



20. Bundeswettbewerb Informatik 2001/2002
Beispiellösungen zu den Aufgaben der 1. Runde

Wolfgang Pohl (Hrsg.)

Vorbemerkungen

Die Aufgabenstellungen des Bundeswettbewerbs Informatik werden vom Aufgabenausschuss des Wettbewerbs erarbeitet, die Aufgaben der ersten Runde in der ersten Hälfte des Ausschreibungsjahres. In dieser Zeit waren aktive Mitglieder und Gäste des Aufgabenausschusses:

Monika à Brassard
Friedrich Gasper
Torben Hagerup
Hans-Werner Hein
Joachim Hertzberg
Ina Leiß
Lothar Oppor
Wolfgang Pohl
Wolfgang Pörschke
Vera Reineke (Vorsitzende)
Uwe Schöning
Monika Seiffert
Bianca Truthe

Die Musterlösungen haben entwickelt und beschrieben
zu Aufgabe 1 (Ribo-Natter): Alexander Hullmann, Wolfgang Pohl und Markus Völker;
zu Aufgabe 2 (Wo bin ich?): Daniel Heck;
zu Aufgabe 3 (Gute Stube): Jochen Eisinger, unter Mitwirkung von Karsten Sperling;
zu Aufgabe 4 (Verstehst du Bahnhof?): Arno Bücken; und
zu Aufgabe 5 (Dosen packen): Johannes Singler.

Allgemeines

Dieses Dokument stellt Beispiellösungen zu den Aufgaben der 1. Runde des 20. Bundeswettbewerbs Informatik (BWINF) vor. Dabei sind für jede Aufgabe genau die Teile vorhanden, die in der Ausschreibung der 1. Runde (also auf dem Aufgabenblatt) gefordert werden: Lösungsidee, Programm-Dokumentation, Programm-Ablaufprotokolle bzw. Beispiele und Programm-Text. Zu diesen Teilen sind einige Anmerkungen zu machen, die bei der Erstellung einer Einsendung zu beachten sind.

- 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. Eine einfache Richtlinie: Bezeichner von Programmelementen wie Variablen, Prozeduren etc. dürfen nicht verwendet werden.
- Die Programm-Dokumentation beschreibt die wichtigsten Elemente des Programm-Textes (Datenstrukturen, Variablen, Prozeduren, Klassen etc.). Anhand dieser Beschreibung soll klar werden, wie die Lösungsidee im Code umgesetzt wird und wo die genannten Elemente im gesamten Quellcode zu finden sind.
- Auch ein Programm-Ablaufprotokoll soll keine Bedienungsanleitung sein. Es beschreibt nicht, wie das Programm (theoretisch) ablaufen sollte, auch nicht die zum Ablauf nötigen Interaktionen mit dem Programm, sondern protokolliert den tatsächlichen, inneren Ablauf eines Programms. Sprich: das Programm protokolliert seinen Ablauf selbst, z. B. durch Herausschreiben von Eingaben, Zwischenschritten oder -resultaten und Ausgaben.
- Beispiele werden als Teile des Programm-Ablaufprotokolls immer erwartet. In diesem Jahr waren sogar bei jeder Aufgabe explizit drei Beispiele gefordert und bei fast allen Aufgaben ein Pflichtbeispiel vorgegeben. Beispiele sind aus folgenden Gründen wichtig: An ihnen ist oft direkt zu sehen, ob bestimmte Punkte korrekt behandelt wurden. Wenn mehrere Beispiele gefordert sind, sollten sie wichtige Varianten des Programmablaufs zeigen, also auch Sonderfälle, die vom Programm berücksichtigt werden.
- Beispiele, aber auch Programm-Dokumentation und Programm-Text müssen ausgedruckt sein. Es nützt nichts, Beispiele auf Diskette/CD abzugeben oder ins Programm einzubauen. Bei der Begutachtung von Einsendungen ist es aus Zeitgründen oft nur möglich, das Papiermaterial anzusehen. Dieses ist alleine verbindlich.

Generell ist zu empfehlen, eine Einsendung zum Bundeswettbewerb Informatik vor dem Abschicken von jemand durchsehen zu lassen, der an der Erstellung unbeteiligt war und so den Standpunkt des Gutachters oder der Gutachterin besser nachvollziehen kann.

Angesichts des kleinen Jubiläums wurden in der 1. Runde des 20. Bundeswettbewerbs Informatik besonders reizvolle, aber auch besonders schwierige Aufgaben gestellt. Der Schwierigkeitsgrad der hier behandelten Aufgaben ist also nicht vollkommen typisch für Erstrundenaufgaben

des BWINF. Auch muss eine Einsendung qualitativ und quantitativ nicht des Niveau der hier vorgestellten Beispiellösungen erreichen, um die Begutachtung ohne Punktabzug zu überstehen.

Aufgabe 1: Ribo-Natter

Aufgabenstellung

Die Ribo-Natter (Kosename: RNA) ist eine lange Schlange mit vielen farbigen Ringeln, die in folgenden Farben vorkommen: Aquamarin (kurz: A), Umbra (U), Grasgrün (G) und Cyan (C).

Diese Schlangenart ist ganz besonders eitel und möchte dem Betrachter ein möglichst gefälliges Aussehen präsentieren. Daher legt sie sich ganz flach so auf den Boden, dass – ihrer Meinung nach – eine farblich besonders harmonische Schlängelung entsteht. Dabei gibt es vier Punkte zu beachten:

1. Sie möchte dem Betrachter gerne viele Harmoniestellen zeigen. Das sind Stellen, an denen A mit U bzw. C mit G in Berührung kommt.
2. Ein Ringel darf zu höchstens einer Harmoniestelle gehören.
3. Damit die Schlängelung nicht zu geknickt aussieht, müssen zwischen zwei Ringeln, die miteinander eine Harmoniestelle bilden, mindestens drei Ringel sein, die zu keiner Harmoniestelle gehören.
4. Besonders schön sind Harmonieabschnitte, d.h. mehrere direkt aufeinander folgende Harmoniestellen.

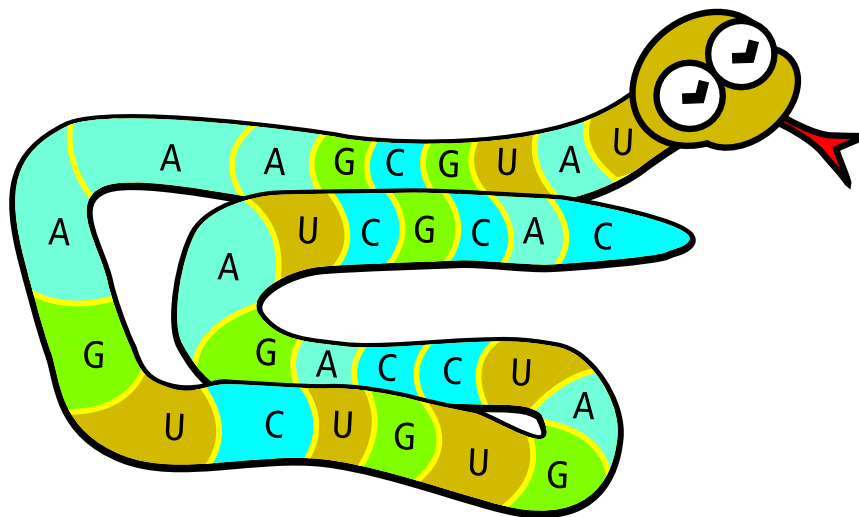


Abbildung 1: Eine Ribo-Natter mit zwei Harmonieabschnitten

Für diese Aufgabe nehmen wir an, dass die Ringel einer Schlange fortlaufend nummeriert sind, und zwar beginnend mit 1 vom Kopf her. Die oben abgebildete Schlange kann also wie folgt beschrieben werden:

Ringel: U A U G C G A A A G U C U G U G A
 Nr.: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
 U C C A G A U C G C A C
 18 19 20 21 22 23 24 25 26 27 28 29

Eine Harmoniestelle entspricht dann einem Nummernpaar, z.B. (3,28); ein Harmonieabschnitt kann durch ein Paar von Nummernbereichen beschrieben werden: (3-7,28-24). Eine komplette Schlängelung wird durch die entstandenen Harmonieabschnitte beschrieben; die Schlängelung aus der Abbildung lautet also

(3-7 , 28-24)
 (12-14 , 22-20)

Aufgabe

1. Schreibe ein Programm, das eine Schlange als Folge von Farbringeln einliest und ihren längsten Harmonieabschnitt findet. Demonstriere das Programm an drei Beispielen. Eines dieser Beispiele soll die folgende Schlange sein:
 GGGAGCGUAGCUCAGUGCGGAGAGCGCCUGCUUUGCACGC
 AGGAGGUCUGCGGUUCGAUCCCGCGCGCUCCACCA
2. Eine Schlängelung mit möglichst vielen Harmoniestellen wäre der Ribo-Natter am liebsten. Beschreibe, wie eine Schlange vorgehen könnte, um die maximale (Farb-)Harmonie zu finden.

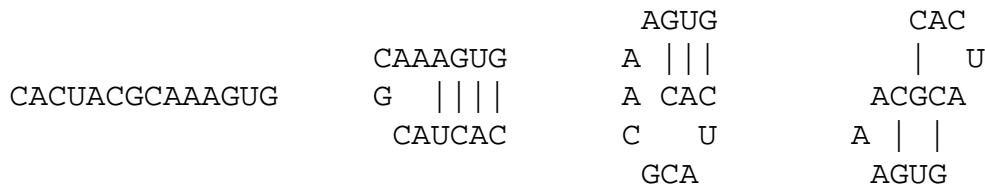
Lösungsidee

In dieser Aufgabe wird das molekularbiologische Phänomen der RNA-Faltung in vereinfachter Form behandelt. Die RNA wird zur Schlange, die Basenbindungen werden zu Harmoniestellen. Als Harmonieabschnitte werden dann Blöcke von Harmoniestellen bezeichnet.

Die Aufgabe besteht aus zwei Teilen. Im ersten Teil geht es darum, den längsten möglichen Harmonieabschnitt zu finden, den eine Schlange (RNA) durch geeignete Schlängelung (Bindungen) bilden kann. Im zweiten Teil soll eine Schlängelung gefunden werden, die zu möglichst vielen Harmoniestellen führt; inwieweit diese auch Harmonieabschnitte bilden, spielt keine Rolle.

Wie sich eine Schlange legen kann, wird in der Aufgabenstellung geregelt. Eine erste Anforderung dieser Aufgabe ist, diese Regeln richtig zu verstehen und korrekt umzusetzen. Allerdings lassen die Regeln zwei Optionen offen, die nicht im Sinne der Aufgabenstellung sind, weil sie bei einer RNA-Faltung nicht vorkommen: spiralförmige Schlängelungen und Schlängelungen, bei denen mehr als zwei Stränge parallel liegen. Beide Varianten mussten nicht berücksichtigt werden.

Hier eine Beispielschlange mit normaler, spiralförmiger und „multiparalleler“ Schlängelung.



Im Aufgabentext ist zunächst vorgegeben, dass sich die Schlange ganz flach auf den Boden legt. Das Problem ist also zweidimensional, Schlangenstücke sollen nicht übereinander liegen. Punkt 2 sagt dann, dass ein Schlangenringel (eine Base) an höchstens einer Harmoniestelle (einer Bindung) beteiligt ist. Punkt 3 hat im Vorfeld am meisten Verwirrung ausgelöst. Damit überhaupt Harmoniestellen zustandekommen, muss sich eine Schlange knicken. Dieser Knick soll nicht zu scharf sein, an der Knickstelle sollen mindestens drei Ringel frei bleiben, d.h. nicht an Harmoniestellen beteiligt sein. Dieser Punkt bezieht sich also nicht etwa darauf, dass zwischen zwei Harmoniestellen immer drei freie Ringel sein sollten – so könnten niemals Harmonieabschnitte entstehen.

Teilaufgabe 1: Finde den längsten Harmonieabschnitt

Bei der Bestimmung des längsten Harmonieabschnitts kann relativ einfach vorgegangen werden. Es genügt, von einem Knick der Schlange auszugehen. Durch mehr Knicke entstehen sicher keine längeren Harmonieabschnitte.

Naheliegender ist z. B., für jede mögliche Abschnittlänge alle Schlangenstücke dieser Länge mit allen möglichen Partnerstücken auf Harmonie zu testen. Bei diesem Test muss berücksichtigt werden, dass das Partnerstück umgedreht wird. Wenn man mit der größten Länge anfängt (halbe Schlangenlänge, ggf. abgerundet, minus 3), kann man aufhören, sobald ein Segment komplett mit einem anderen harmoniert. Alternativ kann man alle möglichen Partnerpositionen durchgehen. Dabei kann viel Zeit gespart werden, wenn für ein Positionenpaar nur noch Stücke berücksichtigt werden, die länger sind als ein bisher gefundener maximaler Harmonieabschnitt.

Eine Variante bei diesen systematischen Ansätzen ist, eine Kopie der Schlange zu erzeugen, diese umzudrehen und auch noch die Ringelfarben gemäß der Harmoniezuordnung zu invertieren. Damit kann man Original und Kopie jeweils einfach von vorne durchlaufen, der Harmonietest vereinfacht sich zum Test auf Gleichheit.

Geschickter ist, die Schlange an jeder möglichen Position zu knicken, ihre Enden gerade und ohne Auslassungen aneinander zu legen und dann in den aneinandergelegten Hälften den längsten Harmonieabschnitt zu finden. Dafür reicht es jedoch nicht aus, alle Knickstellen mit drei freien Ringeln zu betrachten: Eine Knickstelle mit drei ausgelassenen Ringeln bewirkt, dass sich nur Ringelpärchen bilden können, deren Nummern jeweils beide gerade oder ungerade sind. Um nun auch alle restlichen Ringel (von denen eine Nummer immer gerade und die andere ungerade ist) miteinander zu kombinieren, müssen an Knickstellen auch vier Ringel ausgelassen werden.

Zum Finden des längsten Harmonieabschnitts in den beiden aneinandergelegten Hälften reicht

Möglichkeiten zur Bildung von Harmoniestellen durchprobiert. Eine einfache Schilderung eines solchen „brute force“ Ansatzes genügt völlig. Es sollte allerdings erkannt werden, dass auch Schlangelungen mit mehreren Knicken (wie in Abbildung 2) zu berücksichtigen sind.

Wir wollen hier – deutlich detaillierter als in einer Einsendung nötig – auf mögliche Ansätze eingehen.

Schlangen mit einem Knick Zunächst schränken wir das Problem so ein, dass sich die Schlange nur einmal knicken darf. Das Verfahren von Teil 1 ist dennoch nicht ausreichend; es sollen ja nicht mehr die längsten Harmonieabschnitte, sondern möglichst viele Harmoniestellen gefunden werden. Eine höhere Ausbeute kann erzielt werden, wenn man zwischendurch Ringel auslässt, um an anderen Stellen einen höheren Grad von Übereinstimmung zu erzielen.

Ein möglicher rekursiver Algorithmus lautet wie folgt (Die Funktion „Harmoniestelle“ liefert den Wert 1, wenn die Ringel an den beiden Eingabepositionen eine Harmoniestelle bilden können, sonst den Wert 0.):

```
Ausbeute(Position1, Position2) =
    max( Ausbeute(Position1 + 1, Position2),
        Ausbeute(Position1, Position2 - 1),
        Ausbeute(Position1 + 1, Position2 - 1) +
            Harmoniestelle(Position1, Position2)
    );
```

Dass auf diese Weise tatsächlich das Optimum für den Spezialfall mit nur einer Krümmungsstelle gefunden wird, lässt sich darauf zurückführen, dass sich Harmoniestellen nicht überschneiden dürfen und somit nur die maximale Ausbeute für ein Paar von Positionen entscheidend ist, unabhängig davon, wie diese Lösung tatsächlich aussieht. Aufgerufen wird die Funktion „Ausbeute“ mit den Positionen 1 und N (N ist die Länge der Schlange); bei den rekursiven Aufrufen gilt $1 \leq \text{Position1} < \text{Position2} \leq N$.

Schlangen mit mehreren Knicken Möglicherweise lassen sich aber noch mehr Harmoniestellen bilden, wenn mehrere Knicke erlaubt sind (vgl. Abbildung 2). Der rekursive Ansatz lässt sich auf diesen Fall anpassen; es gibt aber auch bessere (also effizientere) Lösungsverfahren, z. B. mit Hilfe dynamischer Programmierung.

Der unten angedeutete Algorithmus arbeitet ausschließlich mit einer zweidimensionalen Wertetabelle, bei dem ein Element angibt, wie viele Harmoniestellen zwischen einer Anfangs- und Endposition erreicht werden können. Anstelle einer Rekursion wird ein zu berechnender Bereich aufgespalten und dann geprüft, ob die beiden Teilabschnitte zusammen nicht mehr Harmoniestellen bilden können. Dazu muss sichergestellt werden, dass alle kürzeren Abschnitte bereits berechnet worden sind:

```

for Laenge = 4 to LaengeSchlange
  for Startpunkt = 0 to Laenge Schlange
    feld[Startpunkt][Startpunkt + Laenge] =
      harmonie(Startpunkt, Startpunkt + Laenge) +
      feld[Startpunkt + 1][Startpunkt + Laenge - 1];
  for Abschnitt = 0 to Laenge
    feld[Startpunkt][Startpunkt + Laenge] =
      max( feld[Startpunkt][Startpunkt + Laenge],
          feld[Startpunkt][Startpunkt + Abschnitt] +
          feld[Startpunkt + Abschnitt + 1][Startpunkt + Laenge]);

```

Programm-Dokumentation (zu Teilaufgabe 1)

Eine Ribo-Natter kann sehr gut als Zeichenkette implementiert werden. Zur Bearbeitung von Zeichenketten eignet sich Perl sehr gut, so dass das Lösungsprogramm in Perl geschrieben wurde. Es implementiert den etwas vereinfachten Ansatz zur Lösung von Teil 1, wo die Originalschlange mit einer gedrehten und gemäß der Harmoniebeziehungen invertierten Kopie verglichen wird. Für Perl-Neulinge: Variablen beginnen immer mit einem $\$$ -Zeichen; „*var* $:=$ *string*“ hängt den *string* an den Wert von *var* an.

Das Perl-Programm ist sehr kurz und deshalb nicht in Unterprogramme strukturiert. Es besteht grob aus fünf Abschnitten: (1) Einlesen der Schlange aus einer Datei, (2) Erzeugen der gedrehten und invertierten Kopie, (3) Initialisierung einiger Hilfsvariablen, (4) Bestimmung der längsten Harmoniestelle und (5) Ausgabe des Ergebnisses. Der Programm-Text ist so ausführlich kommentiert, dass nur Teil 4 näher erläutert werden muss.

In der äußersten Schleife wird die Originalschlange von vorne aus durchlaufen. Die Zählvariable $\$i$ bestimmt die Anfangsposition eines zu einem möglichen Harmonieabschnitt gehörenden Teilstücks. Diese Anfangsposition kann maximal 4 Positionen vor dem Ende der Schlange liegen ($\$laenge-4$), damit die Regel 3 (Abstand zwischen zwei Ringeln einer Harmoniestelle) nicht verletzt wird. Um nun ein passendes Teilstück zu finden, müsste die Schlange von hinten her untersucht werden, wobei beim Vergleich einzelner Ringel die Harmoniebeziehung der Ringelfarben zu berücksichtigen wäre. Der gleiche Effekt wird erzielt, indem die schon gedrehte und invertierte Kopie der Schlange genau wie das Original von vorne durchsucht wird. Die entsprechende Zählvariable der inneren Schleife $\$j$ wird zusätzlich durch die äußere Zählvariable begrenzt (je weiter hinten ein Harmonieabschnitt beginnt, desto weniger Ringel bleiben als mögliche Partner übrig).

Für jedes Paar von Anfangspositionen $\$i$ und $\$j$ in Original und Kopie wird geprüft, ob sich damit ein längerer Harmonieabschnitt als bisher bekannt bilden lässt. Dabei kann ein Harmonieabschnitt höchstens halb so lang werden wie die durch $\$i$ von vorne und $\$j$ von hinten begrenzte Schlange ($2*\$halaenge < \$laenge-\$i-\$j-2$). Die beiden Teilstücke werden mittels der `substr`-Funktion gebildet; wenn sie übereinstimmen, gibt es eine neue (Zwischen-)Lösung und die entsprechenden Variablen werden aktualisiert.

