auf das Sierpinski-Dreieck und die Koch-Kurve anwendbar. Dabei wird die gesamte eingelesene Figur genommen und mehrfach in verkleinerter Kopie in die Ausgabe geschrieben (dreimal bei dem Sierpinski-Dreieck, viermal bei der Koch-Kurve). Dabei müssen die Kopien vor der Ausgabe, den Bildungsregeln des jeweiligen Fraktals entsprechend, verschoben und bei der Koch-Kurve teilweise auch gedreht werden.

4.3 Beispiele

Die in den ersten zwei Verfeinerungsschritten entstandenen Bilder für die vorgegebenen Ausgangsdateien sind im Abschnitt „Lösungsidee“ gezeigt.

Da die Ausgabedatei der zweiten Stufe für den Sierpinski-Teppich schon in der Aufgabenstellung abgedruckt ist, werden hier die zweiten Stufen anderer Fraktale gezeigt:

Die Ausgabedatei für die zweite Stufe des Sierpinski-Dreiecks sieht wie folgt aus:

```
<?xml version="1.0" ?>
<svg xmlns="http://www.w3.org/2000/svg">
<polygon points="100.0,200.0␣150.0,200.0␣125.0,156.69873" fill="black" />
<polygon points="200.0,200.0␣150.0,200.0␣175.0,156.69873" fill="black" />
<polygon points="150.0,113.39746␣125.0,156.69873␣175.0,156.69873" fill="black" />
<polygon points="150.0,200.0␣125.0,156.69873␣175.0,156.69873" fill="white" />
</svg>
```

Die Ausgabedatei für die zweite Stufe der Koch-Kurve sieht wie folgt aus:

```
<?xml version="1.0" ?>
<svg xmlns="http://www.w3.org/2000/svg">
<line x1="100.0" y1="100.0" x2="133.33333" y2="100.0" stroke="black" stroke-width="1" />
<line x1="133.33333" y1="100.0" x2="150.0" y2="71.13249" stroke="black" stroke-width="1" />
<line x1="150.0" y1="71.13249" x2="166.66666" y2="100.0" stroke="black" stroke-width="1" />
<line x1="166.66666" y1="100.0" x2="200.0" y2="100.0" stroke="black" stroke-width="1" />
</svg>
```

Die Ausgabedatei für die zweite Stufe der Koch-Flocke sieht wie folgt aus:

```
<?xml version="1.0" ?>
<svg xmlns="http://www.w3.org/2000/svg">
<line x1="100.0" y1="200.0" x2="133.33333" y2="200.0" stroke="black" stroke-width="1" />
<line x1="133.33333" y1="200.0" x2="150.0" y2="228.86751" stroke="black" stroke-width="1" />
<line x1="150.0" y1="228.86751" x2="166.66666" y2="200.0" stroke="black" stroke-width="1" />
<line x1="166.66666" y1="200.0" x2="200.0" y2="200.0" stroke="black" stroke-width="1" />
<line x1="200.0" y1="200.0" x2="183.33333" y2="171.13249" stroke="black" stroke-width="1" />
<line x1="183.33333" y1="171.13249" x2="200.00002" y2="142.26498" stroke="black" stroke-width
    ="1" />
<line x1="200.00002" y1="142.26498" x2="166.66667" y2="142.26497" stroke="black" stroke-width
    ="1" />
<line x1="166.66667" y1="142.26497" x2="150.0" y2="113.39746" stroke="black" stroke-width="1"
     />
<line x1="150.0" y1="113.39746" x2="133.33333" y2="142.26497" stroke="black" stroke-width="1"
     />
<line x1="133.33333" y1="142.26497" x2="99.99999" y2="142.26498" stroke="black" stroke-width=
    "1" />
```

```
<line x1="99.99999" y1="142.26498" x2="116.66667" y2="171.13249" stroke="black" stroke-width=
    "1" />
<line x1="116.66667" y1="171.13249" x2="100.0" y2="200.0" stroke="black" stroke-width="1" />
</svg>
```

4.4 Bewertungskriterien

Für eine der gegebenen Grundfiguren soll ein Programm geschrieben werden, welches für das passende Fraktal Folgestufen generiert. Am einfachsten dürften der Sierpinski-Teppich (Figur 0) und das Sierpinski-Dreieck (Figur 4) sein. Danach kommen die Koch-Kurve (Figur 1) und die Koch-Flocke (Figur 3). Am schwierigsten ist die Hilbert-Kurve (Figur 2). Es genügt, ein Fraktal korrekt bearbeitet zu haben. Dass eine Figur nicht für das passende Fraktal verwendet wird, sondern etwas anderes daraus gemacht wird, führt alleine nicht zu Punktabzug.

