

Dokumentation zu den Aufgaben

Von Florian Eisele, Teilnehmer Nr. 42.01

Inhaltsverzeichnis

1. Aufgabe 1.....	1
1.1.Lösungsidee.....	1
1.2.Programm-Dokumentation.....	2
1.2.1.Zu Teilaufgabe 2.....	3
1.3.Programm-Ablaufprotokoll.....	3
1.4.Programm-Text.....	3
2. Aufgabe 2.....	4
2.1.Lösungsidee.....	4
2.2.Programm-Dokumentation.....	5
2.3.Programm-Ablaufprotokoll.....	6
2.4.Programm-Text.....	9
3. Aufgabe 3.....	12
3.1.Lösungsidee.....	12
3.2.Programm-Dokumentation.....	13
3.3.Programm-Ablaufprotokoll.....	14
3.4.Programm-Text.....	16
4. Aufgabe 4.....	25
4.1.Lösungsidee.....	25
4.2.Programm-Dokumentation.....	26
4.3.Programm-Ablaufprotokoll.....	27
4.4.Programm-Text.....	28
5. Aufgabe 5.....	30
5.1.Lösungsidee.....	30
5.1.1.Teilaufgabe 2.....	31
5.2.Programm-Dokumentation.....	32
5.3.Programm-Ablaufprotokoll.....	32
5.4.Programm-Text.....	33
6. Zur CD.....	35

1. Aufgabe 1

1.1.Lösungsidee

Die Idee bei dieser Aufgabe ist es, die Schlange an hintereinander jeder Stelle zu knicken und dann den jeweils längsten Harmonieabschnitt herauszusuchen. Im Prinzip wird hier nur jede denkbare Schlängelung mit jeweils nur einem Knick in der Schlange ausprobiert Geknickt bedeutet hier z.B.: Schlange: AAAGGGUUU wird so geknickt: AAAG|GGUUU, wodurch sie zu:

AAA-G\
UUU-GG

oder zu:
?AA-AG\
UUU-GG/

werden kann. Die Bindestriche deuten die Teile der Schlängelung an, die nicht zum Harmonieabschnitt gehören dürfen, da sie weniger als 3 Buchstaben Abstand haben.

Invertiert man in einem der Teilstrings jeweils jeden Buchstaben durch sein Harmoniekomplementär, dann braucht man nur noch alle gleichen Teilstrings mit gleicher Position

in beiden Substrings finden und daraus den längsten auswählen. Der Algorithmus liefert alle möglichen Harmonieabschnitte, was sich leicht beweisen lässt wenn man zwei bestimmte Abschnitte auswählt. Dann lässt sich nämlich immer Weg finden durch knicken der Schlange an einer bestimmten Stelle beide übereinander zu legen. Bei Harmoniestellen mit gleicher Richtung, z. B. (1–3,6–8) ist bei diesem Algorithmus die physikalische Möglichkeit nicht immer gegeben, da höchst unklar ist, was überhaupt bei der Schlange physikalisch möglich ist (Dehnung, Stauchung, ...). Daher werden solche Möglichkeiten nicht beachtet.

1.2. Programm–Dokumentation

Die konkrete Implementation sieht so aus: Zuerst wird der Beschreibungsstring der Schlange eingelesen und in der Variablen `rna` gespeichert. Dann wird ein String `rnaInv` der selben Größe wie `rna` erstellt und in einer `for`-Schleife jedes Zeichen in `rnaInv` durch das Harmonie–Komplementär des Zeichens mit gleicher Position in `rna` ersetzt. Dabei kann auch direkt getestet werden ob `rna` überhaupt eine regelgetreue Schlangenbeschreibung ist. Es werden dann `rnaInv` und `rna` um jeweils 1 verlängert, wobei als Verlängerungen 'X' und 'Y' benutzt werden, was keinen Einfluss auf den längsten Harmonieabschnitt hat. Der Grund hierfür ist folgender:

Hat man eine Schlange AAACCCCUUU, so sollte sie so geknickt werden, um den längsten Harmonieabschnitt zu erhalten:

```
AAA-CC\
UUU-CC/
```

Im Algorithmus werden aber die Stücke so zurechtgeschnitten, dass:

```
AAC | AAC
UUU | UU
```

oder

```
AAA | AAA
UUC | UC
```

jeweils so übereinander liegen.