Programm-Ablaufprotokolle

```
> perl ribo-natter.pl bsp1.dat
Schlange: GGGAGCGUAGCUCAGUGCGGAGAGCGCCUGCUUUGCACGCAGGAGGUCUGCGGUUCGAUC
CCGCGCGCUCCCACCA
Laengster Harmonieabschnitt: (1-7,72-66)
>
> perl ribo-natter.pl bsp2.dat
Schlange: UUUUUGGGGAACCAACAGUUGGUUCCCCCAAAAA
Laengster Harmonieabschnitt: (1-16,35-20)
>
> perl ribo-natter.pl bsp3.dat
Schlange: AAAUAAACCCGCC
Laengster Harmonieabschnitt: Keine Harmoniestellen!
>
```

Programm-Text

```
1  #!/usr/bin/perl

   # BWINF 20.1: Ribo-Natter

5  # Schlange aus übergebener Datei einlesen
   open DATEI, "<$ARGV[0]<";
   $schlange = <DATEI>;
   close DATEI;

10 # Kopie in umgekehrter Reihenfolge erzeugen
   $invschlange = reverse $schlange;
   # in der Kopie A mit U und G mit C vertauschen
   $invschlange =~ tr/ACGU/UGCA/;

15 # Länge der Schlange
   $laenge = length $schlange;
   # Länge des größten gefundenen Harmonieabschnitts
   $maxha = 0;
   # Loesungsspeicher, passend initialisiert
20 $loesung = "Keine Harmoniestellen!";

   # Originalschlange bis Ringel $laenge-4 durchlaufen
   for ($i=0; $i<$laenge-4; $i++) {
       # Schlangenkopie bis Ringel $laenge-$i-4 durchlaufen
25   for ($j=0; $j<$laenge-4-$i; $j++) {
           # nach größerer Übereinstimmung suchen
           for ($halaenge=$maxha+1;
               2*$halaenge < $laenge-$i-$j-2;
               $halaenge++) {
```

```
30     # Vergleichsstück der Ausgangsschlange
      $m1 = substr($schlange, $i, $halaenge);
      # Vergleichsstück der Schlangenkopie
      $m2 = substr($invschlange, $j, $halaenge);
      # wenn beide eine Harmoniestelle bilden...
35     if ($m1 eq $m2) {
          # ...neue Maximallänge festsetzen
          $maxha = $halaenge;
          # ...und Harmonieabschnittsbeschreibung bilden
          $loesung = '('.$(i+1).'-'.($i+$halaenge).',';
40     $loesung .= ($laenge-$j).'-'.($laenge-$j-$halaenge+1).")";
        }
        # ansonsten...
        else {
45     # ...Vergleich der Stellen $i und $j beenden
          last;
        }
      }
    } # Ende for-Schleife Schlangenkopie
  } # Ende for-Schleife Originalschlange
50
# Schlange ausgeben
print "Schlange: $schlange\n";
# Längsten Harmonieabschnitt ausgeben
print "Laengster Harmonieabschnitt: $loesung\n";
```

Aufgabe 2: Wo bin ich?

Aufgabenstellung

Labormaus Ludwig erwacht aus der Narkose. Um sich herum fühlt er Sperrholzwände, und er weiß: Wieder hat die nette Verhaltensforscherin ihn irgendwo in das Labyrinth gesetzt, das er nun schon so gut kennt. Immer an derselben Stelle steht ein Napf mit etwas Futter. Die Verhaltensforscherin wird ihm außerdem noch mehr Futter geben, wenn er am Napf ist, und zwar umso mehr, je eher er dort ist – so hat Ludwig sie inzwischen dressiert.

Ludwig ist hungrig wie immer, und er weiß genau, was er tun muss: Am schnellsten kommt er zum Futter, wenn er zunächst herausfindet, wo exakt er im Labyrinth ist, und von dort den kürzesten Weg zum Futternapf rennt. Da es immer dasselbe Labyrinth ist, hat er die Wegekarte klar im Kopf. (Ludwig entstammt einer alten Dynastie von Labormäusen, die über viele Generationen gelernt haben, sich Labyrinth einzuprägen.) Er kann ferner das Rauschen der Klimaanlage an der Nordwand des Laborraums hören, so dass er jederzeit weiß, wo Norden liegt. Da es dunkel ist im Labyrinth, kann er leider nur Wände unmittelbar um ihn herum wahrnehmen und nicht in Gänge oder Abzweige hineinschauen.

Seine Position im Labyrinth bestimmt Ludwig so: Er läuft die Labyrinthgänge entlang zu Kreuzungen, Einmündungen, Ecken und Sackgassen so lange, bis er an den Ganglängen und an der Form und Lage all der passierten Kreuzungen nach seiner Karte im Kopf eindeutig erkennen kann, wo er sich befindet. Manchmal kommt er zufällig schon während der Positionsbestimmung am Futternapf an: Das spart Lauferei, denn dann kann er direkt fressen.

Das Labyrinth ist abgeschlossen, Ludwig kann es also nicht verlassen. Alle Orte darin sind von allen Orten aus zu erreichen. Die Gänge des Labyrinths laufen in die Richtungen Nord, Ost, Süd oder West. Manche Gänge enden in Sackgassen. Die Futterstelle liegt am Ende einer Sackgasse.

Aufgabe

Die Positionen in einem Labyrinth liegen auf einem Raster mit Spalten und Zeilen, die zur einheitlichen Benennung mit Zahlen markiert sind. Die folgende Abbildung zeigt ein Labyrinth mit je 15 Spalten und Zeilen; der Futternapf steht an Position (3,10).

Schreibe ein Programm, das Ludwigs Methode zur Positionsbestimmung nachvollzieht. Gib das Verhalten der Maus Ludwig an drei Beispielen mit zufällig gewählten Startpunkten wieder. Das abgebildete Labyrinth muss mindestens in einem der Beispiele verwendet werden; dabei soll sich der Startpunkt an Position (14,3) befinden.

Lösungsidee

Für diese Aufgabe existieren im wesentlichen zwei verschiedene Lösungsansätze. Zunächst einmal ist es möglich, Ludwig durch das Labyrinth zu bewegen und dabei die besuchten Felder (d.h. deren Wandkonfiguration) auf einer Karte zu vermerken. Sobald sich diese Karte nur noch

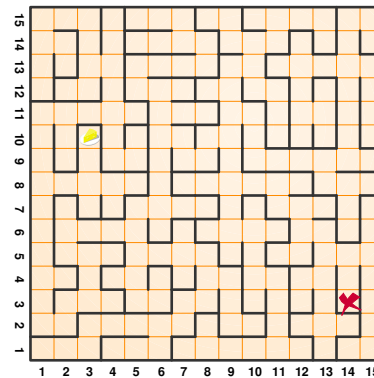


Abbildung 3: Ludwigs Labyrinth

an einer einzigen Stelle im Labyrinth anlegen lässt, kann man aus dieser Anlegestelle auf die momentane Position der Maus schließen.

Der andere (etwas elegantere) Ansatz verwaltet stattdessen eine Liste aller Positionen im Labyrinth, an denen sich Ludwig im Moment prinzipiell befinden könnte. Am Anfang sind das alle Felder, die genauso aussehen wie das, auf dem Ludwig gerade steht. Da wir zunächst nicht mehr Informationen über Ludwigs Aufenthaltsort haben, sind das gerade seine „möglichen Standorte“.

Bewegt sich Ludwig nun im Labyrinth, so muss man natürlich diese „möglichen Standorte“ mit jedem Schritt in dieselbe Richtung wie Ludwig verschieben. Dabei kann es passieren, dass eine dieser Positionen auf ein Feld zeigt, das sich von dem unterscheidet, das Ludwig tatsächlich im Labor sieht. Offensichtlich stellen alle Positionen in der Liste, auf die das zutrifft, *keine* möglichen Standorte mehr dar und können aussortiert werden. Nach endlicher Schrittzahl befindet sich irgendwann zwangsläufig nur noch eine einzige Position in der Liste – das muss dann Ludwigs tatsächlicher Aufenthaltsort sein.

Wie sich Ludwig vor jedem Schritt für seine Bewegungsrichtung entscheidet, ist in der Aufgabenstellung nicht vorgegeben. Das hier vorgestellte Programm verwendet dafür eine so genannte Tiefensuche: Ausgehend von der Startposition besucht Ludwig in einer bestimmten Reihenfolge (hier: Norden, Osten, Westen, Süden) rekursiv die benachbarten erreichbaren Felder, die er noch nicht kennt. Führt das nicht zum Ziel (also zu einer eindeutigen Position), geht er einen Schritt zurück und versucht es von dort aus erneut.

Das Labyrinth lässt sich in einem zweidimensionalen Array verwalten. Jedes Feld mit seinen vier umliegenden Wänden kann dabei als Zahl zwischen 0 und 15 dargestellt werden: In Binärnotation entspricht das den Zahlen 0000_b bis 1111_b , d.h. jeder Wand entspricht ein Bit, das entweder gesetzt ist (Wand vorhanden) oder eben nicht (Wand fehlt). (Genau genommen reichen die Zahlen von 0 bis 14 aus, da Felder mit vier Wänden laut Aufgabenstellung nicht vorkommen.)

Programm-Dokumentation

Die Implementierung in C++ hält sich eng an die eben besprochene Lösungsidee. Die im Programm auftretenden Datentypen sind `Richtung`, `Position`, `Feld`, `Labor` und `Ludwig`; ihre grobe Aufgabe sollte bereits aus den Namen ersichtlich sein.

Die wichtigen Klassen des Programms sind `Labor` und `Ludwig`. Erstere verwaltet das Labyrinth inklusive der Positionen von Futter und Maus. Innerhalb des Programmes wird dabei für die Positionen ein anderes Koordinatensystem verwendet als in der Aufgabenstellung: Die Indizierung beginnt bei 0 und der Ursprung liegt in der nordwestlichen Ecke des Labyrinths. Lediglich bei Ein- und Ausgaben werden die Positionen zwischen den beiden Darstellungen umgerechnet: dazu dienen die Funktionen `interne_koord` und `externe_koord`. Schließlich nimmt `Labor` der Maus noch etwas Arbeit ab und protokolliert für die Tiefensuche, welche Felder schon besucht wurden und welche nicht.

Das eigentliche Durchsuchen des Labyrinths erledigt `Ludwig` persönlich, und zwar in den Funktionen `besuche_momentanes_feld` und `versuche_richtung`. Ein spezieller Trick sorgt dafür, dass die Suche sofort abgebrochen wird, wenn die Position bekannt ist: Das Programm erzeugt eine Ausnahme, die im Hauptprogramm abgefangen wird. Auf diese Weise werden alle rekursiven Funktionsaufrufe auf einmal beendet, und es wird verhindert, dass `Ludwig` den ganzen Weg zum Startpunkt zurückläuft.

Als Eingabe erwartet das Programm eine Datei mit folgendem Format:

```
15 15
3 10
14 3
+--+--+--+--+--+--+
|-| |+-.|-|. +|||
|||. +--+ .+. +.-.-
|+. ||--+ .|-|+||
+--.-|-|-| |||. ||
||+|+|-+. |||||
|||. |. |. |---.-
|-.--|+-.---|--
|+| |+.-+|-|-|+
||--.|++..-|-|
||--|-.-| |+.-.
|-|+. +||-|-+. ||
|+.-|. +. |||||
|. +---. +|-|+.-|
+- .+- .+ .--.-|+.
```

Die erste Zeile enthält Breite und Höhe des Labyrinths, die beiden nächsten die Position des Futters und Ludwigs Startposition. Die folgenden Zeilen enthalten das Labyrinth. Dabei beschreibt

jedes Zeichen die nördliche und westliche Wand des jeweiligen Feldes: Ein '+' bedeutet, dass beide vorhanden sind, '-' und '|' stehen für einzelne Wände im Norden bzw. Westen und jedes andere Zeichen wird als „keine Wand im Norden oder Westen“ interpretiert. Der Vorteil dieser Notation ist, dass sie sowohl von Menschen als auch von Computern mit wenig Aufwand gelesen werden kann.

Programm-Ablaufprotokolle

Beispiel 1

Hier ist zunächst ein Probelauf mit dem vorgegebenen Labyrinth und der vorgegebenen Startposition (14,3). Wie man sich auch leicht von Hand überzeugt, weiß Ludwig hier zwangsläufig nach drei Schritten genau, wo er sich befindet. Die mit ## beginnenden Zeilen geben Ludwigs momentan mögliche Positionen an.

```
## (1,12) (2,13) (2,9) (3,7) (4,7) (6,6) (7,4) (9,5) (10,3)
   (11,10) (12,10) (12,3) (13,14) (13,10) (14,6) (14,3)
   (15,14) (15,10)
```

Ich gehe nach Norden

```
## (1,13) (2,10) (11,11) (14,7) (14,4) (15,11)
```

Ich gehe nach Norden

```
## (2,11) (14,8) (14,5)
```

Ich gehe nach Osten

Meine Position ist 15,5.

Benoetigte Schritte: 3

Beispiel 2

Hier ist ein zweites Beispiel, das sich leicht von Hand nachprüfen lässt. Das verwendete Labyrinth ist das aus der Aufgabenstellung, aber Ludwigs Startposition liegt bei (8,7).

```
## (2,4) (3,15) (4,5) (4,3) (5,11) (5,8) (7,15) (7,12)
   (8,11) (8,7) (9,15) (10,2) (11,6) (12,8) (13,15)
   (13,7) (14,2) (15,15) (15,9) (15,8)
```

Ich gehe nach Westen

```
## (2,15) (6,15) (7,11) (7,7) (12,15)
```

Ich gehe nach Westen

Meine Position ist 6,7.

Benoetigte Schritte: 2

Beispiel 3

Das letzte Beispiel verwendet eine andere Eingabedatei, die ein kleineres Labyrinth von 7×7 Feldern beschreibt. Der Käse liegt dabei in der rechten oberen Ecke. Dieses Beispiel demonstriert vor allem, wie Ludwig reagiert, wenn er unterwegs zufällig den Käse findet: Er lässt alles stehen und liegen und macht sich ans Futtern...

Dass er hier direkt auf den Käse zusteuert, liegt natürlich an der Reihenfolge, in der er bei der Tiefensuche die verschiedenen Himmelsrichtungen ausprobiert (wie schon gesagt: Norden, Osten, Westen, Süden).

```

7 7
7 7
6 6
+-----
|..+..+
|.+.|. |
|..|.+.
|.+.+..
|+.+...
|.....

```

Die Programmausgabe sieht folgendermaßen aus:

```

## (2,2) (2,1) (3,5) (3,3) (4,6) (5,3) (6,6) (6,4)
Ich gehe nach Osten
## (4,3) (7,6)
Ich gehe nach Norden
Lecker, Kaese!
Meine Position ist 7,7.
Benötigte Schritte: 2

```

Programm-Text

```

1 // BWINF 20.1: Wo bin ich?
  #include <assert.h>
  #include <fstream>
  #include <vector>
5  #include <string>
   using namespace std;

   enum Richtung { NORD, OST, SUED, WEST };

10 Richtung gegenrichtung(Richtung r)
    {

```

```

        switch (r) {
        case NORD: return SUED;
        case OST: return WEST;
15      case SUED: return NORD;
        case WEST: return OST;
        }
    }

20 //
    // Position im Labyrinth
    //
    struct Position {
        int x, y;

25      Position(int xx=0, int yy=0) : x(xx), y(yy) {}
        void bewege(Richtung r) {
            switch (r) {
            case NORD: y--; break;
30          case OST: x++; break;
            case SUED: y++; break;
            case WEST: x--; break;
            }
        }
35      bool operator==(Position p) const {
            return (x==p.x && y==p.y);
        }
    };

40 //
    // Ein Feld im Labyrinth
    //
    class Feld {
    public:
45      int waende;      // Bitmaske der vorhandenen Waende
        bool besucht;  // Hat Ludwig dieses Feld schon besucht?

        Feld() : waende(0), besucht(false) {}
        void neue_wand(Richtung r) { waende |= (1 << r); }
50      bool hat_wand(Richtung r) { return waende & (1 << r); }
        bool operator==(Feld f) const { return waende == f.waende; }
    };

    //
55 // Das Labor, in dem sich Ludwig bewegt
    //
    class Labor {
    public:
        Labor();
60      bool lade(char *filename);
        void bewege_maus(Richtung r);
        bool kaese_gefunden();

```

```

    Feld momentanes_feld();
    bool pos_gueltig(Position p);
65    bool feld_bekannt(Richtung r);

    Feld & feld(Position pos);
    int breite() { return breite_; }
    int hoehe() { return hoehe_; }
70    Position kaesepos() { return kaesepos_; }

    Position interne_koord(Position extpos);
    Position externe_koord(Position intpos);
protected:
75    void neue_wand(int x, int y, Richtung r);
private:
    // 2-dimensionales Array von Feldern
    vector<vector<Feld> > labyrinth;
    int breite_, hoehe_;           // Ausmasse des Labyrinths
80    Position mauspos_;           // aktuelle Position der Maus
    Position kaesepos_;           // Position des Futternapfes
};

Labor::Labor() { breite_ = hoehe_ = 0; }
85
bool Labor::feld_bekannt(Richtung r)
{
    Position p=mauspos_;
    p.bewege(r);
90    return feld(p).besucht;
}

bool Labor::pos_gueltig(Position p)
{
95    return (p.x>=0 && p.x<breite() &&
            p.y>=0 && p.y<hoehe());
}

Feld Labor::momentanes_feld() {
100    return feld(mauspos_);
}

bool Labor::kaese_gefunden() {
105    return (mauspos_ == kaesepos_);
}

void Labor::bewege_maus(Richtung r)
{
110    assert(momentanes_feld().hat_wand(r) == false);
    mauspos_.bewege(r);
    feld(mauspos_).besucht = true;
}

```

```
// Liefere Referenz auf ein bestimmtes Feld im Labyrinth.
115 Feld & Labor::feld(Position pos)
    {
        assert(pos_gueltig(pos));
        return labyrinth[pos.y][pos.x];
    }
120
// Setze eine neue Wand in das Labyrinth
void Labor::neue_wand(int x, int y, Richtung r)
    {
        Position p(x,y);
125     if (pos_gueltig(p))
            labyrinth[y][x].neue_wand(r);
    }

// Umwandlung zwischen den beiden Koordinatendarstellungen
130 Position Labor::interne_koord(Position pos) {
    return Position(pos.x-1, hoehe()-pos.y);
}
Position Labor::externe_koord(Position pos) {
135     return Position(pos.x+1, hoehe()-pos.y);
}

bool Labor::lade(char *dateiname)
    {
140     ifstream datei(dateiname);
    if (!datei) {
        cerr << "Kann Datei nicht oeffnen: " << dateiname << endl;
        return false;
    }

145     // Labyrinthgroesse und Start-/Zielposition einlesen
    int b=0, h=0;
    Position kpos, mpos;
    datei >> b >> h;
    datei >> kpos.x >> kpos.y >> mpos.x >> mpos.y;

150     if (datei.fail()) {
        cerr << "Fehlerhafte Datei: " << dateiname << endl;
        return false;
    }

155     breite_ = b;
    hoehe_ = h;
    kaesepos_ = interne_koord(kpos);
    mauspos_ = interne_koord(mpos);

160     // Labyrinth einrichten und Zeile fuer Zeile einlesen
    labyrinth.resize(h);
    for (int y=0; y<h; y++)
        labyrinth[y].resize(b);
```

```

165     for (int y=0; y<h; y++) {
        string zeile;
        datei >> zeile;
        for (int x=0; x<b && x<zeile.length(); x++) {
170             switch (zeile[x]) {
                case '-':
                    neue_wand(x, y, NORD);
                    neue_wand(x, y-1, SUED);
                    break;
175             case '|':
                    neue_wand(x, y, WEST);
                    neue_wand(x-1, y, OST);
                    break;
                case '+':
180                 neue_wand(x, y, NORD);
                    neue_wand(x, y-1, SUED);
                    neue_wand(x, y, WEST);
                    neue_wand(x-1, y, OST);
                    break;
185             default:
                    break;
            }
        }
    }
190
    // Labyrinth nach allen Seiten abschliessen
    for (int y=0; y<h; y++) {
        neue_wand(0, y, WEST);
        neue_wand(b-1, y, OST);
195    }
    for (int x=0; x<b; x++) {
        neue_wand(x, 0, SUED);
        neue_wand(x, h-1, NORD);
    }
200    feld(mauspos_).besucht = true;
    return true;
}

class PositionBestimmt {};
205
//
// Simulation von Ludwig
//
class Ludwig {
210 public:
    Ludwig(Labor *lab);
    void starte_suche();
protected:
    void versuche_richtung(Richtung r);
215    void besuche_momentanes_feld();