- XML-Bibliotheken dürfen genutzt werden (zum Lesen und Schreiben der SVG-Dateien). Lesen, Umrechnen und Schreiben dürfen auch in einem mit z.B. XSLT gemacht werden.
- Die Berechnung der nächsten Stufe ist nachvollziehbar beschrieben. Dazu gehört auch, dass konkret auf die Generierung der neuen SVG-Elemente eingegangen wird.
- Die Beispiele und die selber geschriebenen SVG-Dateien müssen korrekt eingelesen werden können. Wenn in der Eingabedatei die einzelnen SVG-Zeichenelemente genau in einer Zeile stehen müssen, um sie korrekt einlesen zu können, muss dies erkannt und in der Dokumentation erwähnt sein.
- Die nächste Stufe wird als korrekte SVG-Datei geschrieben.
- Die Berechnung der nächsten Stufe wird korrekt durchgeführt. Eine Rechnung nur mit ganzen Zahlen kann nicht akzeptiert werden, da dies in den vorgegebenen Beispielen spätestens bei der dritten Stufe nicht mehr ausreicht. Die Berechnung der Ausgabeelemente darf bei erreichten kleinen Abständen abgebrochen werden.
- Es müssen mindestens zwei verschiedene, generierte Stufen des gewählten Fraktals gezeigt werden. Es ist zwar naheliegend, auch die SVG-Ausgabe des Programms durch zumindest ein SVG-Code-Beispiel zu demonstrieren (eine komplette Ausgabedatei wäre ideal, einzelne SVG-Elemente akzeptabel), aber da das nur sehr selten gemacht wurde, wird für das Fehlen eines solchen Beispiels kein kompletter Punkt abgezogen.

Aufgabe 5: Kaffeerunde

5.1 Lösungsidee

Analyse

Der gesamte Vorgang lässt sich in einzelne Zeitpunkte unterteilen, nämlich die verschiedenen Tage mit den jeweils drei Tageszeiten. An jedem dieser Punkte befindet sich das System in einem bestimmten Zustand, der durch folgende Angaben vollständig bestimmt ist:

- aktuelle Tageszeit;
- Füllstand der Kaffeekanne;
- Gesamtanzahl der von allen Mitarbeitern getrunkenen Kaffeetassen zusammen mit der bisher verstrichenen Zeit seit dem Start der Simulation;
- für jeden Mitarbeiter Anzahl der Kaffeetassen, die an jedem der letzten 10 Tage getrunken wurden;
- für jeden Mitarbeiter Häufigkeit des Kaffeekochens an jedem der letzten 10 Tage.

Mit einem Schritt zum nächsten Zeitpunkt wechselt das System zufällig in einen neuen Zustand, wobei die Wahrscheinlichkeiten der Zielzustände natürlich von den vorherigen Zuständen abhängen. In unserem Modell, in dem wir die Statistik der getrunkenen Kaffeetassen und der Kaffeekoch-Aktivitäten mit in den Zustand eingebaut haben, hängt diese Wahrscheinlichkeitsverteilung ausschließlich vom vorherigen Zustand ab, aber natürlich sind auch andere Möglichkeiten denkbar.

Das Problem ließe sich mit den Mitteln der Wahrscheinlichkeitstheorie analysieren. Wir wollen es aber aus einem etwas pragmatischeren und heuristischeren Blickwinkel betrachten.

Dazu nehmen wir zunächst an, dass jeder Mitarbeiter an jedem Tag ungefähr die gleiche Menge an Kaffee C trinkt. Es ist also sehr unwahrscheinlich, dass eine Person täglich drei Tassen ergattert, während eine andere vollkommen leer ausgeht. Dafür sorgt die Tatsache, dass jeder immer (außer nachmittags als letzter) nicht nur für sich selbst kocht, sondern auch für bis zu 9 andere Personen. Bedingt durch die Durchmischung zu Beginn in der Kaffeeküche bekommen alle in etwa gleichmäßig viel ab (und nicht nur eine kleine Teilmenge der Mitarbeiter).