Das liegt daran, dass die rechte Hälfte die Länge der linken bekommt und falls die rechte dann kleiner als die linke Hälfte ist diese dann verkleinert wird indem der Anfang weggeschnitten wird. Normalerweise macht das keine Probleme, aber bei Ribonattern mit gerader Länge führt das dazu, dass Anfang und Ende sich nicht mehr treffen. Dem kann man mit dem Anhängen eines Buchstabens Abhilfe verschaffen.

Nun folgt der eigentliche Algorithmus. Das Integer–Array `lha[4]` wird mit {0,0,0,0} initialisiert. Aus `lha` lässt sich der jeweils derzeit längste Harmonieabschnitt wie folgt extrahieren:

```
((lha[0]+1)-(lha[1]+1),(lha[2]+1)-(lha[3]+1)).
```

Die Länge des bis jetzt längsten Harmonieabschnitts ist demnach 0. Nun wird in einer `for`-Schleife der String `rna` in den Unterstring `leftHalf=(0 – c–1)`, und `rnaInv` in `rightHalf=(c+2 – c+2+Länge(leftHalf))`. Die Variable `cut` wird `Länge(leftHalf)–Länge(rightHalf)` gesetzt, was der Länge des Stücks entspricht, dass gleich weggeschnitten wird. `leftHalf` wird nun gleich `leftHalf(Länge(leftHalf) – Länge(rightHalf) – Länge(rightHalf))` gesetzt. Danach wird `rightHalf` umgedreht. Das wird für jedes `c` gemacht, für das dieser Vorgang Sinn macht (siehe Quelltext), prinzipiell kann man aber `c` auch jeden Wert geben. Nun folgt in einer `for`-Schleife die Suche nach dem längsten gemeinsamen Substring mit gleicher Position von `rightHalf` und `leftHalf`. Wenn dessen Länge größer ist als die des Harmonieabschnittes, der derzeit in `lha` steht dann wird er in das Array `int lha[4]` eingetragen. Die Einträge lassen sich mithilfe von `cut`, `c`, `pos` (der Anfangsposition der Harmoniestelle) und `len` (deren Länge) wie folgt berechnen:

```
lha[0] = pos - len + cut;
lha[1] = pos - 1 + cut;
lha[2] = c + 2 + Länge(rightHalf) - pos + len - 1; // Länge(rightHalf) - pos
da rightHalf umgedreht ist;
lha[3] = c + 2 + Länge(rightHalf) - pos;
```

Nun wird `rightHalf` das erste Zeichen weggeschnitten und der Vorgang wird wiederholt (Das gilt nicht rekursiv, es passiert nur beim ersten mal!). Am Ende sollte `lha` die längste Harmoniestelle

enthalten. Diese wird ausgegeben und das Programm wird beendet.

1.2.1. Zu Teilaufgabe 2

Die Lösung dieser Aufgabe hängt vor allem davon ab, was eine mögliche Schlängelung ist. Hier richte ich mich einfach nur streng nach den gegebenen Regeln. Als erstes muss man alle Möglichen Harmoniestellen finden. Das geht wie im Programm aus Teilaufgabe 1, nur dass man hier alle Harmonien finden muss, die bei einer spezifischen Knickung der Schlange auftreten und nicht nur den längsten Harmonieabschnitt. Danach kann das Programm rekursiv vorgehen. Dabei werden alle Harmoniestellen in ein Array gepackt und eine Funktion versucht jeweils die Variante, das n-te Element aus dem Array zu nehmen oder nicht zu nehmen, aus und ruft sich dann selbst für das n+1-te Element aus dem Array in die Lösung auf. Vorher muss sie aufgrund von Regel 2 alle Harmoniestellen streichen, die Ringel der gerade eben in die Lösung einbezogene Harmoniestelle beinhalten (natürlich nur bei der Verzweigung, bei der sie das n-te Element in die Lösung einbezogen hat). Am Ende sollten maximal 2 hoch Länge der Schlange Lösungen rauskommen (von der man sich die Längste aussuchen kann), was darauf hindeutet, dass dieses Problem sehr viel Rechenkraft braucht, weswegen ich vermute, dass deshalb diese Aufgabe nicht implementiert werden musste.