```

```

    void bewege(Richtung r);
private:
    Labor *labor;
    vector<Position> moegl_standorte;
220     int nschritte; // Anzahl der bisher gemachten Schritte
};

Ludwig::Ludwig(Labor *lab) {
225     labor = lab;
}

void Ludwig::bewege(Richtung r)
{
230     cout << "## ";
    for (int i=0; i<moegl_standorte.size(); ++i) {
        Position p = labor->externe_koord(moegl_standorte[i]);
        cout << "("<<p.x<<","<<p.y<<") ";
    }
    cout << endl;
235

    char* richtungen[] = {"Norden", "Osten", "Sueden", "Westen"};
    cout << "Ich gehe nach " << richtungen[r] << endl;

    // Maus bewegen
240     labor->bewege_maus(r);
    nschritte++;

    // Alle moeglichen Standorte verschieben und ungueltige
    // aussortieren
245     Feld f = labor->momentanes_feld();
    vector<Position> temp;
    for (int i=0; i<moegl_standorte.size(); ++i) {
        Position p = moegl_standorte[i];
        p.bewege(r);
250         if (labor->pos_gueltig(p) && labor->feld(p) == f)
            temp.push_back(p);
    }
    moegl_standorte = temp;
255 }

// Versuche, rekursiv eines der benachbarten Felder zu besuchen,
// sofern dieses bisher unbekannt und direkt erreichbar ist. Gehe
// danach den gemachten Schritt wieder zurueck.
void Ludwig::versuche_richtung(Richtung r)
260 {
    Feld f=labor->momentanes_feld();
    if (!f.hat_wand(r) && !labor->feld_bekannt(r)) {
        bewege(r);
        besuche_momentanes_feld();
265         bewege(gegenrichtung(r));
    }
}

```

```
    }

void Ludwig::besuche_momentanes_feld()
270 {
    if (labor->kaese_gefunden()) {
        cout << "Lecker, Kaese!\n";
        moegl_standorte.clear();
        moegl_standorte.push_back(labor->kaesepos());
275     }
    if (moegl_standorte.size()==1)
        throw PositionBestimmt();

    versuche_richtung(NORD);
280     versuche_richtung(OST);
        versuche_richtung(WEST);
        versuche_richtung(SUED);
}

285 void Ludwig::starte_suche()
{
    Feld f = labor->momentanes_feld();
    Position pos;
    for (pos.x=0; pos.x<labor->breite(); pos.x++)
290         for (pos.y=0; pos.y<labor->hoehe(); pos.y++)
            if (labor->feld(pos) == f)
                moegl_standorte.push_back(pos);

    // Starte rekursive Suche
295     nschritte = 0;
    try {
        besuche_momentanes_feld();
    } catch (PositionBestimmt) {
        Position ext = labor->externe_koord(moegl_standorte[0]);
300         cout << "Meine Position ist " <<ext.x <<","<<ext.y<<".\n";
        cout << "Benoeetigte Schritte: " << nschritte << endl;
    }
}

305 int main(int argc, char* argv[])
{
    char* datei = "labyrinth.dat";
    if (argc == 2)
        datei = argv[1];
310

    Labor labor;
    if (labor.lade(datei)) {
        Ludwig ludwig(&labor);
        ludwig.starte_suche();
315     }
}
```

Aufgabe 3: Gute Stube

Aufgabenstellung

Die Cyber-Künstlerin EKAI will all die öden Chatrooms möblieren. Sie hat schon hübsche Bilder für die Objekttypen „Sessel“, „Couch“, „Zimmerpflanze“, „Kunstobjekt“, „Sitzkissen“, „Fahrrad“, „Lagerfeuer“, „Teddybär“, „Stehlampe“ und „Haustier“ entworfen. EKAI will, dass Chatter die vorkommenden Gegenstände und ihre Lage zueinander gemeinsam beschreiben und dann alle das Foto ihres Chatrooms auf dem Bildschirm sehen können. Die möglichen Lagen zueinander seien „links von“, „rechts von“, „neben“, „vor“, „hinter“, „über“ und „unter“.

Aufgabe

1. Schreibe für EKAI ein Programm, das
 - (a) aus einer Folge von Sätzen die gemeinten Gegenstände und Lagebeschreibungen herausfiltert,
 - (b) alle Mehrdeutigkeiten, Unvollständigkeiten und Widersprüche in der Gesamtbeschreibung selbstständig irgendwie entscheidet und
 - (c) ein 400x300 Pixel großes Bitmap (Foto) mit einer grafischen Umsetzung der Gesamtbeschreibung erzeugt (ob in Schwarzweiß oder Farbe, ist uns egal). Benutze dazu digitale grafische Darstellungen (Grafiken, Fotos) für die Objekttypen von EKAI.
2. Erläutere kurz, welche Mehrdeutigkeiten, Unvollständigkeiten und Widersprüche dein Programm in Gesamtbeschreibungen erkennen kann und nach welchen Prinzipien es sie entscheidet. Diese Prinzipien müssen nicht begründet werden - künstlerische Freiheit ist hier sowieso unvermeidlich.
3. Schicke uns drei Satzfolgen (als unformatierte Textdateien) und die dazu erzeugten Bitmaps (als GIF-, JPEG- oder BMP-Dateien). Eine der drei Satzfolgen sei:

```
Ich hätte gerne das Fahrrad#1 links von Stehlampe#1.  
Der Sessel#1 steht links von Sessel#2.  
Couch#1 befindet sich vor Zimmerpflanze#2.  
Sessel#1 ist neben dem Lagerfeuer#1.  
Ein Sessel#2 steht hinter Lagerfeuer#1.  
Sitzkissen#1 unter Haustier#1.  
Kunstobjekt#1 schwebt über Lagerfeuer#1.  
Die Zimmerpflanze#1 hinter Teddybär#1.  
Fahrrad#1 ist irgendwo rechts von Sessel#2.  
Stehlampe#1 sieht man links von Sessel#1.  
Doch die Zimmerpflanze#2 steht hinter dem Fahrrad#1.
```

Lösungsidee

Vorbemerkung: Für alle, die nicht schon von alleine darauf gekommen sind: Der Name der Künstlerin „EKAI“ ist zum einen eine phonetische Umschrift von ECAI (European Conference on Artificial Intelligence) und zum anderen eine Permutation des Namens eines beliebten schwedischen Einrichtungshauses.

„Gute Stube“ besteht aus drei Teilaufgaben; der Hauptteil 1 der Aufgabe gliedert sich wiederum in drei Teile, die weitgehend getrennt bearbeitet werden:

1. Analyse der Eingabesätze zur Bestimmung räumlicher Relationen,
2. Auflösung möglicher Widersprüche, Unvollständigkeiten und Mehrdeutigkeiten zwischen den Relationen und
3. grafische Umsetzung der verbleibenden räumlichen Beziehungen.

Zur Lösung der ersten dieser Teilaufgaben muss zunächst eingeschränkt werden, wie eine gültige Eingabe aussieht:

- Jede Eingabezeile wird als ein Satz betrachtet.
- Ein Satz besteht aus einem Objekt gefolgt von einer Relation gefolgt von einem zweiten Objekt, wobei diese zwei Objekte unterschiedlich sein müssen.
- Ein Objekt ist ein Einrichtungsgegenstand direkt gefolgt von einer Raute und einer Nummer.
- Groß- und Kleinschreibung wird nicht beachtet.
- Vor, zwischen und nach den beiden Objekten und der Relation dürfen beliebige andere Zeichenfolgen (also auch andere Objekte und Relationen) vorkommen.
- Sätze, die nicht die oben beschriebene Form haben, werden einfach verworfen.

Sätze dieser Form lassen sich leicht durch reguläre Ausdrücke beschreiben. Erkannte Sätze werden als Drei-Tupel (*Relation, Objekt, Objekt*) abgelegt. Außerdem werden alle Objekte, die verwendet wurden, in einer Liste eingetragen.

Der zweite Teil der Aufgabe ist es, Widersprüche zwischen den eingelesenen Relationen zu entfernen, Mehrdeutigkeiten zu entscheiden und Unvollständigkeiten zu ergänzen. Dies wird dadurch erreicht, dass jedem Objekt eine eindeutige Position zugewiesen wird.

Weiterhin müssen die Relationen genauer definiert werden. „Rechts“ und „links“ beziehen sich auf rechts und links vom Betrachter aus. „Neben“ bedeutet so viel wie „rechts oder links“.

„Rechts“, „links“ und „neben“ decken die (horizontale) x-Achse ab. „Über“ und „unter“ beziehen sich auf die (vertikale) y-Achse, „vor“ und „hinter“ auf die z-Achse (räumliche Tiefe). Alle Relation werden als direkt interpretiert, d.h. „direkt links von“ oder „direkt über“ usw.

Die einzelnen Objekte werden nacheinander platziert, wobei zunächst solche platziert werden, die in möglichst vielen Dimensionen möglichst viele Relationen mit anderen Objekten haben.

Ist das zu platzierende Objekt gewählt, so werden alle Relationen, die dieses Objekt betreffen, abgearbeitet: Ist die Relation bereits erfüllt (lautet die Relation zum Beispiel „a links von b“ und „a“ ist bereits irgendwo links von „b“), dann wird diese Relation als bearbeitet abgehakt. Ist die Relation noch nicht erfüllt, so wird das Objekt entsprechend der Relation platziert. Ist an der Stelle bereits ein anderes Objekt, so werden dieses und alle auf der gleichen Seite platzierten Objekte verschoben. Gibt es mehrere Stellen, an denen das Objekt eingefügt werden könnte, so wird es zunächst auch an all diesen Stellen eingetragen. Auf diese Weise werden zunächst Mehrdeutigkeiten behandelt. Eine Relation wird grundsätzlich nur einmal bearbeitet, so dass Widersprüche wie Mehrdeutigkeiten behandelt werden, indem auch hier zunächst alle möglichen Varianten eingetragen werden. Würde nämlich eine Relation mehrmals bearbeitet werden können, so würden Relationen wie „a links von b“ und „b links von a“ zu Endlosschleifen führen. Es kann vorkommen, dass beide Objekte, die an der Relation beteiligt sind, noch nirgends platziert worden sind. Dies ist dann eine Unvollständigkeit in der Relation. Das wird dadurch gelöst, dass ein Objekt an einer möglichst zentralen Stelle einfach erzeugt und das andere entsprechend angefügt wird.

Erst, wenn ein Objekt in keiner weiteren Relation genannt wird, wird ihm eine eindeutige Position zugewiesen, nämlich die von allen möglichen am zentralsten gelegene.

Nachdem alle Relationen abgearbeitet sind, ist also allen Objekten eine eindeutige Position zugewiesen. Mit diesen Informationen kann nun der dritte Teil der Aufgabe, die Ausgabe als Grafik, leicht gelöst werden.

Zunächst wird die Anzahl der unterschiedlichen Plätze ermittelt, die für die x-Achse und die y-Achse benötigt werden. Dann werden die Objekte von hinten nach vorne, also in einer einfachen Guckkastenperspektive, gezeichnet.

Programm-Dokumentation

Als Programmiersprache wurde Perl gewählt, eine weit verbreitete Skriptsprache. Für das Erzeugen des Bildes wurde die ImageMagick Bibliothek verwendet, die ebenfalls weit verbreitet ist.

Genau wie auch die Aufgabenstellung ist das Programm in drei Teile gegliedert, die in den Funktionen `eingabe()`, `strukturaufbau()` und `ausgabe()` zusammengefasst sind.

Teil I: Einlesen und Analysieren der Eingabe

Die Eingabe wird, wie in der Lösungsidee beschrieben, zeilenweise entweder von der Tastatur oder ggf. von einer Datei, deren Namen als Parameter übergeben wurde, eingelesen (Perl öffnet eine als Parameter übergebene Datei automatisch als Standardeingabe).

Jede eingelesene Zeile wird gegen den regulären Ausdruck

```
/\b($regobj\b#\d+)\b.+?\b($regrel)\b.+?\b($regobj#\d+)\b/
```

verglichen. Dabei ist `\b` eine Wortgrenze, `\d` ist eine Ziffer und `$regobj` und `$regrel` enthalten alle möglichen Objekte, bzw. Relationen.

Der reguläre Ausdruck wird dabei unabhängig von der Groß- und Kleinschreibung angewendet. Dazu ist es aber nötig, eine entsprechende Ländereinstellung (`locale`) installiert zu haben, um Umlaute richtig zu behandeln.

Wurde eine Zeile erkannt, so wird diese als Drei-Tupel im Array `@beschreibungen` abgelegt und für die zwei beteiligten Objekte vermerkt, auf welche Ebene (`x`, `y` oder `z`) sich diese Relation bezieht. Diese Information wird später benötigt und kann hier einfach ermittelt werden.

Die Objekte selber sind in dem Hash `%objekte` gespeichert, wobei ihr Name als Schlüssel verwendet wird. Jeder Eintrag enthält Informationen darüber, wieviele Relationen in welchen Ebenen das Objekt betreffen (`rel`) und an welchen Orten dieses Objekt sein könnte (`orte`).

Die Eingabe endet mit dem Ende der Datei oder der Eingabe des end-of-file Zeichens (meist `^D`).

Teil II: Ermitteln einer Platzierung für jedes Objekt

Wahl der Datenstruktur Für diesen Teil ist eine etwas größere Datenstruktur nötig, in der die Objekte gespeichert werden. Es sollte einfach möglich sein, Objekte zu verschieben, zu löschen und wiederzufinden. Außerdem müssen freie Ebenen zwischen Objekten eingefügt und entfernt werden können.

Dazu soll folgende Datenstruktur verwendet werden:

- Die Objekte werden in einem dreidimensionalen Raster gespeichert, dies wird durch ein entsprechendes dreidimensionales Array erreicht. Dieses Array heißt `%raster`. Dabei handelt es sich um ein assoziatives Array, einen sogenannten Hash. Dies hat den Vorteil, dass auf die einzelnen Stellen im Raster, die ein Objekt enthalten, schnell zugegriffen werden kann.

Befindet sich ein Objekt an der Stelle (x, y, z) im Raster, so enthält `$raster{x}{y}{z}` einen Verweis auf dieses Objekt.

2. Ansonsten wird das Objekt nach vorne sortiert, das an mehr Relationen beteiligt ist.
3. Sind beide Objekte an gleich vielen Relationen beteiligt, so wird dasjenige bevorzugt, dessen Relationen möglichst viele Ebenen (also x -, y -, oder z -Ebene) betreffen.

In anderen Worten, es wird das Objekt gewählt, das entweder bereits zum Teil bearbeitet ist, oder möglichst viele Relationen und möglichst viele Ebenen hat.

Ist nun ein Objekt gefunden, das bearbeitet werden soll, so werden zunächst alle Relationen abgearbeitet, die sowohl dieses Objekt betreffen als auch ein Objekt, das bereits im Raster eingetragen ist. Dadurch wird erreicht, dass die Objekte möglichst kompakt verteilt sind. Für jede solche Relation wird geprüft, ob sie bereits erfüllt ist. Ist also die Relation (``rechts von'', $\$objekt$, $\$partner$) (dabei ist $\$objekt$ das Objekt, das gerade bearbeitet wird), so wird geprüft, ob sich $\$objekt$ irgendwo rechts von $\$partner$ befindet. Ist dies nicht der Fall, so wird es genau rechts neben $\$partner$ eingefügt. Wie ein Objekt an einer bestimmten Stelle eingefügt wird, wird weiter unten detaillierter beschrieben. Die Relation „neben“ wird dabei einfach zweimal bearbeitet, einmal als „links von“ und einmal als „rechts von“

Wurden alle solchen Relationen abgearbeitet, oder wurden keine solchen gefunden, so wird überprüft, ob das Objekt, das bearbeitet wird, zwischenzeitlich mindestens eine mögliche Position innerhalb des Rasters hat. Ist dies nicht der Fall, so wird es an einer möglichst zentralen Stelle eingefügt. Auch das ist weiter unten detaillierter beschrieben. Wenn noch gar kein Objekt existiert (das ist beim Aufruf von `strukturaufbau()` der Fall), dann wird an dieser Stelle das Objekt an der Position (0, 0, 0) erzeugt.

Nun werden die übrigen Relationen bearbeitet, die das Objekt betreffen. Das sind solche, bei denen sich der Partner in der Relation noch nicht im Raster befindet. Ansonsten ist das Vorgehen das gleiche wie oben. Nach einem Durchlauf wurden also für ein bestimmtes Objekt alle Relationen bearbeitet.

Anschließend werden alle Objekte, die an keinen weiteren Relationen beteiligt sind, auf jeweils ein Vorkommen reduziert. Dies ist also zumindest das Objekt, das gerade bearbeitet wurde, und unter Umständen auch noch andere.

Nun werden die Objekte erneut sortiert und das nächste bearbeitet. Nachdem diese Schleife für alle Objekte durchlaufen ist, sind also alle Relationen abgearbeitet und alle Objekte haben eine eindeutige Position zugewiesen bekommen.

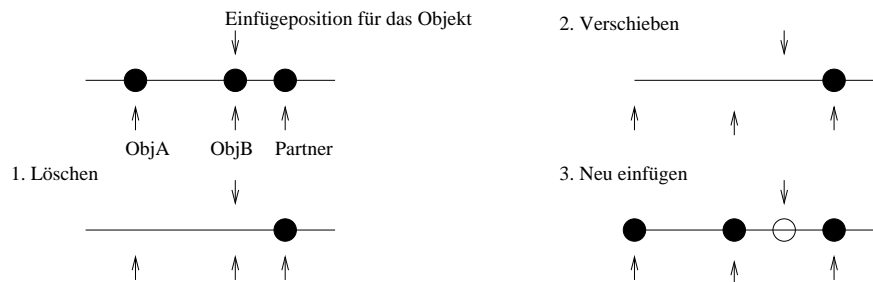
Einfügen eines Objekts Die Aufgabe, ein Objekt in Relation zu einem anderen im Raster einzufügen, wird von der Funktion `positioniere()` übernommen.

Dabei wird zunächst geprüft, ob die Stelle, an der das Objekt eingefügt werden soll, noch frei ist (es sind dabei nur eindeutige Relationen zugelassen, also nicht „neben“). Ist dies der Fall, so wird das Objekt einfach dort eingetragen.

Ist die Stelle belegt, so werden alle Objekte ab dieser Position in Richtung der Relation verschoben. Dies wird durch die Funktion `verschiebe()` bewerkstelligt. Zunächst werden alle

Objekte, die verschoben werden sollen aus dem Raster gelöscht. Die Objekte selber behalten aber die Information, wo sie sich befunden haben. Diese Information wird dann in die gegebene Richtung geändert. Anschließend werden die Objekte an ihren neuen Positionen im Raster wieder eingetragen. Die Information, die in dem Feld @achsen gespeichert ist, wird einfach entsprechend verschoben.

Im eindimensionalen wäre das dieser Vorgang:

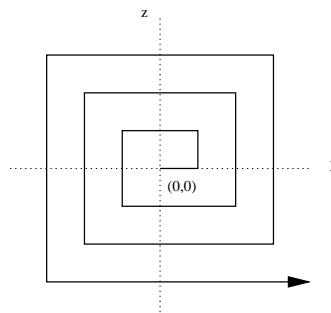


Danach ist die benötigte Position wieder frei und das Objekt kann an dieser Stelle eingetragen werden.

Erzeugen eines Objekts an einer zentralen Stelle Ein Objekt, das neu eingefügt wird, soll an einer möglichst zentralen, aber freien Stelle eingefügt werden, so dass sowohl die Gesamtverteilung nicht zu sehr gestreut wird als auch die anderen Objekte nicht verschoben werden müssen. Diese Aufgabe wird von der Funktion `neueintragen()` übernommen.