Ganz analog kann man argumentieren, dass jeder Mitarbeiter im Durchschnitt in 10 Tagen die gleiche Menge an Kaffee kocht. Die benötigte Kaffeemenge in diesem Zeitraum ergibt sich zu $15 \cdot 10 \cdot C$ (für 15 Mitarbeiter und 10 Tage). Unter Vernachlässigung der Entleerung der Kaffeekanne am Ende jeden Tages muss ein Mitarbeiter in 10 Tagen also ungefähr

$$m = \frac{15 \cdot 10 \cdot C}{15 \cdot 10} = C$$

mal Kaffee kochen. Gleichzeitig will er $10C$ Tassen Kaffee in diesen 10 Tagen konsumieren und damit ist

$$\frac{m}{n} = \frac{C}{10C} = \frac{1}{10}.$$

Da dieser Wert unabhängig von C ist, ergeben sich hier prinzipiell nur zwei wesentliche Szenarien. Die Nash GmbH startet auf jeden Fall mit einer Reihe von glücklichen Mitarbeitern, da in den letzten 10 Tagen keiner von ihnen die Kaffeemaschine bedienen musste. Daher wird zu Beginn der Konsum von 3 Tassen pro Tag zunächst fortgesetzt. Der Wert für $\frac{m}{n}$ wird sich im Laufe der Zeit immer mehr $\frac{1}{10}$ annähern. Ist also $\alpha > \frac{1}{10}$, werden die Mitarbeiter weiter munter Kaffee kochen und dementsprechend auch die maximale Anzahl von 3 Tassen pro Tag zu sich nehmen.

Ist dagegen $\alpha < \frac{1}{10}$, werden die Mitarbeiter in der Regel nicht mehr glücklich sein, egal wie wenig Tassen sie pro Tag auch trinken mögen. Entsprechend wird es zunächst nur noch sporadische Kaffeekoch-Aktivitäten in der Küche geben und Mitarbeiter werden nur noch vereinzelt eine Tasse trinken. Irgendwann wird $n < 10$ für jeden Mitarbeiter sein und der Kaffeekonsum wird vollkommen zusammenbrechen, da es dann gar keine Möglichkeit mehr gibt, dass irgendjemand glücklich ist.

Das heißt also, wir erwarten für den durchschnittlichen Kaffeekonsum eine Art Sprungfunktion ähnlich zu

$$C(\alpha) = \begin{cases} 3 & : \alpha > \frac{1}{10} \\ 0 & : \alpha < \frac{1}{10} \end{cases}.$$

Der genaue Verlauf im Bereich der Sprungstelle bzw. deren Breite ist aber noch unklar und wird sich in der Simulation zeigen.

Simulation

Bei der Simulation durchläuft man für verschiedene Werte von α eine gewisse Anzahl von Tagen und Tageszeiten und geht nach den genannten Regeln in einen neuen Zustand über.

Für die Speicherung des Trink- und Kaffekochverhaltens der letzten 10 Tage bietet sich jeweils eine Warteschlange oder verkettete Liste an, wo man nach dem FIFO (*first in, first out*) - Prinzip in konstanter Zeit Elemente einfügen und in gleicher Reihenfolge wieder löschen kann. In dieser ist für jede Tageszeit enthalten, ob der entsprechende Mitarbeiter eine Tasse getrunken bzw. eine Kanne gekocht hat. Zusätzlich kann man noch die Gesamtanzahl der letzten 10 Tage vermerken, damit man diese nicht bei jeder „Glücklichkeits-Prüfung“ mittels Iteration neu berechnen muss. Mit dem Anbruch jeder neuen Tageszeit wird dabei jeweils das vordere Element entfernt, ein neues hinten eingefügt und dazu die Gesamtanzahl entsprechend angepasst.

Um das Erscheinen der Mitarbeiter in der Kaffeeküche zu simulieren, ist das Erzeugen einer zufälligen Permutation der Zahlen $1 \ldots 15$ notwendig, die für die Mitarbeiter stehen. Dieses Problem kann in der Zeit $O(n)$ (wobei n die Größe des Arrays, also in dem Fall 15, bezeichne)

gelöst werden mit folgendem einfachen Algorithmus, der auf YATES, FISHER und KNUTH zurückgeht:

```
Für jedes i = n-1 bis 1
  Erzeuge zufälliges j zwischen 0 und i
  Tausche Element i und j
```