1.3. Programm-Ablaufprotokoll

Schlange:

Hier wird der Beschreibungsstring der Schlange als Eingabe erwartet:

```
GGGAGCGUAGCUCAGUGCGGAGAGCGCCUGCUUUGCACGCAGGAGGUCUGCGGUUCGAUCCCGCGCGCUCCCACC
(1-7,72-66)
```

Der Längste Harmonieabschnitt im Beispiel der Aufgabe ist also (1-7,72-66).

Weitere Beispiele:

Schlange:

```
GGAGCCAUUGAAGCUCCCAUUGGCCAA
(1-5,17-13)
```

Schlange:

```
AAACCCUGCAAGUGCCAUUGACAGGUUUGACACCUGGAC
(22-26,36-32)
```

1.4. Programm-Text

```
1. /**
2.  * Lösung zu:
3.  * BWINF 20, Runde 1, Aufgabe 1
4.  **
5.  * Von Florian Eisele
6.  **/
7.
8. #include <iostream>
9. #include <string>
10. #include <cstdlib>
11. #include <algorithm>
12.
13. using namespace std;
14.
15. int main()
16. {
17.     string rna; // Die Beschreibung der Ribo-Natter;
18.     string rnaInv; // Die Beschreibung der Ribo-Natter, bei der jede Farbe
        // durch ihr Komplement zur Bildung eines
19.     // Harmonieabschnittes ersetzt wurde;
20.     int lha[4] = {0, 0, 0, 0}; // Bei der Suche jeweils der längste
        // Harmonieabschnitt. Format {start1, end1, start2, ende2 };
21.
22.     cout << "Schlange:\n";
```

```

23.  cin >> rna;
24.
25.  // Erstelle rnaInv:
26.  rnaInv.resize(rna.length());
27.  for (int c = 0; c < int(rna.length()); ++c)
28.      switch (rna[c]) {
29.          case 'A': rnaInv[c] = 'U'; break;
30.          case 'U': rnaInv[c] = 'A'; break;
31.          case 'C': rnaInv[c] = 'G'; break;
32.          case 'G': rnaInv[c] = 'C'; break;
33.          default:
34.              cerr << "Die Beschreibung der Schlange enthält
Buchstaben, die sie nicht enthalten darf.\n";
35.              return EXIT_FAILURE;
36.      }
37.  rnaInv += 'X'; rna += 'Y'; // Zur korrekten Funktion des Algorithmuses;
siehe Doku;
38.
39.  // Knicke die Schlange an allen Stellen:
40.  for (int c = 2; (c < int(rna.length()) - 2) && (lha[1] - lha[0] + 1 <
int(rna.length()) - c - 1); ++c) {
41.      string leftHalf = rna.substr(0, c - 1);
42.      string rightHalf = rnaInv.substr(c + 2, leftHalf.length());
43.      reverse(rightHalf.begin(), rightHalf.end());
44.      int cut = leftHalf.length() - rightHalf.length();
45.      if (leftHalf.length() > rightHalf.length())
46.          leftHalf = leftHalf.substr(leftHalf.length() -
rightHalf.length(), rightHalf.length());
47.      int size = rightHalf.length();
48.      for (int d = 0; (d < 2) && (size > 0); ++d) {
49.          int pos = 0;
50.          int len = 0;
51.          while (pos < size + 1) {
52.              if ((leftHalf[pos] == rightHalf[pos]) && (pos != size))
53.                  len++;
54.              else { // Harmonieabschnitt bei pos zuende:
55.                  if (len > lha[1] - lha[0]) {
56.                      lha[0] = pos - len + cut; lha[1] = pos - 1 +
cut;
57.                      lha[2] = c + 2 + rightHalf.length() - pos +
len - 1; lha[3] = c + 2 + rightHalf.length() - pos;
58.                  }
59.                  len = 0;
60.              }
61.              pos++;
62.          }
63.          rightHalf = rightHalf.substr(1, rightHalf.length() - 1);
64.          size--;
65.      }
66.  }
67.
68.  // Gebe längsten Harmonieabschnitt aus:
69.  cout << '(' << lha[0] + 1 << '-' << lha[1] + 1 << ',' << lha[2] + 1 << '-'
<< lha[3] + 1 << ")\n";
70.
71.  return EXIT_SUCCESS;
72.}