Dazu wird zunächst mit Hilfe des Arrays @achsen festgestellt, welche x -Ebene noch am wenigsten belegt ist (dabei werden solche bevorzugt, die weiter in der Mitte liegen). Ist ein solches x gefunden, muss auf der zugehörigen y/z -Ebene eine möglichst zentrale, freie Stelle gefunden werden. Dazu wird $(y, z) = (0, 0)$ gesetzt und von dort aus spiralförmig nach außen gegangen. Die erste freie Stelle wird dann dem Objekt zugewiesen.

Diese spiralförmige Bewegung wird mit einem Vektor realisiert. In eine Richtung werden jeweils k Schritte gemacht, beginnend mit 1. Danach wird der Vektor um 90° gedreht und es werden weitere k Schritte gemacht. Anschließend wird k um eins erhöht und der Vektor um weitere 90° gedreht und es geht weiter:



Reduzieren eines Objekts auf ein Vorkommen Sind für ein oder mehrere Objekte alle Relationen abgearbeitet, so wird für jedes Objekt unter den möglichen Positionen eine ausgesucht, an der sich dieses Objekt endgültig befinden soll. Dies passiert in der Funktion `reduzieren()`.

Zunächst werden alle Vorkommen des Objekts im Raster gelöscht. Dabei wird die Position ermittelt und gespeichert, die am nächsten am Punkt $(0, 0, 0)$ liegt. An dieser Position wird dieses Objekt dann wieder erzeugt.

Nachdem dies für alle betroffenen Objekte gemacht wurde, kann es vorkommen, dass einzelne Ebenen keine Objekte mehr enthalten. Diese Ebenen müssen noch entfernt werden, damit die Objekte möglichst dicht gepackt bleiben. Dies erledigt die Funktion `entferneEbenen()`.

Leere Ebenen lassen sich sehr leicht im Feld `@achsen` finden. Ist ein Eintrag zum Beispiel auf der x -Achse in diesem Feld gleich Null, so ist die y/z -Ebene, die diesen x -Wert hat, leer. Befindet sich solch eine leere Ebene zwischen anderen Ebenen mit Objekten, so kann sie gelöscht werden. Ob sich eine Ebene zwischen anderen Ebenen mit Objekten befindet, kann leicht mit den Feldern `@min` und `@max` getestet werden.

Um eine Ebene zu löschen, werden einfach alle Objekte, die in weiter von der Mitte entfernten Ebenen liegen, rückwärts verschoben. Dies übernimmt die gleiche Funktion, die auch neue Ebenen eingefügt hat.

Teil III: Grafische Ausgabe

Der dritte und letzte Teil ist die grafische Ausgabe der gefundenen Darstellung.

Hierzu wird die Bibliothek ImageMagick verwendet. Sie ist für die meisten Betriebssysteme frei verfügbar und bei den meisten Perl-Installationen bereits vorhanden.

Sollte man sich versucht sehen, das Programm unter einem aus Copyrightgründen nicht genannten Betriebssystem auszuführen, kann es sein, dass ImageMagick zunächst installiert werden muss. Die am weitesten verbreitete Perl-Installation ist hier ActivePerl, der Befehl zum Installieren von ImageMagick lautet

```
ppm install Image-Magick
```

Mit der Struktur, in der die Daten jetzt vorliegen, ist es sehr einfach, eine grafische Darstellung zu generieren. Die Darstellung soll ähnlich einem Guckkasten werden. Objekte mit größerer z -Koordinate sollen hinten, solche mit kleinerer weiter vorne gemalt werden. Die x -Achse wächst nach rechts, die y -Achse nach oben.

Zunächst wird anhand der Informationen in den Feldern `@min` und `@max` ermittelt, welche Dimensionen ein einzelnes Objekt belegen kann. Dann werden die in Frage kommenden Koordinaten von hinten nach vorne, von oben nach unten und von links nach rechts durchlaufen. Falls an einer Stelle ein Objekt gefunden wird, wird dieses an dieser Stelle gezeichnet.

Für jedes Objekt existiert eine GIF-Grafik. Diese Grafiken haben einen transparenten Hintergrund, d.h. Objekte, die hinter anderen in der endgültigen Grafik liegen, können immer noch erkannt werden. Die Namen der GIF-Grafiken ergeben sich aus dem Namen des dargestellten Objekts in Kleinschreibung mit der Erweiterung `.gif`, die Ausgabe erfolgt in die Datei `ausgabe.gif`.

Programm-Ablaufprotokoll

1. Beispiel

Der Ablauf des Programms lässt sich am besten anhand eines einfachen Beispiels zeigen. Praktischerweise spiegelt die Ausgabe des Programmes die Arbeitsweise detailliert wieder. . .

Die Eingabe sei

```
Dies ist eine ung"ultige Eingabezeile.
Teddyb"ar#1 ist neben Haustier#1.
```

Ausgabe des Programms ...

```
$ perl gstube.pl ablauf_beispiel
```

Einlesen des Chatprotokolls:

```
verworfen: 'Dies ist eine ungültige Eingabezeile.'
erkannt:   'teddybär#1 neben haustier#1'
1 Beschreibungen und 2 Objekte eingelesen.
```

Aufbau der Datenstruktur:

```
bearbeite teddybär#1
  erzeuge teddybär#1 bei (0 0 0)
  positioniere haustier#1 neben teddybär#1
  fuege haustier#1 bei (1 0 0) rechts von teddybär#1 ein
  fuege haustier#1 bei (-1 0 0) links von teddybär#1 ein
reduziere haustier#1
  moegliche Position (1 0 0), Entfernung 1
  moegliche Position (-1 0 0), Entfernung 1
  fixiere bei (1 0 0)
```

```
+---+---+ z=0
|TB1|HT1|  0
+---+---+
  0  1
X-Achse: 0..1 = 1 1
Y-Achse: 0..0 = 2
```

Z-Achse: 0..0 = 2

Ausgabe:

darzustellender Bereich: (0 0 0) bis (1 0 0)

Groesse eines Objektes: 200 x 300 Pixel

Bilderzeugung

zeichne teddybär#1 von (0 0 0) an Position (0 0)

zeichne haustier#1 von (1 0 0) an Position (200 0)

Bild in 'ausgabe.gif' gespeichert

... und das erzeugte Bild:



2. Beispiel

Als zweites Beispiel habe ich ein Szenario gewählt, das keine Widersprüche enthält, so dass die Ausgabe dem entsprechen sollte, was man beim Durchlesen der Beschreibung erwartet. Hier die Eingabe

```
couch#1 ist links von zimmerpflanze#1.  
kunstobjekt#1 ist ueber couch#1.  
sessel#1 ist vor zimmerpflanze#1  
sitzkissen#1 ist links von sessel#1.  
stehlampe#1 ist links von sitzkissen#1.
```

Ausgabe des Programms ...

```
$ perl gstube.pl beispiel3
```

Einlesen des Chatprotokolls:

```
erkannt: 'couch#1 links von zimmerpflanze#1'
erkannt: 'kunstobjekt#1 über couch#1'
erkannt: 'sessel#1 vor zimmerpflanze#1'
erkannt: 'sitzkissen#1 links von sessel#1'
erkannt: 'stehlampe#1 links von sitzkissen#1'
5 Beschreibungen und 6 Objekte eingelesen.
```

Aufbau der Datenstruktur:

```
bearbeite couch#1
  erzeuge couch#1 bei (0 0 0)
  positioniere zimmerpflanze#1 rechts von couch#1
  fuege zimmerpflanze#1 bei (1 0 0) rechts von couch#1 ein
  positioniere kunstobjekt#1 über couch#1
  fuege kunstobjekt#1 bei (0 1 0) über couch#1 ein
```

```
+---+---+ z=0
|KO1|   | 1
+---+---+
|CH1|ZP1| 0
+---+---+
  0  1
X-Achse: 0..1 = 2 1
Y-Achse: 0..1 = 2 1
Z-Achse: 0..0 = 3
```

```
bearbeite zimmerpflanze#1
  positioniere sessel#1 vor zimmerpflanze#1
  fuege sessel#1 bei (1 0 -1) vor zimmerpflanze#1 ein
```

```
+---+---+ z=0
|KO1|   | 1
+---+---+
|CH1|ZP1| 0
+---+---+
```

```
+---+---+ z=-1
|   |   | 1
+---+---+
|   |SE1| 0
+---+---+
  0  1
X-Achse: 0..1 = 2 2
Y-Achse: 0..1 = 3 1
Z-Achse: -1..0 = 1 3
```

```
bearbeite sessel#1
  positioniere sitzkissen#1 links von sessel#1
  fuege sitzkissen#1 bei (0 0 -1) links von sessel#1 ein
```

```
+----+----+ z=0
|KO1|   | 1
+----+----+
|CH1|ZP1| 0
+----+----+
```

```
+----+----+ z=-1
|   |   | 1
+----+----+
|SK1|SE1| 0
+----+----+
    0  1
```

X-Achse: 0..1 = 3 2
 Y-Achse: 0..1 = 4 1
 Z-Achse: -1..0 = 2 3

```
bearbeite sitzkissen#1
  positioniere stehlampe#1 links von sitzkissen#1
  fuege stehlampe#1 bei (-1 0 -1) links von sitzkissen#1 ein
```

```
+----+----+----+ z=0
|   |KO1|   | 1
+----+----+----+
|   |CH1|ZP1| 0
+----+----+----+
```

```
+----+----+----+ z=-1
|   |   |   | 1
+----+----+----+
|SL1|SK1|SE1| 0
+----+----+----+
  -1  0  1
```

X-Achse: -1..1 = 1 3 2
 Y-Achse: 0..1 = 5 1
 Z-Achse: -1..0 = 3 3

Ausgabe:

darzustellender Bereich: (-1 0 -1) bis (1 1 0)

Groesse eines Objektes: 133 x 150 Pixel

Bilderzeugung

zeichne couch#1 von (0 0 0) an Position (133 150)

zeichne zimmerpflanze#1 von (1 0 0) an Position (266 150)

zeichne kunstobjekt#1 von (0 1 0) an Position (133 0)

zeichne stehlampe#1 von (-1 0 -1) an Position (0 150)

zeichne sitzkissen#1 von (0 0 -1) an Position (133 150)

zeichne sessel#1 von (1 0 -1) an Position (266 150)

Bild in 'ausgabe.gif' gespeichert

... und das erzeugte Bild:



Gefordertes Beispiel

Als letztes das geforderte Beispiel:

Ausgabe des Programms ...

```
$ perl gstube.pl gefordert
```

Einlesen des Chatprotokolls:

```
erkannt: 'fahrrad#1 links von stehlampe#1'
erkannt: 'sessel#1 links von sessel#2'
erkannt: 'couch#1 vor zimmerpflanze#2'
erkannt: 'sessel#1 neben lagerfeuer#1'
erkannt: 'sessel#2 hinter lagerfeuer#1'
erkannt: 'sitzkissen#1 unter haustier#1'
erkannt: 'kunstobjekt#1 über lagerfeuer#1'
erkannt: 'zimmerpflanze#1 hinter teddybär#1'
erkannt: 'fahrrad#1 rechts von sessel#2'
erkannt: 'stehlampe#1 links von sessel#1'
erkannt: 'zimmerpflanze#2 hinter fahrrad#1'
11 Beschreibungen und 12 Objekte eingelesen.
```

Aufbau der Datenstruktur:

```
bearbeite lagerfeuer#1
  erzeuge lagerfeuer#1 bei (0 0 0)
  positioniere sessel#1 neben lagerfeuer#1
  fuege sessel#1 bei (1 0 0) rechts von lagerfeuer#1 ein
  fuege sessel#1 bei (-1 0 0) links von lagerfeuer#1 ein
  positioniere sessel#2 hinter lagerfeuer#1
```

```
fuege sessel#2 bei (0 0 1) hinter lagerfeuer#1 ein
positioniere kunstobjekt#1 über lagerfeuer#1
fuege kunstobjekt#1 bei (0 1 0) über lagerfeuer#1 ein
```

```
+---+---+---+ z=1
|   |   |   | 1
+---+---+---+
|   |SE2|   | 0
+---+---+---+
```

```
+---+---+---+ z=0
|   |KO1|   | 1
+---+---+---+
|SE1|LF1|SE1| 0
+---+---+---+
-1  0  1
```

X-Achse: -1..1 = 1 3 1

Y-Achse: 0..1 = 4 1

Z-Achse: 0..1 = 4 1

bearbeite sessel#1

```
positioniere sessel#1 links von sessel#2
fuege sessel#1 bei (-1 0 1) links von sessel#2 ein
positioniere stehlampe#1 links von sessel#1
```

verschiebe vorwaerts entlang -X ab 0

sessel#1: (-1 0 0) -> (-2 0 0)

sessel#1: (-1 0 1) -> (-2 0 1)

lagerfeuer#1: (0 0 0) -> (-1 0 0)

sessel#2: (0 0 1) -> (-1 0 1)

kunstobjekt#1: (0 1 0) -> (-1 1 0)

fuege stehlampe#1 bei (0 0 0) links von sessel#1 ein

fuege stehlampe#1 bei (-3 0 0) links von sessel#1 ein

fuege stehlampe#1 bei (-3 0 1) links von sessel#1 ein

reduziere sessel#1

moegliche Position (1 0 0), Entfernung 1

moegliche Position (-2 0 0), Entfernung 2

moegliche Position (-2 0 1), Entfernung 3

fixiere bei (1 0 0)

loesche Ebene X = -2

verschiebe rueckwaerts entlang +X ab -2

sessel#1: (1 0 0) -> (0 0 0)

lagerfeuer#1: (-1 0 0) -> (-2 0 0)

sessel#2: (-1 0 1) -> (-2 0 1)

stehlampe#1: (0 0 0) -> (-1 0 0)

kunstobjekt#1: (-1 1 0) -> (-2 1 0)

```
+---+---+---+ z=1
|   |   |   | 1
+---+---+---+
|SL1|SE2|   | 0
+---+---+---+
```

```

+---+---+---+---+ z=0
|   |KO1|   |   |   | 1
+---+---+---+---+
|SL1|LF1|SL1|SE1| 0
+---+---+---+---+
  -3  -2  -1   0
X-Achse: -3..0 = 2 3 1 1
Y-Achse: 0..1 = 6 1
Z-Achse: 0..1 = 5 2

```

```

bearbeite stehlampe#1
  positioniere fahrrad#1 links von stehlampe#1
  verschiebe vorwaerts entlang -X ab -2
    lagerfeuer#1: (-2 0 0) -> (-3 0 0)
    sessel#2: (-2 0 1) -> (-3 0 1)
    stehlampe#1: (-3 0 0) -> (-4 0 0)
    stehlampe#1: (-3 0 1) -> (-4 0 1)
    kunstobjekt#1: (-2 1 0) -> (-3 1 0)
  fuege fahrrad#1 bei (-2 0 0) links von stehlampe#1 ein
  fuege fahrrad#1 bei (-5 0 0) links von stehlampe#1 ein
  fuege fahrrad#1 bei (-5 0 1) links von stehlampe#1 ein
reduziere stehlampe#1
  moegliche Position (-1 0 0), Entfernung 1
  moegliche Position (-4 0 0), Entfernung 4
  moegliche Position (-4 0 1), Entfernung 5
  fixiere bei (-1 0 0)
loesche Ebene X = -4
  verschiebe rueckwaerts entlang +X ab -4
    sessel#1: (0 0 0) -> (-1 0 0)
    lagerfeuer#1: (-3 0 0) -> (-4 0 0)
    sessel#2: (-3 0 1) -> (-4 0 1)
    stehlampe#1: (-1 0 0) -> (-2 0 0)
    kunstobjekt#1: (-3 1 0) -> (-4 1 0)
    fahrrad#1: (-2 0 0) -> (-3 0 0)

```

```

+---+---+---+---+ z=1
|   |   |   |   |   | 1
+---+---+---+---+
|FR1|SE2|   |   |   | 0
+---+---+---+---+

```

```

+---+---+---+---+ z=0
|   |KO1|   |   |   | 1
+---+---+---+---+
|FR1|LF1|FR1|SL1|SE1| 0
+---+---+---+---+
  -5  -4  -3  -2  -1
X-Achse: -5..-1 = 2 3 1 1 1
Y-Achse: 0..1 = 7 1
Z-Achse: 0..1 = 6 2

```

```

bearbeite fahrrad#1
  positioniere fahrrad#1 rechts von sessel#2
  fuege fahrrad#1 bei (-3 0 1) rechts von sessel#2 ein
  positioniere zimmerpflanze#2 hinter fahrrad#1
  verschiebe vorwaerts entlang +Z ab 1
    sessel#2: (-4 0 1) -> (-4 0 2)
    fahrrad#1: (-5 0 1) -> (-5 0 2)
    fahrrad#1: (-3 0 1) -> (-3 0 2)
  fuege zimmerpflanze#2 bei (-3 0 1) hinter fahrrad#1 ein
  fuege zimmerpflanze#2 bei (-5 0 1) hinter fahrrad#1 ein
  fuege zimmerpflanze#2 bei (-5 0 3) hinter fahrrad#1 ein
  fuege zimmerpflanze#2 bei (-3 0 3) hinter fahrrad#1 ein
reduziere fahrrad#1
  moegliche Position (-3 0 0), Entfernung 3
  moegliche Position (-5 0 0), Entfernung 5
  moegliche Position (-5 0 2), Entfernung 7
  moegliche Position (-3 0 2), Entfernung 5
  fixiere bei (-3 0 0)
  
```

```

+---+---+---+---+---+ z=3
|   |   |   |   |   | 1
+---+---+---+---+---+
|ZP2|   |ZP2|   |   | 0
+---+---+---+---+---+
  
```

```

+---+---+---+---+---+ z=2
|   |   |   |   |   | 1
+---+---+---+---+---+
|   |SE2|   |   |   | 0
+---+---+---+---+---+
  
```

```

+---+---+---+---+---+ z=1
|   |   |   |   |   | 1
+---+---+---+---+---+
|ZP2|   |ZP2|   |   | 0
+---+---+---+---+---+
  
```

```

+---+---+---+---+---+ z=0
|   |KO1|   |   |   | 1
+---+---+---+---+---+
|   |LF1|FR1|SL1|SE1| 0
+---+---+---+---+---+
  
```

```

-5 -4 -3 -2 -1
X-Achse: -5..-1 = 2 3 3 1 1
Y-Achse: 0..1 = 9 1
Z-Achse: 0..3 = 5 2 1 2
  
```

```

bearbeite zimmerpflanze#2
  positioniere couch#1 vor zimmerpflanze#2
  verschiebe vorwaerts entlang -Z ab 0
  
```

```

sessel#1: (-1 0 0) -> (-1 0 -1)
lagerfeuer#1: (-4 0 0) -> (-4 0 -1)
stehlampe#1: (-2 0 0) -> (-2 0 -1)
kunstobjekt#1: (-4 1 0) -> (-4 1 -1)
fahrrad#1: (-3 0 0) -> (-3 0 -1)
fuege couch#1 bei (-3 0 0) vor zimmerpflanze#2 ein
fuege couch#1 bei (-5 0 0) vor zimmerpflanze#2 ein
reduziere zimmerpflanze#2
moegliche Position (-3 0 1), Entfernung 4
moegliche Position (-5 0 1), Entfernung 6
moegliche Position (-5 0 3), Entfernung 8
moegliche Position (-3 0 3), Entfernung 6
fixiere bei (-3 0 1)
reduziere couch#1
moegliche Position (-3 0 0), Entfernung 3
moegliche Position (-5 0 0), Entfernung 5
fixiere bei (-3 0 0)

```

```

+---+---+---+---+ z=2
|   |   |   |   | 1
+---+---+---+---+
|SE2|   |   |   | 0
+---+---+---+---+

```

```

+---+---+---+---+ z=1
|   |   |   |   | 1
+---+---+---+---+
|   |ZP2|   |   | 0
+---+---+---+---+

```

```

+---+---+---+---+ z=0
|   |   |   |   | 1
+---+---+---+---+
|   |CH1|   |   | 0
+---+---+---+---+

```

```

+---+---+---+---+ z=-1
|KO1|   |   |   | 1
+---+---+---+---+
|LF1|FR1|SL1|SE1| 0
+---+---+---+---+

```

```

-4 -3 -2 -1
X-Achse: -4..-1 = 3 3 1 1
Y-Achse: 0..1 = 7 1
Z-Achse: -1..2 = 5 1 1 1