5.2 Umsetzung

Eine Implementierung könnte grob wie folgt aussehen:

```
Für jedes alpha = 0 bis 1.0, Schrittweite dalpha
  Initialisierung Startzustand
  Für jedes d = 1 bis d_max
    KaffeeKanne <- 0
    Für jedes t = 1 bis 3
      Mitarbeiter := {1,...,15}
      ZufälligPermutieren(Mitarbeiter)

      Für jedes i = 1 bis 15
        m <- Mitarbeiter[i]
        Falls Kaffeekanne > 0
          Kaffeekanne <- Kaffeekanne - 1
          Kaffee[m].Push(ja)
          Kochen[m].Push(nein)
        Sonst
          Falls Kaffee[m].Letzte10Tage >= 10
            Falls Kochen[m].Letzte10Tage
                  / Kaffee[m].Letzte10Tage < alpha
              KaffeeKanne <- 9
              Kaffee[m].Push(ja)
              Kochen[m].Push(ja)
            Sonst
              Kaffee[m].Push(nein)
              Kochen[m].Push(nein)
          Sonst
            Kaffee[m].Push(nein)
            Kochen[m].Push(nein)
        Kaffee[m].Pop
        Kochen[m].Pop
```

Man durchläuft also die verschiedenen α in einer bestimmten Schrittweite, anschließend die Tage d, die Tageszeiten t und schließlich alle Mitarbeiter in einer zufälligen Permutation.

`Kaffee` und `Kochen` bezeichnen ein entsprechendes Array von Warteschlangen, eine für jeden Mitarbeiter. Die Methode `Letzte10Tage` gebe dabei jeweils die Summe der Werte über die letzten 10 Tage zurück. Dies kann, wie bereits erwähnt, separat gespeichert und dann beim `Push` und `Pop` jeweils aktualisiert werden.