```

2. Aufgabe 2

2.1. Lösungsidee

Der erste Schritt zur Lösung ist das Finden einer geeigneten Darstellungsform für das Labyrinth. Hier eignet sich ein 2-dimensionales Array aus 15*15 Bytes, wobei von jedem Byte nur die ersten (*least significant*) 4 Bit geraucht werden. Bei jedem Element steht je ein Bit für die Existenz einer Mauer im Norden, eins für die Existenz einer Mauer im Süden, eins für die Existenz einer Mauer im Westen und eins für die Existenz einer Mauer im Osten. Am Anfang der Suche werden dann alle Felder im Labyrinth gesucht, die gleich dem sind, auf dem sie Maus steht. Ich bezeichne diese

Menge im Folgenden als M . Nun fängt die Maus an, durch Umherlaufen nach später festgelegten Regeln, Felder aus M als mögliche Startfelder auszuschließen, also M zu verkleinern, bis M nur noch aus einem Feld besteht, dem richtigen Startfeld. Das wird in einer While-Schleife implementiert, in der die Maus immer einen wirklichen Schritt macht und denselben Schritt (im Gedächtnis) von jedem Feld in M aus (Die Ergebnismenge bezeichne ich im Folgenden als M_{next}), wobei sie die neue Position nur relativ zur Startposition speichert, da diese relativen Koordinaten bei allen Feldern in M und dem richtigen Feld gleich sind. Nun vergleicht sie die Felder aus M_{next} mit dem Feld, auf dem sie wirklich steht. Die Pendants in M der Felder in M_{next} , die nicht übereinstimmen, werden aus M entfernt. Das wird solange gemacht, bis M nur noch ein Feld enthält, welches dann logischerweise das Startfeld ist.

Damit die Maus aber nicht unnötige Schritte macht (auf ein Feld geht, wo sie schon war, obwohl sie auch auf eins gehen könnte, auf dem sie noch nicht war), was auch zu einer Endlosschleife führen könnte, werden die schon besuchten Koordinaten (Relativ zum Startfeld) in einem 15*15-Array aus Booleans gespeichert und die Maus besucht bevorzugt Noch unbekannte Felder und geht nur auf ein schon bekanntes, wenn keine anderen zur Verfügung stehen. In einem Labyrinth, in dem man von jedem Feld zu jedem anderen kommt führt das dazu, dass sie in kürzstmöglicher Zeit alle Felder im Labyrinth aufgesucht hat (wobei natrlich auch Glück bei der Dauer eine Rolle spielt, da sich der optimale Weg ohne Kenntnis der Startposition nicht bestimmen lässt).

2.2. Programm-Dokumentation

Das Labyrinth wird in einem 15*15-Array aus Chars gespeichert, wobei das Format (wie in 2.1. beschrieben) binär[0000NSWO] ist. Dieses Array wird aus einer Datei ausgelesen, die für das in der Aufgabe spezifizierte Labyrinth wie folgt aussieht: (Druckfehler korrigiert)

[illegible]

Dabei steht jedes "x" für ein Feld, jedes "#" für eine Mauer zwischen den Feldern bzw. zwischen Feld und Rand, zwischen denen es horizontal oder vertikal steht und jedes "+" für keine Mauer bzw. einen Platzhalter. Dabei will ich es aber auch belassen, da das Dateiformat keine allzu gro Rolle spielt.

Das Objekt Labyrinth beinhaltet das 15*15-Array, das das Labyrinth definiert, als Variable

labData. Der Konstruktor von Labyrinth liest das Labyrinth aus der Datei aus und die Funktion simulateMouse(), die die Koordinaten von Maus und Käse als Argumente nimmt und einen Vektor aus Move zurückgibt, der das Verhalten der Maus beschreibt. Dieser wird in main() ausgegeben. Intern arbeitet simulateMouse() übrigens mit Koordinaten von 0 – 14.

Der eben genannte Datentyp Move besteht aus einem Attribut des Types move, wobei es sich um eine Enumeration handelt, die die verschiedenen Schritte der Maus beschreibt (Norden, Süden, Westen, Osten, Geortet, Käse gefunden). Da der Wert Geortet (LOCATED) zusätzliche Parameter benötigt (die Position), hat Move noch die Attribute argX und argY, die beim Schritt 'Geortet' die Startkoordinaten darstellen, ansonsten unbenutzt bleiben.