```

```

bearbeite haustier#1
erzeuge haustier#1 bei (0 0 0)
positioniere sitzkissen#1 unter haustier#1
fuege sitzkissen#1 bei (0 -1 0) unter haustier#1 ein

```

```
+---+---+---+---+---+ z=2
|   |   |   |   |   | 1
+---+---+---+---+---+
|SE2|   |   |   |   | 0
+---+---+---+---+---+
|   |   |   |   |   | -1
+---+---+---+---+---+
```

```
+---+---+---+---+---+ z=1
|   |   |   |   |   | 1
+---+---+---+---+---+
|   |ZP2|   |   |   | 0
+---+---+---+---+---+
|   |   |   |   |   | -1
+---+---+---+---+---+
```

```
+---+---+---+---+---+ z=0
|   |   |   |   |   | 1
+---+---+---+---+---+
|   |CH1|   |   |HT1| 0
+---+---+---+---+---+
|   |   |   |   |SK1| -1
+---+---+---+---+---+
```

```
+---+---+---+---+---+ z=-1
|KO1|   |   |   |   | 1
+---+---+---+---+---+
|LF1|FR1|SL1|SE1|   | 0
+---+---+---+---+---+
|   |   |   |   |   | -1
+---+---+---+---+---+
```

```
    -4  -3  -2  -1  0
X-Achse: -4..0 = 3 3 1 1 2
Y-Achse: -1..1 = 1 8 1
Z-Achse: -1..2 = 5 3 1 1
```

```
bearbeite teddybär#1
  erzeuge teddybär#1 bei (-1 0 0)
  positioniere zimmerpflanze#1 hinter teddybär#1
  fuege zimmerpflanze#1 bei (-1 0 1) hinter teddybär#1 ein
```

```
+---+---+---+---+---+ z=2
|   |   |   |   |   | 1
+---+---+---+---+---+
|SE2|   |   |   |   | 0
+---+---+---+---+---+
|   |   |   |   |   | -1
+---+---+---+---+---+
```

```
+---+---+---+---+---+ z=1
|   |   |   |   |   | 1
```

```

+---+---+---+---+---+
|   |ZP2|   |ZP1|   |   | 0
+---+---+---+---+---+
|   |   |   |   |   | -1
+---+---+---+---+---+

```

```

+---+---+---+---+---+ z=0
|   |   |   |   |   | 1
+---+---+---+---+---+
|   |CH1|   |TB1|HT1| 0
+---+---+---+---+---+
|   |   |   |   |SK1| -1
+---+---+---+---+---+

```

```

+---+---+---+---+---+ z=-1
|KO1|   |   |   |   | 1
+---+---+---+---+---+
|LF1|FR1|SL1|SE1|   | 0
+---+---+---+---+---+
|   |   |   |   |   | -1
+---+---+---+---+---+

```

```

-4 -3 -2 -1 0
X-Achse: -4..0 = 3 3 1 3 2
Y-Achse: -1..1 = 1 10 1
Z-Achse: -1..2 = 5 4 2 1

```

Ausgabe:

darzustellender Bereich: (-4 -1 -1) bis (0 1 2)

Groesse eines Objektes: 80 x 100 Pixel

Bilderzeugung

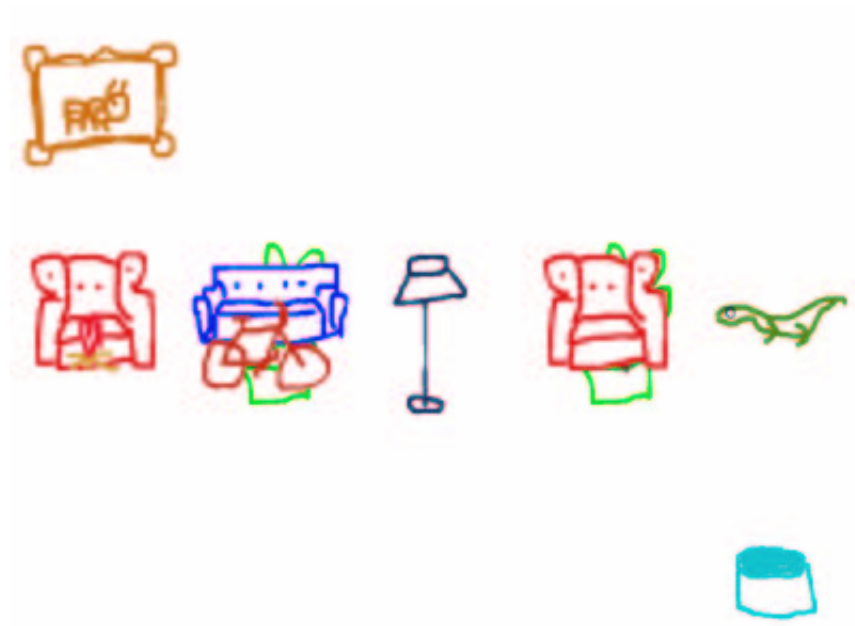
```

zeichne sessel#2 von (-4 0 2) an Position (0 100)
zeichne zimmerpflanze#2 von (-3 0 1) an Position (80 100)
zeichne zimmerpflanze#1 von (-1 0 1) an Position (240 100)
zeichne sitzkissen#1 von (0 -1 0) an Position (320 200)
zeichne couch#1 von (-3 0 0) an Position (80 100)
zeichne teddybär#1 von (-1 0 0) an Position (240 100)
zeichne haustier#1 von (0 0 0) an Position (320 100)
zeichne lagerfeuer#1 von (-4 0 -1) an Position (0 100)
zeichne fahrrad#1 von (-3 0 -1) an Position (80 100)
zeichne stehlampe#1 von (-2 0 -1) an Position (160 100)
zeichne sessel#1 von (-1 0 -1) an Position (240 100)
zeichne kunstobjekt#1 von (-4 1 -1) an Position (0 0)

```

Bild in 'ausgabe.gif' gespeichert

... und das erzeugte Bild:



Programm-Text

```

1  #!/usr/bin/perl -w
    use strict;

    # Deutsche Umlaute korrekt verarbeiten.
5  use locale;
    use POSIX;
    setlocale(LC_ALL, "de_DE");

    # ImageMagick, eine weit verbreitete Grafikbibliothek.
10 # http://www.simplesystems.org/ImageMagick/
    # Installation:
    #   Unix: über die CPAN Shell (perldoc CPAN)
    #   Windows (ActivePerl): ppm install Image-Magick
    use Image::Magick;
15

    ### Konstanten:

    my @OBJEKTE = (                                # Gegenstände
20     "Sessel", "Couch", "Zimmerpflanze", "Kunstobjekt", "Sitzkissen",
        "Fahrrad", "Lagerfeuer", "Teddybär", "Stehlampe", "Haustier",
    );

    my @ACHSEN = qw(X Y Z);                        # Namen der Achsen 0, 1, 2
25
    my %RICHTUNG = (                               # Ordnet jeder Relation die Achse und
        "rechts von" => [0, 1],                  # die Richtung auf dieser Achse zu,

```

```

    "links von" => [0, -1],      # die die Relation darstellen:
    "neben"     => [0,  0],      # relation => [achse, richtung]
30  "über"      => [1,  1],
    "unter"     => [1, -1],
    "hinten"    => [2,  1],
    "vor"       => [2, -1],
);
35
my %UMKEHR = (                  # Ordnet jeder Relation die
    "rechts von" => "links von", # umgekehrte Relation zu.
    "links von"  => "rechts von",
40  "neben"      => "neben",
    "über"       => "unter",
    "unter"      => "über",
    "hinten"     => "vor",
    "vor"        => "hinten",
45  );

### Globale Variablen:

my @beschreibungen; # Beschreibungen in Form von Tripeln:
50  # [relation, objekt1, objekt2]

my %objekte;         # Hash aller Objektinstanzen:
                    # ObjektName => {
                    #   name => Objektname
55  #   rel  => [x,y,z],      Anzahl Relationen in
                    #   orte => [[x,y,z],...] Mögliche Positionen
                    #                               des Objekts
                    # }
60

my %raster;         # 3D-"Array", in dem die Objekte angeordnet werden

my @achsen;        # Vermerkt für [Achse]{Koordinate} die Anzahl der
                    # Objekte, die auf der Achse diese Koordinate haben
65 my (@min, @max); # Vermerkt für jede Achse die maximale bzw. minimale
                    # Koordinate

# Gibt eine ASCII-Grafik aus, die den aktuellen
70 # Zustand der Datenstrukturen wiedergibt.
sub asciiausgabe {
    my ($bild, $xskal, $yskal);
    my %KURZNAMEN = (
75  "sessel"        => "SE",
    "couch"         => "CH",
    "zimmerpflanze" => "ZP",
    "kunstobjekt"   => "KO",
    "sitzkissen"    => "SK",

```

```

    "fahrrad"      => "FR",
80    "lagerfeuer" => "LF",
    "teddybär"    => "TB",
    "stehlampe"   => "SL",
    "haustier"    => "HT"
);
85
for (my $z=$max[2]; $z>=$min[2]; $z--) {
    print("\n");
    for (my $x=$min[0]; $x<=$max[0]; $x++) {
90        print("+---");
    }
    print("+ z=$z\n");
    for (my $y=$max[1]; $y>=$min[1]; $y--) {
        for (my $x=$min[0]; $x<=$max[0]; $x++) {
95            if (defined($raster{$x}{$y}{$z})) {
                my $obj = $raster{$x}{$y}{$z}->{name};
                $obj =~ s/\#(\d+)/;
                print("|$KURZNAMEN{$obj}$1");
            } else {
100                print("|   ");
            }
        }
        printf("| %2d\n", $y);
        for (my $x=$min[0]; $x<=$max[0]; $x++) {
105            print("+---");
        }
        print("+\n");
    }
}
for (my $x=$min[0]; $x<=$max[0]; $x++) {
110    printf(" %3d", $x);
}
print("\n");
for my $achse (0..2) {
    print("$ACHSEN[$achse]-Achse: $min[$achse]..$max[$achse] =");
115    print(" " . $achsen[$achse]{$_}) for $min[$achse]..$max[$achse];
    print("\n");
}
print("\n");
}
120

### Teil 1: Einlesen des Chatprotokolls.

sub eingabe {
125 # Aufbau von regulären Ausdrücken, die ein Objekt bzw.
    # eine Relation erkennen.
    my $regobj = "(?:" . join("|", map {quotemeta} @OBJEKTE) . ")";
    my $regrel = "(?:" . join("|", map {quotemeta} keys(%RICHTUNG)) . ")";

```

```

130   # Eingabe einlesen und auf den Ausdruck matchen
      print("Einlesen des Chatprotokolls:\n");
      while (<>) {
          my ($obj1, $obj2, $rel);

135         chomp;
          ($obj1, $rel, $obj2) = lc =~ /
              \b($regobj#\#d+)\b.+?      # Objekt mit Nummer
              \b($regrel)\b.+?          # Relation
              \b($regobj#\#d+)\b        # Objekt mit Nummer
140         /oxi
          and $obj1 ne $obj2
          or print("   verworfen: '$_'\n"), next;

          # Beschreibung in beiden Objekten vermerken und speichern.
145         print("   erkannt:   '$obj1 $rel $obj2'\n");
          $objekte{$_}{rel}[$RICHTUNG{$rel}[0]]++ for $obj1, $obj2;
          push(@beschreibungen, [$rel, @objekte{$obj1, $obj2}]);
      }

150     # Objekte initialisieren
      for my $obj (keys(%objekte)) {
          $objekte{$obj}{name} = $obj;
          $objekte{$obj}{rel}[$_] ||= 0 for 0..2;
          $objekte{$obj}{orte} = [];

155     }

      print("   " . @beschreibungen . " Beschreibungen und " .
            keys(%objekte) . " Objekte eingelesen.\n\n");
    }

160

    ### Teil 2: Auflösen der Beschreibungen
    # Es werden zunächst alle möglichen Interpretationen der eingegebenen
    # Beschreibungen erzeugt, anschließend wird auf eine einzelne
165     # Darstellung reduziert.

    # Summe aller Argumente
    sub summe { my $sum=0; $sum += $_ for @_; $sum }

170     # Anzahl der Argumente, die != 0 sind
    sub anzahl { scalar grep { $_ } @_ }

    # Trägt ein Objekt ins Raster ein.
    sub trageEin {
175         my ($obj, $ort) = @_;

          push(@{$obj->{orte}}, $ort);          # im Objekt
          $raster{$ort->[0]}{$ort->[1]}{$ort->[2]} = $obj; # im Raster
          for my $i (0..2) {                    # auf den Achsen
180             $achsen[$i]{$ort->[$i]}++;

```

```

    $min[$i] = $ort->[$i]
        if !defined($min[$i]) or $ort->[$i] < $min[$i];
    $max[$i] = $ort->[$i]
        if !defined($max[$i]) or $ort->[$i] > $max[$i];
185     }
    }

# Entfernt ein Objekt aus dem Raster. Der Ort wird nicht aus der Orte-
# Liste des Objekts entfernt, dies ist Aufgabe des Aufrufers.
190 sub trageAus {
    my ($ort) = @_ ;
    delete($raster{$ort->[0]}{$ort->[1]}{$ort->[2]});
    for my $i (0..2) {
        $achsen[$i]{$ort->[$i]}--;
195     $min[$i]++ until $achsen[$i]{$min[$i]} or $min[$i] > $max[$i];
        $max[$i]-- until $achsen[$i]{$max[$i]} or $min[$i] > $max[$i];
        $min[$i] = $max[$i] = undef if $min[$i] > $max[$i];
    }
    }
200 }

# Verschiebt alle Objekte entlang der gegebenen Achse in die gegebene
# Richtung vor oder zurück, wenn sie bereits jenseits oder auf
# der Startposition liegen.
# Beispiele:
205 # verschiebe(0, 1, 5, 1): x=x+1 für obj. x>=5
# verschiebe(0, 1, 5,-1): x=x-1 für obj. x>5, x=5 obj. werden entfernt
# verschiebe(1,-1,-3, 1): y=y-1 für obj. y<=-3
sub verschiebe {
210     my ($achse, $richt, $start, $delta) = @_ ;

    my $schritt = $richt * $delta;
    my $von = $start;
    my $bis = ($richt > 0) ? $max[$achse]+1 : $min[$achse]-1;
    ($von, $bis) = ($bis, $von) if $delta < 0;
215

    print("    verschiebe " .
        (($delta > 0) ? "vorwaerts" : "rueckwaerts") .
        " entlang " . (($richt > 0) ? "+" : "-") .
        "$ACHSEN[$achse] ab $start\n");
220

    # Orte der Objekte entfernen und ggf. aus dem Raster entfernen
    for my $obj (values(%objekte)) {
        for my $ort (@{$obj->{orte}}) {
            next if ($ort->[$achse] <=> $start) == -$richt;
225            print("        $obj->{name}: (@$ort)");
            delete($raster{$ort->[0]}{$ort->[1]}{$ort->[2]});
            $ort->[$achse] += $schritt;
            print(" -> (@$ort)\n");
        }
230     }
}

```

```

# Achsen anpassen, Schleife rückwärts durchlaufen, um nichts
# zu überschreiben.
for (my $i=$bis; $i!=$von; $i--$schritt) {
235     $achsen[$achse][$i] = $achsen[$achse][$i-$schritt];
}

# freigewordene/gelöschte Ebene initialisieren
$achsen[$achse][$von] = 0;
240 for my $m (\@min, \@max) {
    next if !defined($m->[$achse])
        or ($m->[$achse] <=> $start) == -$richt;
    $m->[$achse] += $schritt;
}
245

# Objekte wieder eintragen
for my $obj (values(%objekte)) {
    for my $ort (@{$obj->{orte}}) {
        next if ($ort->[$achse] <=> $start) == -$richt;
250         $raster{$ort->[0]}{$ort->[1]}{$ort->[2]} = $obj;
    }
}

255 # Überprüft, ob sich ein Objekt in einer bestimmten Relation zu
# einer Position befindet. Die Relation "neben" ist nicht erlaubt.
sub inRelation {
    my ($obj, $rel, $pos) = @_;

260     my $achse1 = $RICHTUNG{$rel}[0];
    my $achse2 = ($achse1 + 1) % 3;
    my $achse3 = ($achse1 + 2) % 3;

    for my $ort (@{$obj->{orte}}) {
265         return 1 if
            ($ort->[$achse1] <=> $pos->[$achse1]) == $RICHTUNG{$rel}[1]
            && $ort->[$achse2] == $pos->[$achse2]
            && $ort->[$achse3] == $pos->[$achse3];
    }

270     return 0;
}

# Fügt ein gegebenes Objekt neben seinem Partner ein. Die
275 # genaue Relation muss gegeben sein. Da der Partner an
# mehreren Stellen sein kann, muss der Ort mit angegeben werden.
# Wenn die beiden Objekte bereits in der gegebenen Relation stehen,
# wird keine Aktion durchgeführt.
sub positioniere {
280     my ($obj, $rel, $partner, $ort) = @_;

    if ($rel eq "neben") {

```

```

        positioniere($obj, "rechts von", $partner, $sort);
        positioniere($obj, "links von", $partner, $sort);
285     }
    elseif (!inRelation($obj, $rel, $sort)) {
        my ($achse, $richt) = @{$RICHTUNG{$rel}};
        my $pos = [ @$sort ];
        $pos->[$achse] += $richt;
290
        # wenn die Position belegt ist, Platz schaffen:
        verschiebe($achse, $richt, $pos->[$achse], 1)
            if $raster{$pos->[0]}{$pos->[1]}{$pos->[2]};

295     print("    fuege $obj->{name} bei " .
            "(@$pos) $rel $partner->{name} ein\n");
        trageEin($obj, $pos);
    }
}
300
# Trägt ein Objekt im Raster ein, das noch nirgends eingetragen ist.
sub neueintragen {
    my ($obj) = @_;
    my ($x, $y, $z, $dy, $dz, $k);
305
    # y-z Ebene mit den wenigsten Objekten suchen, möglichst nah an x=0.
    $x = 0; $achsen[0]{0} ||= 0;
    if (defined($min[0]) and defined($max[0])) {
        for (my $i=1; $i<=$max[0] or -$i>=$min[0]; $i++) {
310             for ($i, -$i) {
                $x = $_ if $achsen[0]{$_}
                    and $achsen[0]{$_} < $achsen[0]{$x};
            }
        }
315     }

    #
    #      _____
    #      /      2      \      Spiralförmig in der y-z Ebene um (0,0)
    #      /E---D---C      herumlaufen. Dazu werden jeweils k Schritte
320     #      | |          |1      nach vorne gemacht und dann um 90° nach links
    #      2| F    A-1-B    .      gedreht.
    #      | |          .      k erhöht sich nach jeder zweiten Drehung.
    #      \G---H---I---J
    #      \_____ /
325     #
    #      3
    $k = 1;
    ($y, $z) = (0, 0);
    ($dy, $dz) = (1, 0);
    while (1) {
330         for (1..$k) {
            unless (exists($raster{$x}{$y}{$z})) {
                print("    erzeuge $obj->{name} bei ($x $y $z)\n");
                trageEin($obj, [$x,$y,$z]);
            }
        }
    }
}

```

```

        return;
335     }
        $y += $dy; $z += $dz;
    }
    ($dy, $dz) = (-$dz, $dy);
    $k++ if $dy != 0;
340 }
}

# Entfernt alle Ebenen auf denen keine Objekte sind.
sub entferneEbenen {
345   for my $achse (0..2) {
        for (my $i=$min[$achse]; $i<=$max[$achse]; $i++) {
            unless ($achsen[$achse]{$i}) {
                print("  loesche Ebene $ACHSEN[$achse] = $i\n");
                verschiebe($achse, 1, $i--, -1);
350             }
        }
    }
}

355 # Reduziert Objekte, die keine unbearbeiteten
# Relationen mehr haben, auf einen Ort.
sub reduzieren {
    for my $obj (values(%objekte)) {
        next if summe(@{$obj->{rel}}) or @{$obj->{orte}} == 1;
360     print("  reduziere $obj->{name}\n");

        my ($min, $pos);
        while (my $sort = shift(@{$obj->{orte}})) {
            my $abst = summe(map { abs } @$sort);
365     print("    moegliche Position (@$sort), Entfernung $abst\n");
            $min=$abst, $pos=[ @$sort ] if !defined($min) or $abst < $min;
            trageAus($sort);
        }

370     print("    fixiere bei (@$pos)\n");
        trageEin($obj, $pos);
    }

    entferneEbenen();
375 }

# Liefert das Objekt, das als nächstes bearbeitet werden soll,
# um eine möglichst gute Anordnung zu erhalten.
sub naechstes {
380   return
        (sort {
            # Falls beide Objekte (noch) Relationen haben, Anzahl der
            # Orte, an denen die Objekte vorkommen.
            (anzahl(@{$a->{rel}}) > 0 &&