5.3 Beispiele

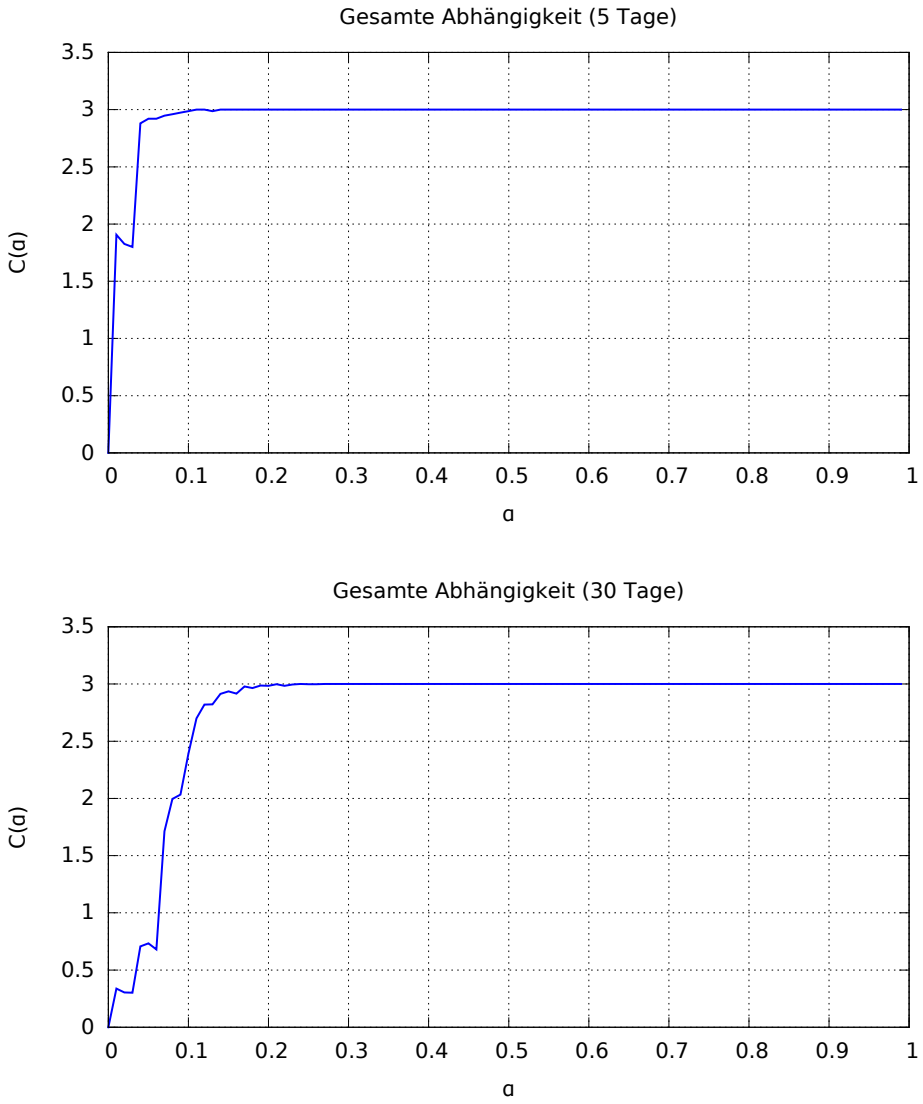


Abbildung 10: Verläufe mit wenigen Tagen

Die Simulation wurde zuerst mit 5 und 30 Tagen (im Anschluss an die Startphase) durchgeführt, mit Variation von α in $\frac{1}{100}$-Schritten. Der diskutierte erwartete Verlauf ist hierbei schon grob erkennbar, allerdings noch mit deutlichen zufälligen Störungen. Besonders, wenn man die Simulation nur 5 Tage durchführt, ist der Einfluss der Startphase mit stets gefüllter Kanne noch zu groß.

Bei einer Simulationsdauer von 2000 Tagen ist die ideale Sprungfunktion schon besser zu erkennen. Im Unterschied zu eben erkennt man auch weniger Abweichungen, wenn man die Simulation mehrmals hintereinander ausführt und die Ergebnisse plotten lässt. Das legt nahe, dass in einem geeigneten Wahrscheinlichkeitsmodell der Erwartungswert tatsächlich für jedes α konvergiert. Man erkennt auch, dass bereits ab etwa $\alpha = 0{,}08$ der Anfang eines Kaffeekon-

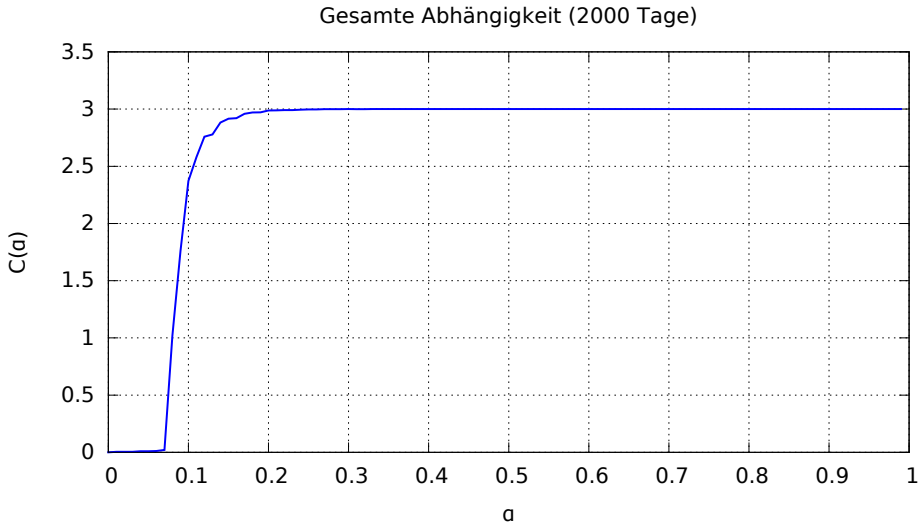


Abbildung 11: Verlauf für längere Zeit

sums zu erkennen ist, während er erst bei etwa 0,2 das tatsächliche Maximum von 3 Tassen pro Tag erreicht. Der Unterschied zum Ergebnis der Analyse (Wechsel des Konsumverhaltens bei etwa 0,1) erklärt sich aus den stark vereinfachten Annahmen, die wir dabei getroffen haben.

5.4 Bewertungskriterien

- Der Ablauf der Simulation und deren relevante Größen sollen nachvollziehbar beschrieben sein. Die richtige Stelle dafür ist die Lösungsidee, häufig wurde aber erst in der Umsetzung darauf eingegangen.
- Die Simulation in all ihren Details und ihre Implementierung sollte genau der Aufgabe entsprechen.
- Das Programm sollte auch längere Zeitspannen in akzeptabler Zeit berechnen können. Unnötige Berechnungsaufwände, etwa ein minutenweises Vorgehen bei der Simulation der Tageszeiten, sollen vermieden werden.
- Es müssen ausreichend viele und ausreichend lange Simulationsläufe durchgeführt worden sein. In den Simulationen sollen genügend verschiedene Werte für α verwendet werden, die nicht zu weit auseinander liegen. Als Simulationsdauer sind ca. 200 Tage akzeptabel (das sind grob die Werktage eines Jahres). Wenn nur kürzere Dauern verwendet werden, sollte erkannt sein, dass die Simulationsergebnisse recht ungenau ausfallen.
- Simulationsergebnisse sollen in geeigneter Weise (am besten grafisch, evtl. auch tabellarisch) dargestellt sein.