```

```

385     anzahl(@{$b->{rel}}) > 0 &&
        @{$a->{orte}} <=> @{$b->{orte}}
        ||
        # Anzahl der Relationen
        summe(@{$a->{rel}}) <=> summe(@{$b->{rel}})
390     ||
        # Anzahl der Ebenen mit Relationen
        anzahl(@{$a->{rel}}) <=> anzahl(@{$b->{rel}});
        } values(%objekte)[-1];
    }
395 # Aufbau der Datenstruktur.
    sub strukturaufbau {
        print("Aufbau der Datenstruktur:\n");

400     # Jeweils das Objekt mit den meisten Relationen
        # auswählen und dies bearbeiten.
        for (my $obj=naechstes(); anzahl(@{$obj->{rel}}); $obj=naechstes()) {
            print " bearbeite $obj->{name}\n";

405     # Überprüfen, ob das Objekt an ein schon im Raster
        # positioniertes Objekt angefügt werden kann.
        for (my $i=0; $i<@beschreibungen; $i++) {
            my ($rel, $obj1, $obj2) = @{$beschreibungen[$i]};
            next unless $obj1 == $obj or $obj2 == $obj;

410     if ($obj2 == $obj) {
                ($obj1, $obj2) = ($obj2, $obj1);
                $rel = $UMKEHR{$rel};
            }

415     next unless @{$obj2->{orte}};

            # Objekt positionieren
            print " positioniere $obj1->{name} $rel $obj2->{name}\n";
            positioniere($obj1, $rel, $obj2, $_) for @{$obj2->{orte}};

            splice(@beschreibungen, $i--, 1);
            $obj1->{rel}[$RICHTUNG{$rel}[0]]--;
            $obj2->{rel}[$RICHTUNG{$rel}[0]]--;

425     }

            # Wenn das Objekt nirgends angefügt werden konnte und selber noch
            # nicht existiert, wird es an einer "möglichst optisch schönen"
            # Stelle erzeugt.
430     neueintragen($obj) unless @{$obj->{orte}};

            # Alle Beschreibungen bearbeiten, bei denen ein Objekt an
            # das aktuelle angefügt wird.
            for (my $i=0; $i<@beschreibungen; $i++) {
435     my ($rel, $obj1, $obj2) = @{$beschreibungen[$i]};

```

```

next unless $obj1 == $obj or $obj2 == $obj;

if ($obj1 == $obj) {
  ($obj1, $obj2) = ($obj2, $obj1);
440   $rel = $UMKEHR{$rel};
}

print "    positioniere $obj1->{name} $rel $obj2->{name}\n";
445 positioniere($obj1, $rel, $obj2, $_) for @{$obj2->{orte}};

splice(@beschreibungen, $i--, 1);
$obj1->{rel}[$RICHTUNG{$rel}[0]]--;
$obj2->{rel}[$RICHTUNG{$rel}[0]]--;
450 }

# Objekte, die in keiner Beschreibung mehr
# vorkommen, auf einen Ort reduzieren.
reduzieren();

455 # Um den Ablauf besser nachvollziehen zu können...
asciiausgabe();
}

print("\n");
460 }

### Teil 3: Ausgabe der Datenstruktur als Grafik

465 # Ausgabe der Grafik. Die Datenstruktur %raster enthält
# für jedes Objekt eine eindeutige Position.
sub ausgabe {
  my ($bild, $xskal, $yskal);

470   print("Ausgabe:\n");
   print("  darzustellender Bereich: (@min) bis (@max)\n");

   $bild = Image::Magick->new();
   $bild->Read("hintergrund.gif");
475   $xskal = int($bild->get("width") / ($max[0] - $min[0] + 1));
   $yskal = int($bild->get("height") / ($max[1] - $min[1] + 1));
   print("  Groesse eines Objektes: $xskal x $yskal Pixel\n");

   print("  Bilderzeugung\n");
480   for (my $z=$max[2]; $z>=$min[2]; $z--) {
     for (my $y=$min[1]; $y<=$max[1]; $y++) {
       for (my $x=$min[0]; $x<=$max[0]; $x++) {
         next unless defined($raster{$x}{$y}{$z});
485         my $obj = $raster{$x}{$y}{$z};
         my $xpos = ($x - $min[0]) * $xskal;
         my $ypos = ($max[1] - $y) * $yskal;

```

```
print("    zeichne $obj->{name} von ($x $y $z) " .
      "an Position ($xpos $ypos)\n");
490
      $obj = $obj->{name};
      $obj =~ /(\w+)/ and $obj = $1;
      my $sprite = Image::Magick->new();
      $sprite->Read("$obj.gif");
495      $sprite->Resize(width => $xskal, height => $yskal);
      $bild->Composite(image => $sprite, x => $xpos, y => $ypos);
      }
    }
  }
500
  $bild->Write("ausgabe.gif");
  print("  Bild in 'ausgabe.gif' gespeichert\n");
}
505
### Hauptprogramm
eingabe();
if (@beschreibungen) {
  strukturaufbau();
510  ausgabe();
}
```

Aufgabe 4: Verstehst du Bahnhof?

Aufgabenstellung

Jeden Tag treffen ein Güterzug aus Produdorf und einer aus Herstätten in Mittelstadt ein, wo sie zu zwei neuen Zügen zusammengestellt werden, von denen einer nach Konuweiler und einer nach Verberau fährt. Jeder Zug besteht aus einer Lok gefolgt von Waggons. Du möchtest dem Bahnhofsvorsteher von Mittelstadt helfen, das Auseinander- und Zusammenkoppeln der Waggons zu planen.

Die Waggons sind vom Typ A, B, C usw., je nachdem, ob sie Autoklaven, Bumerange, Celli usw. transportieren. Für jeden Typ wird der Tagesbedarf (eine ganze Anzahl Waggons) von Konuweiler und ebenso der von Verberau morgens den anderen Orten telefonisch mitgeteilt, und es wird dafür gesorgt, dass die Züge aus Produdorf und Herstätten zusammen von jedem Typ genau die insgesamt geforderte Anzahl an Waggons enthalten. Ferner kann man davon ausgehen, dass in jedem dieser Züge alle Waggons eines Typs unmittelbar aufeinander folgen, wogegen das für die aus Mittelstadt abfahrenden Züge nicht gefordert wird.

In Mittelstadt werden die Züge für Konuweiler und Verberau aus den eintreffenden Zügen entsprechend dem gemeldeten Tagesbedarf zusammengestellt. Das Auseinander- und Zusammenkoppeln der Waggons ist aufwändig und zeitraubend. Daher sollte die Anzahl der Koppelvorgänge minimiert werden, wobei ein Koppelvorgang zwei Waggons bzw. eine Lok und einen Waggon auseinander- oder zusammenkoppelt. In Mittelstadt stehen hinreichend viele Rangiergleise und Rangierloks zur Verfügung, deren Benutzung „gratis“ ist (Koppelvorgänge zwischen einer Rangierlok und einem Waggon werden nicht gezählt). Die aus Produdorf und Herstätten ankommenden Loks müssen aber nach Konuweiler und Verberau weitergeführt werden.

Aufgabe

Schreibe ein Programm, das den Tagesbedarf von Konuweiler und den von Verberau sowie die in Mittelstadt eintreffenden Züge als Folgen von Waggontypen einliest und eine Folge von Koppel-Anweisungen ausgibt, mit der die beiden in Mittelstadt eintreffenden Züge mit möglichst wenig Koppelvorgängen zu den beiden abfahrenden Zügen zusammengestellt werden können. Teste dein Programm anhand von drei Eingaben, von denen eine die folgende Beispielingabe sein soll (L = Lok):

Zug aus Produdorf: LAAAAAAAAABBBBBBCCCCCDDD
 Zug aus Herstätten: LDDDDDEEEEEEECCCCCCCCAAAA

Tagesbedarf an:	A	B	C	D	E
Konuweiler:	5	0	7	8	3
Verberau:	7	6	8	0	4

Lösungsidee

Nach Eingabe der Züge aus Produdorf und Herstätten sowie des Bedarfes in Konsuweiler und Verberau soll das Programm eine geschickte, also möglichst kurze Folge von Koppelvorgängen ausgeben, mit der die Anhänger entsprechend verteilt werden können. Um diese Aufgabe zu lösen gibt es prinzipiell zwei Lösungswege:

Vollständige Suche Eine systematische Suche unter allen Möglichkeiten, die Wagen der Züge aus Produdorf und Herstätten nach Konsuweiler und Verberau zu verteilen, führt auf jeden Fall zum bestmöglichen Ergebnis. Hierzu werden sämtliche Kombinationen an Koppelstellen überprüft. Diejenige Kombination, die zu einem korrekten Ergebnis führt und ein Minimum an Koppelvorgängen benötigt, ist die Lösung des Problems.

Eine solche vollständige Suche kann aber lange dauern, auch schon beim vorgegebenen Beispiel. Etwas besser ist, die Suche einzuschränken, z. B. indem man folgende Überlegungen einbezieht:

- In einer Gruppe von Wagen (Wagen mit der gleichen Fracht) ist maximal ein Koppelvorgang nötig.
- Wurden Wagen eines Typs in Zug A auseinandergespeert, so ist in Zug B kein Koppelvorgang innerhalb der Wagen des selben Typs notwendig.

Approximativer Algorithmus Alternativ kann man versuchen, die Anzahl der Kopplungen mit Hilfe von geschickten Strategien zu minimieren, und damit einen „approximativen“ Ansatz verfolgen. Der Name sagt, dass mit einer solchen Methode das Optimum angenähert, aber nicht unbedingt gefunden wird. Es ist ein Algorithmus möglich, der sowohl in der Länge der Züge aus Produdorf bzw. Herstätten als auch in der Anzahl der verschiedenen Güter linear ist. Das Ergebnis ist jedoch nicht notwendigerweise optimal.

Ein möglicher approximativer Algorithmus arbeitet in folgender Art und Weise:

1. Betrachte den Zug aus Produdorf.
2. Wähle einen Zielort.
3. Befördere so viele Wagen wie möglich zum Zug an diesen Zielort. Dabei werden immer nur Wagen von vorne betrachtet und dabei geprüft, ob am Zielort noch weiterer Bedarf ist.
4. Ist der Zug aus Produdorf nun vollständig aufgelöst, so kann man den Zug aus Herstätten analog betrachten. Falls nicht: Zielort wechseln und bei Schritt 3 weitermachen.
5. Betrachte den Zug aus Herstätten.
6. Nimm zunächst den in Schritt 2 nicht gewählten Zielort.

7. wie Schritt 3

8. Ist der Zug aus Herstätten nun vollständig aufgelöst, ist die Verteilung abgeschlossen. Ansonsten wechsele den Zielort und mache bei Schritt 7 weiter.

Anmerkung zu Schritt 3: Eine Reihe von Wagen zu befördern beinhaltet zwei Koppelvorgänge, einmal beim Abkoppeln vom Zug (bzw. von den noch verbliebenen Wagen) und einmal beim Ankoppeln an den neuen Zug. Ausnahmen: Die ersten Wagen (samt Lok), die zu einem neuen Zug hinzugefügt werden, benötigen einen Koppelvorgang weniger (kein Ankoppeln an den neuen Zug), und die jeweils letzten Wagen aus Produdorf und Herstätten benötigen ebenfalls einen Koppelvorgang weniger (kein Abkoppeln vom restlichen Zug).

Anmerkung zu Schritt 6: Wichtig ist, dass man hier zunächst den nicht in 2 betrachteten Zielort wählt. Ansonsten werden die ersten Wagen beider Züge (jeweils inklusive Lok) an den gleichen Zielort gelenkt. Die verbleibenden Wagen stünden ohne Lok in Mittelstadt dumm herum.

Anmerkung zu Schritt 7: Es kann durchaus sein, dass der Zug, an den die Lok beim ersten Ausführen dieses Schrittes geführt wird, nicht mehr leer ist (falls schon Wagen aus Produdorf dort eingereiht sind). Für die Ordnungsfreunde: Die Lok kann in diesem Fall über Rangiergleise geleitet und vorne angekoppelt werden. (Siehe Beispiel!)

Der oben beschriebene einfache Algorithmus arbeitet nach dem „Greedy“-Prinzip („greedy“ heißt gierig), weil er immer so viele Wagen wie möglich einem Zielort zuordnet. Der Algorithmus findet auf jeden Fall eine Lösung. Das Ergebnis lässt sich zum Teil noch verbessern, in dem man ihn ein zweites Mal mit vertauschten Eingabezügen anwendet. Für das vorgegebene Beispiel kommt dieser Algorithmus mit 10 Kopplungen aus.

Gieriges Koppeln ist nicht immer optimal

Man kann allerdings für jeden approximativen Algorithmus, der nach dem Greedy-Prinzip arbeitet, ein Gegenbeispiel konstruieren, das auf folgender Überlegung aufbaut: Ein Greedy-Algorithmus „Verstehst Du Bahnhof“ nimmt immer möglichst viele Wagen aus einem der Züge aus Herstätten oder Produdorf und fügt sie einem Zug nach Konsuweiler oder Verberau hinzu. Manchmal kann es aber sinnvoll sein, weniger Wagen zu verschieben. Beispiel:

Zug aus Produdorf: ABC

Zug aus Herstätten: ABA

Bedarf in Konsuweiler: 1B 1C

Bedarf in Verberau: 3A 1B

Ein Greedy-Algorithmus, der zunächst den Zug aus Produdorf betrachtet, wird feststellen, dass schon der erste Wagen nach Verberau gebracht werden muss. Daher wird der Teil AB zum Zug nach Verberau verschoben (1 Koppelvorgang). Der verbleibende Teil (C) wird ohne Koppelvorgang zum Zug nach Konsuweiler hinzugefügt. Anschließend wird der Zug aus Herstätten betrachtet. Die Lok muss nach Konsuweiler (2 Koppelvorgänge). Der nächste Wagen (A) wird

nach Verberau geleitet (2 Koppelvorgänge). Der Bedarf an Waren des Typs B ist in Verberau schon gedeckt, so dass dieser Wagen nach Konuweiler geschickt wird (2 Koppelvorgänge). Der verbleibende Wagen des Typs A wird an den Zug nach Verberau angehängt (1 Koppelvorgang).

Die Teilung sieht also wie folgt aus (Lok ist jeweils durch ein L symbolisiert):

$$\begin{array}{l} \text{LAB} \mid \text{C} \\ \text{L} \mid \text{A} \mid \text{B} \mid \text{A} \end{array}$$

Es werden 8 Koppelvorgänge benötigt.

Eine wesentlich bessere Lösung wäre es, vom ersten Zug lediglich die Lok und den ersten Wagen nach Verberau zu leiten. Dann ergeben sich folgende Koppelstellen:

$$\begin{array}{l} \text{LA} \mid \text{BC} \\ \text{L} \mid \text{ABA} \end{array}$$

Man kommt mit 4 Koppelvorgängen aus.

Mit dem Schema “Es ist günstiger, einen Wagen weniger zu verschieben” kann man für jeden gegebenen Greedy-Algorithmus ein Beispiel konstruieren, für das er nicht optimal arbeitet. Dabei ist es egal, ob der Algorithmus lediglich einen Durchlauf macht, oder ob er alle Variationen aus zunächst betrachteter Zug, zunächst gewählter Zielort und Betrachtung vom Zuganfang bzw. Zugende aus durchprobiert (ggf. muss das Prinzip mehrfach ins Beispiel eingebaut werden) oder ob der Algorithmus die Züge wechselweise abarbeitet (hier reichen ein paar zusätzliche “erzwungene” Wechsel, um die Bearbeitung eines Zuges hinreichend zu verzögern).

Programm-Dokumentation

Das Programm fordert die Eingabe der beiden Züge aus Herstellen und Produdorf als Strings. Jeder Wagen wird als ein Zeichen kodiert. Der Bedarf von Konuweiler wird der Einfachheit halber ebenfalls als String eingelesen. Der Bedarf spezifiziert aber keine Anordnung für den Zug, der nach Konuweiler geschickt wird, er muss nur die richtige Anzahl an Waggons eines jeden Typs beinhalten. Der Bedarf von Verberau entspricht der Gesamtzahl an eingehenden Waggons abzüglich des Bedarfs von Konuweiler und wird in der Prozedur `bedarfsermittlung` entsprechend berechnet.

Die Prozedur `verteil` übernimmt die Hauptarbeit; sie verteilt die Wagen eines eingehenden Zuges auf die beiden Ziele. Sie muss aus dem Hauptprogramm also zweimal aufgerufen werden. Da jede Trennung in `verteil` als zwei Koppelvorgänge gezählt wird, muss der Zähler mit -2 initialisiert werden: Die jeweils ersten Stücke für Konuweiler und Verberau müssen nirgendwo angekoppelt werden.

Programm-Ablaufprotokolle

Für das vorgegebene Beispiel berechnet das Programm die in der folgenden Abbildung gezeigten Kopplungen. Für das vorgegebene Beispiel sind also 10 Kopplungen nötig. Die folgende Tabelle zeigt eine mögliche Koppelfolge ausführlich.