Aus den Einsendungen: Perlen der Informatik

Allgemeines

Diese Endlosschleife wird im Uhrzeigersinn durchlaufen.

Die Aufgabe des Einsenders lag nun darin, dies per Erleuchtung zu erkennen und in C++ umzuwandeln.

In dieser Aufgabe steht nicht die Entwicklung eines Logarithmus im Vordergrund.

Die Methode ... beinhaltet eine if-Verzweigung mit sieben anhänglichen else-if-Verzweigungen und einer else-Verzweigung.

Alle unsere Programme basieren auf Delphi 7. Für mehr reichen die Mittel unserer Schule leider nicht.

Nutzungsgrenzen: Das Programm setzt die Intelligenz des Nutzers voraus.

Skyline

`getDistanceToNoodle()`

Weil Bauarbeiter nicht durch ihr technisches Verständnis Geld verdienen, habe ich mich entschieden, das Programm mit einer grafischen Oberfläche als JFrame umzusetzen. *Bloß nicht Bauarbeiter unterschätzen!*

Wenn die gerundete Höhe höher ist als die ungerundete Höhe, wird von der gerundeten Höhe ein Meter abgezogen.

Aufgrund der vielen gemachten Einschränkungen in der Aufgabe umfasst das Abbild der Problematik aus der Aufgabe nur die folgenden Teilprobleme.

Verben

Alles in allem lohnt es sich nicht, das Programm zu schreiben. In der Zeit der Programmierung hätte man alle Regeln auswendig lernen können.

Frühsport

Wenn man dieses Glied gesehen hat, kann man dort die Schlange unendlich weit strecken. Die gestreckten Glieder werden bis zum Ende der Schlange bewegt. Dort entstrecken sich die Glieder wieder.

Geldtransporter

Haudrauf-Algorithmus

Die Abbruchbedingung für genetische Algorithmen ist normalerweise der Mensch.

Eine gegebene Menge von Koffern soll als Individuum angesehen werden.

Turn90

```
Richtung := ((Richtung + 1) + 4) mod 4;
```

Die Aufgabe war programmiertechnisch etwas anspruchsvoller, besonders da das Backtracking beim Roboter ein gewagter Lösungsansatz war.

Dies sieht nach einer Endlosschleife aus. Einfach gesagt: sie ist es auch.

Es gibt drei mögliche Zustände für Felder: BLOCKED(False), FREE(True), und EXIT(42).

. . . lief er gegen das Ende der Welt.

SVG-Fraktale

Stierpinski-Teppich

Ich entschied mich für den Sierpinski-Teppich, da ich mich am besten mit ihm identifizieren konnte.

Als Grundfigur habe ich mir ein Dreieck in Form eines Polygons ausgesucht.

. . . mit Hilfe einer „2D“-Schleife zu neun neuen Zeilen nach SVG-Format zusammengesetzt.

Kaffeerunde

Nasch GmbH

Die Namen der Angestellten sind alphabetisch vergeben und aus Protest gegen die Frauenquote ausschließlich männlich. *von einer Teilnehmerin!*

Die Kaffeekanne muss ihren Inhalt zurückgeben . . .

Ein Kaffeetrinker muss Schnittstellen haben, die ihm mitteilen, dass ein neuer Tag beginnt.

Wichtig ist dabei die objektorientierte Kaffeemaschine.

Jede Person ist ein Objekt, welches die getrunkenen Kaffees, die gemachten Kaffees und die Laune beinhaltet.

In diesem Fall wurden ca. 3 Jahre (genau: 160 Tage) simuliert. *Es waren dann doch 1060 Tage.*

Dabei wird immer in einem Textfeld vermerkt, was gerade passiert – sozusagen der Twitterfeed des Büros!

. . . ruft die Methode `mitarbeiterAbfuellen()` auf.

Er hat einen Schwellenwert von 0,1. Insgesamt ist er böse.

Und wenn sie an Kaffee nicht gestorben sind, dann müllen sie den Arbeitsspeicher noch heute zu.