(1)	LAAAAAAAAABBBBBBCCCCCDDD	Zug aus Produdorf
(2)	LDDDDDEEEEEEECCCCCCCCCAAAA	Zug aus Herstellen
(1)	LAAAAAAA ABBBBBCCCCCDDD	Abkoppeln 7/8
(2)	LDDDDDEEEEEEECCCCCCCCCAAAA	
(1)	LAAAAAAA BBBBBBCCCCCDDD	Abkoppeln 8/9, nach Gleis 2
(2)	LDDDDDEEEEEEECCCCCCCCCAAAA A	
(1)	LAAAAAAAABBBBBBCCCCC DDD	Ankoppeln, Abkoppeln 20/21
(2)	LDDDDDEEEEEEECCCCCCCCCAAAA A	
(1)	LAAAAAAAABBBBBBCCCCC	
(2)	LDDDDDEEEEEEECCCCCCCCCAAAA ADDD	Ankoppeln von Gleis 1
(1)	LAAAAAAAABBBBBBCCCCC	
(2)	LDDDDDEEEEADDD EEECCCCCCCCCAAAA	Abkoppeln 8/9, Ankoppeln
(1)	LAAAAAAAABBBBBBCCCCCEEEEECC	Ankoppeln von Gleis 2
(2)	LDDDDDEEEEADDD CCCCCCAAAA	Abkoppeln 18/19, dann ↑
(1)	LAAAAAAAABBBBBBCCCCCEEEEECC	Zug nach Verberau
(2)	LDDDDDEEEEADDDCCCCCCCCCAAAA	Ankoppeln, Zug nach Konuweiler

Zwei weitere Beispiele werden in Form der originalen Programmein- und ausgabe dargestellt:

Zug aus Produdorf: AAAAAABBBBBCCCCDDE
 Zug aus Herstellen: AAAAEEEEEDDD
 Bedarf in Konuweiler: AAAAABBBBBCCCCDDE

Kopplungsvorgänge Zug aus Produdorf:
 Alles bis auf ABBBBCCCCDDE nach Verberau
 Alles bis auf BCCCCDDE nach Konuweiler
 Alles bis auf CCCCDE nach Verberau
 Rest nach Konuweiler

Kopplungsvorgänge Zug aus Herstellen:
 Alles bis auf EEEEEEDDD nach Konuweiler
 Rest nach Verberau

Kopplungsvorgänge insgesamt: 8

Zug aus Produdorf: ABCCDDDDDEEEEEE
 Zug aus Herställen: AAAAABBBBCCCDDE
 Bedarf in Konsuweiler: AAABBBCCDDDEEE

Kopplungsvorgänge Zug aus Produdorf:
 Alles bis auf DEEEEEE nach Konsuweiler
 Alles bis auf EE nach Verberau
 Rest nach Konsuweiler

Kopplungsvorgänge Zug aus Herställen:
 Alles bis auf AABBBBCCCDDE nach Verberau
 Alles bis auf BBCCCDDE nach Konsuweiler
 Alles bis auf E nach Verberau
 Rest nach Konsuweiler

Kopplungsvorgänge insgesamt: 10

Programm-Text

```

1  program verstehst_du_bahnhof;

    uses crt;

5  var bedarf: array['A'..'Z',1..2] of integer;
    pzug,hzug,kzug: string;
    koppelvorgang: integer;
    ziel: integer;

10 const stadt: array[1..2] of string=('Konsuweiler','Verberau');

    function count(s: string; c: char): integer;
    {Zählt das Vorkommen von c in s.}
    var i: integer;
15     anz: integer;
    begin
        anz := 0;
        for i := 1 to length(s) do
            if s[i] = c then inc(anz);
20     count := anz;
    end;

    procedure bedarfsermittlung;
    var c:char;
25  begin
        for c := 'A' to 'Z' do
            begin
                {Der Bedarf von Konsuweiler wurde explizit angegeben.}
                bedarf[c,1] := count(kzug,c);
            end;
        end;
    end;
  
```

```
30      {Die restlichen Wagen gehen dann nach Verberau.}
      bedarf[c,2] := count(pzug,c) + count(hzug,c) - count(kzug,c);
      end;
    end;

35 procedure verteil(zug: string; ziel: integer);
   {Hauptprozedur, die die Wagen eines Zuges auf die Zielstädte verteilt.}
   var typ: char;
       anzahl_wagen: integer;
   begin
40     while zug <> '' do begin
           typ := zug[1];
           anzahl_wagen := count(zug,zug[1]);
           {Wenn die Anzahl der vorhandenen Wagen kleiner ist als der Bedarf,}
           {dann können alle Wagen zum aktuellen Ziel, und es kann mit dem}
45     {gleichen Ziel weitergemacht werden.}
           if anzahl_wagen <= bedarf[typ,ziel] then begin
               dec(bedarf[typ,ziel],anzahl_wagen);
               zug := copy(zug,anzahl_wagen+1,length(zug)-anzahl_wagen);
           end
50     {ansonsten ... }
           else begin
               anzahl_wagen := bedarf[typ,ziel];
               bedarf[typ,ziel] := 0;
               zug := copy(zug,anzahl_wagen+1,length(zug)-anzahl_wagen);
55     writeln('Alles bis auf ',zug,' nach ',stadt[ziel]);
               inc(koppelvorgang,2); { ... wird gekoppelt}
               if ziel=1           {und das Ziel gewechselt.}
                   then ziel := 2
                   else ziel := 1;
           end;
60     end;
           writeln('Rest nach ',stadt[ziel]);
           inc(koppelvorgang);
       end;

65     begin
           clrscr;
           {Einlesen der Züge und des Bedarfs (von Konsuweiler)}
           write('Zug aus Produdorf: ');
70     readln(pzug);
           write('Zug aus Herstätten: ');
           readln(hzug);
           write('Bedarf in Konsuweiler: ');
           readln(kzug);
75     {beim Abkoppeln der vorderen Wagen wird bei jedem Zug ein}
           {Koppelvorgang zuviel gezählt; deshalb mit -2 starten.}
           koppelvorgang := -2;
           bedarfsermittlung;
           {Das Ziel zuerst wählen, zu dem der erste Wagenblock aus Produdorf}
80     {komplett geschickt werden kann.}
```

```
    if count(pzug,pzug[1]) <= bedarf[pzug[1],1]
        then ziel := 1
    else ziel := 2;
    writeln;
85  writeln('Kopplungsvorgänge Zug aus Produdorf:');
    verteil(pzug,ziel);
    {Nun Zug aus Herställen betrachten, vorher Ziel wechseln.}
    if ziel=1
        then ziel := 2
90  else ziel := 1;
    writeln;
    writeln('Kopplungsvorgänge Zug aus Herställen:');
    verteil(hzug,ziel);
    writeln;
95  writeln('Kopplungsvorgänge insgesamt: ',koppelvorgang);
end.
```

Aufgabe 5: Dosen packen

Aufgabenstellung

Ein Industrieroboter an einem Fließband muss so programmiert werden, dass er möglichst viele ankommende zylinderförmige Dosen in einer Schicht in eine 1 m mal 1 m große Kiste packt. Die Durchmesser der Dosen sind unterschiedlich und liegen zwischen 10 cm und 20 cm. Der Roboter muss die Dosen der Reihe nach einpacken.

Aufgabe

- Schreibe eine Funktion „verschiebe“, die eine teilweise gefüllte Kiste, die Position einer Dose in der Kiste (x- und y-Koordinaten) und eine Richtung (einen Winkel) als Eingabe nimmt und die Dose so lange in die gegebene Richtung verschiebt, bis sie an eine schon platzierte Dose oder an den Rand der Kiste stößt. Die abschließende Position der Dose soll ausgegeben werden. Demonstriere die Funktion an drei Beispielen.
- Überlege dir und beschreibe in einem kurzen Text mindestens zwei Methoden, die die Funktion „verschiebe“ verwenden, um eine möglichst gute Packung zu erzielen.

Lösungsidee

Die Hauptfiguren dieser Aufgabe, nämlich die Kiste und die Dosen, sind zwar in Wirklichkeit dreidimensionale Strukturen, doch da nur in einer Schicht gestapelt wird, spielt die Höhe keine Rolle. Das Problem reduziert sich damit auf das Verschieben von Kreisen in einem Rechteck.

Für die Lösung des Verschiebe-Problems gibt es dann zwei prinzipiell unterschiedliche Ansätze. Zum einen kann man die Dose iterativ in im Verhältnis zur Umgebung „kleinen“ Schritten bewegen. Die Dose wird solange stückchenweise verschoben, bis sie im nächsten Schritt ein Hindernis schneiden würde. Bei diesem Ansatz legt man eine Verschiebedistanz als Einheit fest und kann dann die neue Position der Dose berechnen, indem zur x-Koordinate der Cosinus und zur y-Koordinate der Sinus des Verschiebewinkels addiert wird. Bevor diese Position angenommen wird, ist z. B. mit Hilfe des Satzes von Pythagoras zu prüfen, ob keine andere Dose oder der Rand geschnitten wird. Damit dieser Ansatz nicht allzu grob ist, sollte in ausreichend kleinen Schritten verschoben werden: bei einer Kistengröße von 100x100cm sollte die Distanz höchstens 1mm betragen.

Die zweite Möglichkeit besteht darin, alle Positionen in hoher Genauigkeit zu speichern und mit geometrisch hergeleiteten Formeln die maximale Verschiebedistanz und damit sofort die Endposition einer Dose zu berechnen. Diese Variante hat mehr mathematischen Anspruch. Dabei sind Rundungsfehler zu berücksichtigen, die durch die immer noch beschränkte Genauigkeit des Rechnens mit Fließkommazahlen entstehen und das Ergebnis beeinträchtigen können. In dieser Beispiellösung wird der zweite Ansatz verfolgt.

Implementierung der Funktion „verschiebe“

Es wird wie folgt vorgegangen: Bezüglich der vier Außenwände und der anderen Dosen wird unabhängig voneinander berechnet, wie weit die zu verschiebende Dose in die gewünschte Richtung bewegt werden kann. Zum Schluss wird der kleinste *nichtnegative* Wert ermittelt (schlimmstenfalls 0). Um diese Länge wird die Dose dann in die gewünschte Richtung „verschoben“, d. h. die neue Position geometrisch ausgerechnet und dann ausgegeben.

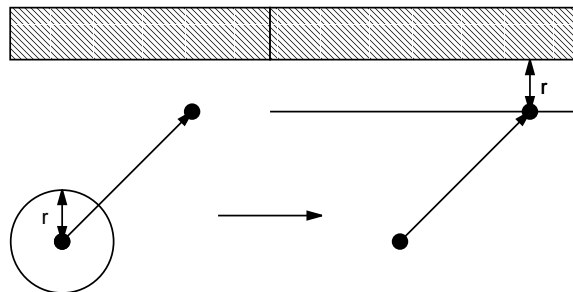
Berechnung der Abstände

Abstand von Dose zu Wand Die Berechnung des Abstands von Dose zu Wand vereinfacht man, indem man die Wand um den Radius der Dose nach innen rückt, um dann nur noch den Schnittpunkt letzterer mit dem Verschiebungsvektor zu schneiden.

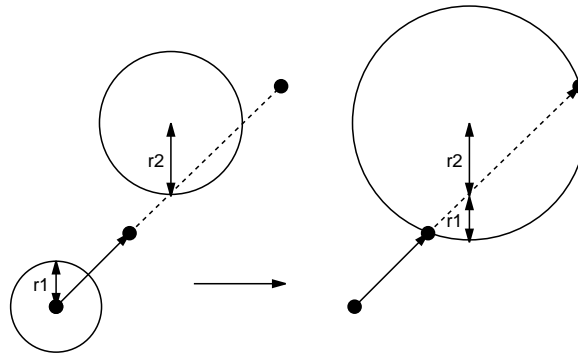
Es ergeben sich folgende Formeln für die maximale Verschiebelänge:

$$l_{oben} = \frac{l_y - y - r}{\sin(\alpha)} \quad l_{unten} = \frac{-y + r}{\sin(\alpha)} \quad l_{links} = \frac{-x + r}{\cos(\alpha)} \quad l_{rechts} = \frac{l_x - x - r}{\cos(\alpha)}$$

l_y ist dabei die Länge, l_x die Breite der Kiste; x , y und r sind Position bzw. Radius der Dose.



Abstand von Dose zu Dose Die Berechnung der Strecke, um die Dose 1 in Richtung Winkel α verschoben werden kann, ohne mit Dose 2 zu kollidieren, lässt sich zurückführen auf den Schnitt des Vektors mit dem Richtungswinkel α vom Mittelpunkt der Dose 1 aus mit einem Kreis um den Mittelpunkt der Dose 2, aber mit dem Radius $r_3 = r_1 + r_2$.



Löst man folgendes Gleichungssystem

$$\begin{aligned}(x - x_m)^2 + (y - y_m)^2 &= r^2 \\ x &= l \cos(\alpha) \\ y &= l \sin(\alpha)\end{aligned}$$

nach der Länge l auf, ergibt sich:

$$l_{1/2} = y_m \sin(\alpha) + x_m \cos(\alpha) \pm \sqrt{(x_m \cos(\alpha) + y_m \sin(\alpha))^2 - x_m^2 - y_m^2 + r^2}$$

Die beiden Lösungen existieren natürlich nur, wenn der Radikand nichtnegativ ist. Ansonsten gibt es keinen Schnittpunkt und somit auch kein Anstoßen. Das Hindernis ist dann also uninteressant.

Ist der Radikand 0, so liegen beide Schnittpunkte übereinander. Das bedeutet, dass sich beide Dosen beim Verschieben nur touchieren.

Neue Position berechnen

Ist die Länge l gefunden, um die die Dose maximal verschoben werden kann, errechnet sich die neue Position ähnlich wie beim iterativen verschiebe-Ansatz:

$$\begin{aligned}x_{neu} &= x_{alt} + l \cos(\alpha) \\ y_{neu} &= y_{alt} + l \sin(\alpha)\end{aligned}$$

Methoden für eine gute Packung

Im zweiten Teil der Aufgabe war verlangt, sich Strategien zur Erzielung einer möglichst hohen Packung zu überlegen. Das Szenario ist wie in der Aufgabenstellung angegeben: Der Industrieroboter bekommt eine nach der anderen Dose angeliefert. Er packt sie so lange in die Kiste, bis

er keine mehr unterbringen kann. Die „Packungsdichte“ definiert sich dann über die von den Dosen bedeckte Fläche. Anhand dieses Szenarios scheiden alle Methoden aus, die eine Vorsortierung der Dosen beinhalten – die Dosen müssen direkt in die Kiste, es gibt keinen Platz zum Zwischenlagern.

Hier zwei Beispiele für sinnvolle Strategien.

Strategie 1 Im ersten Schritt wird versucht, die neue Dose an einer zufälligen Position in die Kiste zu stellen. Da es gut passieren kann, dass dort kein Platz mehr ist, gibt man dem Programm ein gewisse Anzahl Chancen, beispielsweise 100 pro Dose. Konnte die Dose platziert werden, wird sie noch möglichst weit nach außen geschoben, d. h. auf der gedachten Linie zwischen Mittelpunkt der Dose und Mittelpunkt der Kiste. Diese Strategie arbeitet relativ schnell, da sie pro Dose nur einmal die aufwändige *verschiebe*-Funktion aufruft. Aber vor allem für die großen Dosen der Aufgabenstellung funktioniert sie schlecht. Mit kleinen Dosen (Durchmesser kleiner 5cm) erreicht die Strategie immerhin eine Dichte von ca. 45%.

Strategie 2 Neue Dosen werden immer zunächst passgenau in eine Ecke gelegt, sei diese o. B. d. A. die linke untere. Da von hier der Nachschub kommt, muss man sich bemühen, die Dosen von hier aus möglichst weit weg zu schieben. Im ersten Schritt wird jede Dose also zunächst um einen zufälligen Winkel zwischen 0° (nach oben) und 90° (nach rechts) mit Hilfe der Prozedur verschoben. Dann wird dieses Ergebnis noch folgendermaßen verbessert: Man denke sich eine Verbindungslinie zwischen der linken unteren Ecke und dem Mittelpunkt der Dose. Dann wird abwechselnd in die Richtungen verschoben, die senkrecht zur Verbindungslinie stehen, dabei die gedachte Linie immer wieder aktualisiert. Mit welcher Richtung angefangen wird (zuerst rechts oder links), bestimmt ebenfalls der Zufall. Das alternierende Verschieben wird so lange fortgesetzt, bis eine gewisse Verbesserungsgrenze unterschritten wird. Der Clou der Strategie besteht darin, dass sich der Abstand der Dose von der Ecke nie verringert, es gibt also keine Rückschritte. Die Ecke als Startpunkt bietet sich an, da diese nicht so leicht von einer übergroßen Dose überdeckt werden kann. Mit dieser Strategie kann man unabhängig von der Dosengröße Packungsdichten von ungefähr 70% erreichen.

Programm-Dokumentation

Vorbemerkungen

In der Aufgabe war eigentlich nur die Implementierung der Funktion „verschiebe“ gefordert. Aber allein zu Testzwecken muss eine Schnittstelle entwickelt werden, die Beispieldaten aufnehmen, diese an die Funktion „veschiebe“ weiterleiten und das Ergebnis anzeigen kann. Deshalb beinhaltet diese Lösung eine grafische Ausgabe. Außerdem soll die zu verschiebende Dose durch Angabe von Koordinaten identifiziert werden. Es wird hier aber nicht der Mittelpunkt erwartet

(könnte nach einer Verschiebung eine sehr „krumme“ Zahl sein), sondern einfach ein Punkt innerhalb des Dosengrundrisses.

Das Programm erwartet, dass alle übergebenen Daten widerspruchsfrei sind, d. h. keine zwei Dosen sich überschneiden und keine Dose über den Rand hinaus geht bzw. sich ganz außerhalb der Kiste befindet. Diese Randbedingungen werden vom Algorithmus auch erhalten.

Das Programm ist in Java 1.1 geschrieben. Das GUI verwendet das in Java enthaltene AWT zur rudimentären Benutzerführung.

Datenstruktur

Die sich in der Kiste befindlichen Dosen werden in einem Array der Klasse *Dose* gespeichert, als Attribute enthält diese die x - und die y -Koordinate des Mittelpunkts sowie den *radius* der *Dose*. Vom Benutzer wird zwar bei der Eingabe der Durchmesser erwartet, zum Rechnen verwendet man jedoch besser den Radius. Die Umwandlung erfolgt sofort nach der Eingabe.

Die Kiste wird durch nur zwei Größen beschrieben: Die *breite* in x -Richtung und die *laenge* in y -Richtung.

Alle Positions- und Größenwerte werden intern mit *double*-Variablen in Zentimetern repräsentiert. Die Winkel werden in Grad eingegeben, doch intern im Bogenmaß weiterverarbeitet. 0° entspricht „rechts“ auf dem Bildschirm. Das programmeigene Koordinatensystem hat den Ursprung links unten, für die grafische Ausgabe wird auf Bildschirmkoordinaten umgerechnet.

Rundungsfehler

Bei der Gleitkommaarithmetik treten naturgemäß unvermeidbare Rundungsfehler auf, besonders die trigonometrischen Funktionen sind anfällig dafür. Folgendes Problem tritt zu Tage: Liegt eine Dose schon an einer anderen Dose oder der Wand an, errechnet der Algorithmus einen Abstand von ungefähr 0, vielleicht ein bißchen positiv, vielleicht ein bißchen negativ, unabhängig von der Verschieberichtung. Das kann nun in einen Fall entscheidend sein (Wert müsste eigentlich leicht positiv sein, die Dose sitzt fest), im anderen Fall unwichtig (Wert müsste eigentlich leicht negativ sein, die Dose ist frei). Daher muss der Algorithmus bei Werten nahe 0 noch einmal genauer prüfen. Bei der Dose/Dose-Berechnung ist entscheidend, welche Abstände der beiden Schnittpunkte fast 0 sind. Sind es beide, so ist auch deren Abstand fast 0 und die Dose wird nur touchiert, also kein Problem. Ist nur der erste fast 0, so ist die Dose ein Hindernis. Ist nur der zweite fast 0, so liegt das eventuelle Hindernis schon zurück und ist nicht relevant. Bei der Dose/Wand-Kollision wird zunächst geprüft, ob die Dose sich überhaupt in die Richtung der zu überprüfenden Wand bewegt, um dieses Problem zu vermeiden.

Gibt es keinen Schnittpunkt, ist das Ergebnis eigentlich *unendlich*. In diesem Fall wird von den Unterfunktionen *breite+laenge* zurückgegeben, denn weiter als diesen Wert kann die Dose unter keinen Umständen kommen.

Bedienung des Programms

Zu Beginn erscheint links die leere Kiste. Rechts befindet sich der Eingabebereich. Um eine neue Dose in die Kiste zu setzen, muss der Benutzer die Koordinaten des Mittelpunkts und den Durchmesser eingeben und dann auf neue „Dose“ klicken. Falls die neue Dose an dieser Stelle keinen Platz mehr hat, wird in der Statuszeile eine Fehlermeldung ausgegeben und die Dose nicht gesetzt, um keine unerlaubten Konfigurationen zu erhalten.

Um eine Dose zu verschieben, klickt der Benutzer sie zunächst in der Übersicht an. Die aktuell selektierte Dose wird schwarz dargestellt. Nun muss er noch den Winkel in Grad eingeben und „Verschieben“ betätigen. Dann wird die Berechnung ausgeführt und die neue Position grafisch sowie exakt in Zentimetern in der Statuszeile angezeigt.

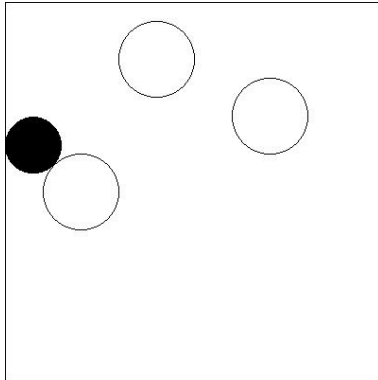
Um das Programm zu beenden, klicke man den gleichnamigen Button.

Hinweise zum Quelltext

<code>class Dose</code>	Repräsentation einer Dose mit <code>x</code> , <code>y</code> (Koordinaten des Mittelpunkts) und <code>radius</code>
<code>class Kiste</code>	Stellt die Kiste grafisch dar und reagiert auf Mausklicks.
<code>class Fenster</code>	Das Hauptfenster. Nimmt die Benutzereingaben entgegen.
<code>class DosenPacken</code>	Klasse des Hauptprogramms. Öffnet ein Fenster mit der Kiste und den Bedienelementen.
<code>Kiste.fastnull(x)</code>	Überprüft, ob das Argument sehr nahe an 0 liegt (festgelegt durch <code>epsilon</code>) und daher verdächtig auf Rundungsfehler ist.
<code>Kiste.absmin(a,b)</code>	Gibt das Minimum der beiden Argumente zurück. Dabei werden allerdings nur positive Argumente betrachtet. Minimal negative Zahlen (fast 0) werden vorher aber zu 0 verfügt und somit respektiert.
<code>Kiste.zuNaheAmRand(Dose)</code>	Überprüft, ob die übergebene Dose komplett innerhalb der Kiste liegt.
<code>Kiste.neueDose(Dose)</code>	Setzt die neue Dose in die Kiste, falls an dieser Position noch Platz ist.
<code>Kiste.verschiebe(winkel)</code>	Die Kernprozedur des Programms. Führt den beschriebenen Algorithmus aus. In der Variable <code>minabstand</code> ist der minimale bisher berechnete Abstand gespeichert (am Anfang noch unendlich).
<code>Fenster.Fenster</code>	Baut das GUI auf und initialisiert das Programm.
<code>Fenster.actionPerformed</code>	Wird von der API aufgerufen, wenn der Benutzer auf einen Button klickt.

Programmablauf-Protokolle

Ablaufprotokoll 1

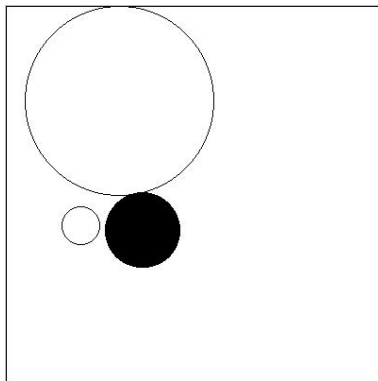


```

Neue Dose gesetzt bei (50,00|50,00) mit Durchmesser 20,00!
Neue Dose gesetzt bei (5,00|35,00) mit Durchmesser 10,00!
Dose mit Winkel 0,00 Grad verschoben nach (95,00|35,00)!
Dose mit Winkel 180,00 Grad verschoben nach (5,00|35,00)!
Dose mit Winkel 1,00 Grad verschoben nach (45,45|35,71)!
Dose mit Winkel 1,00 Grad verschoben nach (45,45|35,71)!

```

Ablaufprotokoll 2

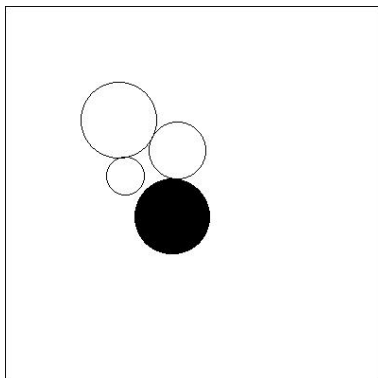


```

Neue Dose gesetzt bei (70,00|70,00) mit Durchmesser 20,00!
Neue Dose gesetzt bei (20,00|50,00) mit Durchmesser 20,00!
Neue Dose gesetzt bei (60,00|40,00) mit Durchmesser 15,00!
Neue Dose gesetzt bei (40,00|85,00) mit Durchmesser 20,00!
Dose selektiert bei (59,25|39,75)
Dose mit Winkel 170,00 Grad verschoben nach (36,49|44,15)!
Dose mit Winkel 75,00 Grad verschoben nach (36,49|44,15)!
Dose mit Winkel 70,00 Grad verschoben nach (45,29|68,32)!
Dose mit Winkel -20,00 Grad verschoben nach (53,09|65,48)!
Dose mit Winkel 175,00 Grad verschoben nach (7,50|69,47)!
Dose mit Winkel -90,00 Grad verschoben nach (7,50|62,25)!

```

Ablaufprotokoll 3



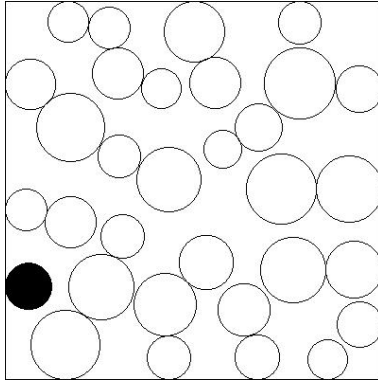
```

Neue Dose gesetzt bei (7,50|7,50) mit Durchmesser 15,00!
Dose selektiert bei (5,25|7,50)
Dose mit Winkel 30,00{!}`} verschoben nach (92,50|56,57)!
Neue Dose gesetzt bei (30,00|70,00) mit Durchmesser 20,00!
Neue Dose gesetzt bei (38,00|50,00) mit Durchmesser 10,00!
Dose selektiert bei (95,00|59,25)
Dose mit Winkel 180,00 Grad verschoben nach (48,63|56,57)!
Dose mit Winkel 135,00 Grad verschoben nach (48,63|56,57)!
Dose mit Winkel 120,00 Grad verschoben nach (45,53|61,94)!
Dose selektiert bei (39,00|50,00)
Dose mit Winkel 140,00 Grad verschoben nach (31,90|55,12)!
Neue Dose gesetzt bei (40,00|20,00) mit Durchmesser 20,00!
Dose mit Winkel 80,00 Grad verschoben nach (44,32|44,48)!

```

Strategie 1

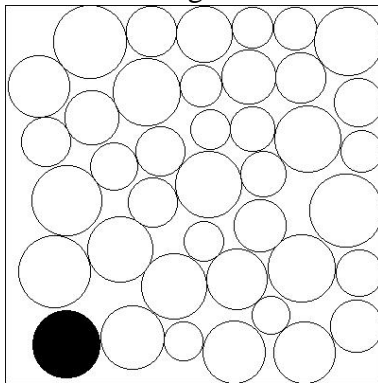
Test der Strategie 1 mit zufällig verteilten Durchmesser zwischen 10 und 20 cm.



Strategie 1: 50,70% Füllung

Strategie 2

Test der Strategie 2 mit zufällig verteilten Durchmesser zwischen 10 und 20 cm.



Strategie 2: 70,12% Füllung

Programmtext

```

1  import java.awt.*;
   import java.awt.event.*;
   import java.text.DecimalFormat;

5  import java.util.Random;

   class Dose {
       double x,y,radius;

10     public Dose(double x,double y,double durchmesser) {
           this.x = x;
           this.y = y;
           //von Durchmesser nach Radius umrechnen
           this.radius = durchmesser / 2;

15     }

```

```

//überlappen sich "this" und "d2"?
public boolean ueberlappung(Dose d2) {
    double abstand = Math.sqrt( Kiste.sqr(x - d2.x)
                                + Kiste.sqr(y - d2.y) );
    return (abstand < (radius+d2.radius));
}

//liegt der Zielpunkt innerhalb des Dosengrundrisses?
public boolean getroffen(double xcm, double ycm) {
    double abstand = Math.sqrt(Kiste.sqr(x - xcm)
                                + Kiste.sqr(y - ycm));
    return abstand <= radius;
}
}

class Kiste extends Canvas implements MouseListener {
    public static final int ppcm = 4;          //Pixel pro cm
    //maximal tolerierter Rundungsfehler:
    public static final double epsilon = 1e-6;
    public static final double pi = 3.141592653589793238;
    public static final double deg2rad = 180.0/pi;
    public double breite,laenge,unendlich;
    private int anzahl;          //Anzahl Dosen
    private int selektiert;      //Index der aktuell selektierten Dose
    private double xcm,ycm;      //Koordinaten des letzten Klicks,
                                //schon umgerechnet nach cm

    Dose dosen[];

    public Kiste(double breite,double laenge) {
        this.breite = breite;
        this.laenge = laenge;
        unendlich = breite + laenge;
        dosen = new Dose[10000];    //maximal 10000 Dosen
        anzahl = 0;
        selektiert = -1;    //noch keine Dose selektiert
        addMouseListener(this);
    }

    //exakte Vergleiche sind gefährlich!
    static boolean fastnull(double x)
        {return Math.abs(x)<=epsilon;}

    //quadrieren
    static double sqr(double x) {return x*x;}

    //liegt x im Intervall [von,bis]?
    static boolean zwischen(double von,double x,double bis)
        {return (von<=x) && (x<=bis);}
}

```

```
double absmin(double a,double b)
{
    if(fastnull(a)) a=0;
70     if (fastnull(b)) b=0;
    //damit uns nicht kaum negative Zahlen unterkommen,
    //die durch Rundungsfehler entstanden sind
    if(a<0) a = unendlich;
    if(b<0) b = unendlich;
75     return Math.min(a,b);
}

public Dimension getPreferredSize() {
80     return new Dimension((int)breite * ppcm, (int)laenge * ppcm);
}

public void zeichneDose(Graphics g,Dose d,boolean selektiert) {
    if(selektiert)
85         g.fillOval((int) ((d.x - d.radius) * ppcm),
                    (int) ((laenge - d.y - d.radius) * ppcm),
                    (int) (d.radius * ppcm * 2),
                    (int) (d.radius * ppcm* 2));
    else
90         g.drawOval((int) ((d.x - d.radius) * ppcm),
                    (int) ((laenge - d.y - d.radius) * ppcm),
                    (int) (d.radius * ppcm * 2),
                    (int) (d.radius * ppcm* 2));
}

95 public void paint(Graphics g) {
    //Hintergrund löschen
    g.drawRect(0, 0, (int)breite * ppcm - 1, (int) laenge * ppcm - 1);
    //alle Dosen der Reihe nach zeichnen
100     for(int i = 0; i < anzahl; i++)
        zeichneDose(g, dosen[i], i == selektiert);
}

//der User hat mit der Maus in die Kiste geklickt
105 public void mouseClicked(MouseEvent e) {
    xcm = (double) e.getX() / (double) ppcm;
    ycm = laenge - (double) e.getY() / (double) ppcm;
    DecimalFormat df = new DecimalFormat("0.00");
    for(int i = 0; i < anzahl; i++)
110         if(dosen[i].getroffen(xcm, ycm))
            {
                selektiert = i;
                System.out.println("Dose selektiert bei (" +
                    df.format(xcm) + "|" + df.format(ycm) + ")");
                break;
115             }
        repaint();
}
```

```

//auf sonstige Mausaktionen nicht reagieren
120 public void mouseEntered(MouseEvent e) { }
public void mouseExited(MouseEvent e) { }
public void mousePressed(MouseEvent e) { }
public void mouseReleased(MouseEvent e) { }

125 //kann die Dose überhaupt so nahe am Rand stehen?
boolean zuNaheAmRand(Dose d) {
    return !zwischen(0.0,d.x - d.radius,breite)
        || !zwischen(0.0,d.x + d.radius,breite)
        || !zwischen(0.0,d.y - d.radius,laenge)
130     || !zwischen(0.0,d.y + d.radius,laenge);
}

//Kollision zwischen Dose d und Dose mit Mittelpunkt (x0,y0) und
//Radius radius0, die mit dem Winkel winkel verschoben werden soll
135 double schnitt(Dose d, double x0, double y0,
    double radius0, double winkel) {
    //Startpunkt relativ zum Mittelpunkt,
    //um die Rechnung zu vereinfachen
    double mx = d.x - x0, my = d.y - y0;
140 double radikand = sqr(my * Math.sin(winkel))
        + 2.0 * mx * Math.cos(winkel)
        * my * Math.sin(winkel)
        + sqr(mx * Math.cos(winkel))
        - sqr(my) - sqr(mx)
145     + sqr(d.radius + radius0);
    //Schnittpunkte liegen aufeinander -> nur Berührung:
    if(radikand <= 0.0 || fastnull(radikand)) return unendlich;
    //Mitte zwischen Schnittpunkten:
    double lm = my * Math.sin(winkel) + mx * Math.cos(winkel);
150 double l1 = lm - Math.sqrt(radikand); //erster Schnittpunkt
    double l2 = lm + Math.sqrt(radikand); //zweiter Schnittpunkt

    if(fastnull(l1)) return 0; //Sonderüberprüfungen
    if(fastnull(l2)) return unendlich;
155 return absmin(l1,l2);
}

boolean neueDose(Dose d) {
160 boolean ul = false; //schon eine Überlappung aufgetreten?

    for(int i = 0; i < anzahl; i++)
        if(d.ueberlappung(dosen[i])) ul = true;
    if(ul || anzahl==10000 || zuNaheAmRand(d)
        || d.radius<5 || d.radius>10)
165     return false; //K.O.-Kriterien
    else {
        selektiert = anzahl;
        if(true) dosen[anzahl]=d;
    }
}

```

```
        anzahl++;
        repaint();
        return true;
    }
}

175 //Die Hauptprozedur
Dose verschiebe(double winkel) {
    if(selektiert == -1) //Keine Dose zum verschieben gefunden
        return null;

180     double minabstand = unendlich;
        Dose d = dosen[selektiert];
        double bogenmass = winkel/deg2rad; //ins Bogenmaß umrechnen

//Kollision mit den anderen Dosen
185     for(int i = 0; i < anzahl; i++)
        if(selektiert!=i)
            minabstand =
190                 absmin(schnitt(dosen[i],d.x,d.y,d.radius,bogenmass),
                            minabstand);

//Kollision mit den Wänden

//Abstand nach oben
195     if(Math.sin(bogenmass) > 0.0) //zeigt nach oben
        minabstand =
            absmin((laenge-d.y-d.radius)/Math.sin(bogenmass),
                minabstand);
//Abstand nach unten
200     if(Math.sin(bogenmass) < 0.0) //zeigt nach unten
        minabstand =
            absmin(-(d.y-d.radius)/Math.sin(bogenmass),
                minabstand);
//Abstand nach links
205     if(Math.cos(bogenmass) < 0.0) //zeigt nach links
        minabstand =
            absmin(-(d.x-d.radius)/Math.cos(bogenmass),
                minabstand);
//Abstand nach rechts
210     if(Math.cos(bogenmass) > 0.0) //zeigt nach rechts
        minabstand =
            absmin((breite-d.x-d.radius)/Math.cos(bogenmass),
                minabstand);

215     //Dose letztendlich verschieben
        dosen[selektiert].x += minabstand*Math.cos(bogenmass);
        dosen[selektiert].y += minabstand*Math.sin(bogenmass);
        repaint();
        return dosen[selektiert];
}
```

```
220     }  
    }  
  
    class Fenster extends Frame implements ActionListener {  
        public static final double pi = 3.141592653589793238,  
225         deg2rad = 180.0/pi;  
        Panel p1 = new Panel(), p2 = new Panel(), p3 = new Panel();  
        Kiste kiste;  
        TextField eX = new TextField(4),  
                eY = new TextField(4),  
230         eDurchmesser = new TextField(3),  
                eWinkel = new TextField(4);  
        Button neu = new Button("Neue Dose setzen"),  
              verschieben = new Button("Verschieben"),  
              strategie1 = new Button("Strategie 1"),  
235         strategie2 = new Button("Strategie 2"),  
              beenden = new Button("Beenden");  
        Label meldung = new Label("Bitte geben Sie die Dosen ein!");  
        Random zufall = new Random();  
        DecimalFormat df = new DecimalFormat("0.00");  
240         private final static double breite=100, laenge=100;  
  
        Fenster() {  
            //Aufbau des GUI  
            p1.setLayout(new GridLayout(8,1,10,10));  
245             p2.add(new Label("x:"));  
                p2.add(eX);  
                p2.add(new Label("y:"));  
                p2.add(eY);  
                p2.add(new Label("Durchmesser:"));  
250             p2.add(eDurchmesser);  
            p1.add(p2);  
            p1.add(neu);  
                p3.add(new Label("Winkel:"));  
                p3.add(eWinkel);  
255             p1.add(p3);  
            p1.add(verschieben);  
            p1.add(strategie1);  
            p1.add(strategie2);  
            p1.add(beenden);  
260             p1.add(meldung);  
            kiste = new Kiste(breite, laenge);  
            add("West", kiste);  
            add("East", p1);  
            neu.addActionListener(this);  
265             strategie1.addActionListener(this);  
            strategie2.addActionListener(this);  
            beenden.addActionListener(this);  
            verschieben.addActionListener(this);  
            setSize(800, 500);  
270             zufall = new Random();
```

```

    }

    public void strategie1Ausfuehren() {
        double flaeche = 0;
        int maxiterationen = 100;
        Dose dose = new Dose(0,0,0);
        while(true) {
            int i;
            double dm = zufall.nextDouble()*10+10; //10cm bis 20cm
            for(i=0; i < maxiterationen; i++) {
                if(kiste.neueDose(dose =
                    new Dose(zufall.nextDouble()*(breite-dm),
                        zufall.nextDouble()*(laenge-dm),dm))
                    break;
            }
            if(i==maxiterationen) break;
            flaeche += pi*dose.radius*dose.radius;
            double winkel =
                Math.atan((dose.y-(laenge/2))/(dose.x-(breite/2)))
                * deg2rad;
            if((dose.x-(breite/2)) < 0) winkel += 180;
            dose = kiste.verschiebe(winkel);
        }
        meldung.setText("" + df.format(flaeche/(breite*laenge)*100)
            + "% Füllung");
        System.out.println("Strategie 1: "
            + df.format(flaeche/(breite*laenge)*100)
            + "% Füllung");
    }

    private void strategie2Ausfuehren() {
        double flaeche = 0, minverbesserung = 2;
        Dose dose;
        double dm = zufall.nextDouble()*10+10; //zwischen 10cm und 20cm
        while(kiste.neueDose(dose = new Dose(dm/2,dm/2,dm)) {
            flaeche += pi*dose.radius*dose.radius;
            dose = kiste.verschiebe(zufall.nextDouble()*90.0);
            double x,y;
            int i=zufall.nextInt(2)*2-1; //+1 oder -1
            do {
                x = dose.x;
                y = dose.y;
                double winkel = Math.atan(dose.y/dose.x)*deg2rad;
                dose = kiste.verschiebe(winkel+90.0);
                winkel = Math.atan(dose.y/dose.x)*deg2rad;
                dose = kiste.verschiebe(winkel-90.0);
            } while ((dose.x*dose.x+dose.y*dose.y)-(x*x+y*y)
                > minverbesserung);
            dm = zufall.nextDouble()*10+10; //zwischen 10cm und 20cm
        }
        meldung.setText("" + df.format(flaeche/(breite*laenge)*100)

```

```
        + "% Füllung");
System.out.println("Strategie 2: "
325         + df.format(flaeche/(breite*laenge)*100)
        + "% Füllung");
    }

    public void actionPerformed(ActionEvent event) {
        if(event.getSource() == neu) { //neue Dose setzen
330            try {
                double x = Double.parseDouble(eX.getText());
                double y = Double.parseDouble(eY.getText());
                double durchmesser =
335                    Double.parseDouble(eDurchmesser.getText());
                if(kiste.neueDose(new Dose(x,y,durchmesser)) {
                    meldung.setText("Neue Dose gesetzt bei (" +
                        df.format(x) + "|" +
                        df.format(y) + ") mit Durchmesser " +
                        df.format(durchmesser) + "!");
340                    System.out.println("Neue Dose gesetzt bei (" +
                        df.format(x) + "|" +
                        df.format(y) + ") mit Durchmesser " +
                        df.format(durchmesser) + "!");
                    }
345                else meldung.setText("Fehler bei der Eingabe!");
            }
            //Beim Parsen der Eingabe könnten Fehler auftreten:
            catch(Exception e) {
                meldung.setText("Fehler bei der Eingabe!");
350                System.out.println("Fehler bei der Eingabe!");
            }
        }
        if(event.getSource() == verschieben) { //Dose verschieben
            try {
355                double winkel = Double.parseDouble(eWinkel.getText());
                Dose jetzt = kiste.verschiebe(winkel);
                if(jetzt != null) {
                    meldung.setText(
360                        "Neue Position ist (" + df.format(jetzt.x) + "|" +
                            + df.format(jetzt.y) + ")!");
                    System.out.println(
                        "Dose mit Winkel " + df.format(winkel) +
                        "Grad verschoben nach (" + df.format(jetzt.x) + "|" +
                        + df.format(jetzt.y) + ")!");
365                }
                else meldung.setText("Bitte wählen Sie eine Dose aus!");
            }
            catch(Exception e) { }
        }
370        if(event.getSource() == strategie1) strategie1Ausfuehren();
        if(event.getSource() == strategie2) strategie2Ausfuehren();
        if(event.getSource() == beenden) System.exit(0);
    }
}
```

```
    }  
  }  
375 public class DosenPacken {  
    public static void main(String args []) {  
      Fenster f = new Fenster(); //Fenster erzeugen  
      f.show();                 //und öffnen  
380    }  
  }
```