In simulateMouse() werden nun possible, der Vektor aller möglichen Startfelder, d.h. die, die nicht schon durch die unmittelbare Umgebung der Maus ausgeschlossen werden, und yetVisited, ein Array, in dem alle schon besuchten Felder markiert werden und die Positionsoffsets ox und oy initialisiert.

Dann folgt eine while-Schleife, in der erst einmal getestet wird, ob die Position der Maus zufällig der Position des Essens entspricht, sie also weiss, wo sie ist. Dann wird die jetzige Position der Maus relativ zum Startfeld in yetVisited als besucht markiert. Nun wird das jetzige Feld der Maus in labData herausgenommen und die ersten 4 Bits (die relevanten) werden mit 0xf ge-XOR-t, was mit deren Invertierung gleichkommt. Daher ist jetzt für jede Richtung, in der keine Mauer steht, ein Bit gesetzt. Dieser Wert wird nun mit einer aus yetVisited extrahierten Bitmaske ge-AND-ded, in der für jedes noch nicht besuchtes Nachbarfeld ein Bit gesetzt ist. Das Ergebnis ist also logischerweise eine Bitmaske, in der für jede Richtung, in die die Maus gehen kann und wo sie auf ein Feld kommt, auf dem sie noch nicht war, ein Bit gesetzt, und wird in notYet gespeichert. Sollte diese Maske gleich 0 sein, also alle Felder, auf die die Maus gelangen kann, schon einmal besucht worden sind, so wird sie durch die Bitmaske aller möglichen Schritte der Maus ersetzt (diese kann definitionsgemäß nicht 0 sein, da man im Labyrinth von jedem Feld zu jedem Feld kommen können soll). Nun wird mithilfe von notYet eine Richtung ausgewählt, in die die Maus gehen soll. Das dazu verwendete Verfahren ist irrelevant, in meinem Fall wird einfach das erste gesetzte Bit von rechts genommen. Nun werden die Positionsoffsets aktualisiert, der Schritt wird in die Liste der Aktionen der Maus eingetragen und possible wird verkleinert, d.h. das derzeitige Feld der Maus wird mit denen, auf denen sie von jedem Startfeld aus possible jetzt stünde verglichen und alle nicht übereinstimmenden aus possible entfernt. Der ganze Vorgang wird wiederholt bis possible nur noch ein Element hat.

Nun wird das letzte verbliebene Feld in die Liste der Aktionen der Maus als Aktion "LOCATED" eingetragen, die Liste der Aktionen in einen Vektor umgewandelt, der dann zurückgegeben wird.

Das Ergebnis von simulateMouse() wird dann in main() ausgegeben.

2.3.Programm–Ablaufprotokoll

Beispiel 1

Dies ist das Beispiel aus der Aufgabe.

Labyrinth–Quelldatei: lab01.txt

Startkoordinaten:

x: 14

y: 3

Käsekoordinaten:

x: 3

y: 10

Bis hier hin wurde die Labyrinth–Quelldatei und Start– und Fresskoordinaten eingelesen.

Die Maus verhält sich wie folgt:

```
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Westen
Die Starstelle der Maus war [14,3]
```

Das war eine mögliche Vorgehensweise der Maus, wobei nach dem Lokalisieren der Startstelle abgebrochen wird, da das finden des kürzesten Weges nicht mit zur Aufgabe gehört.

Beispiel 2

Noch einmal dasselbe Labyrinth, diesmal aber mit den Startkoordinaten [4, 1] und den Käsekoordinaten [15, 15].

Labyrinth-Quelldatei: lab01.txt

Startkoordinaten:

x: 4

y: 1

Käsekoordinaten:

x: 15

y: 15

Die Maus verhält sich wie folgt:

Ein Schritt nach Osten

Ein Schritt nach Osten

Ein Schritt nach Norden

Die Starstelle der Maus war [4,1]

Die Ausgabe ist selbsterklärend.

Beispiel 3

Und als letztes Beispiel ein absolut leeres Labyrinth, in dem es nur die Außenmauern gibt. Die Startstelle ist [12, 12], die Käsestelle [3, 3].

Labyrinth-Quelldatei: lab02.txt

Startkoordinaten:

x: 12

y: 12

Käsekoordinaten:

x: 3

y: 3

Die Maus verhält sich wie folgt:

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Westen

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Süden

Ein Schritt nach Westen

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

Ein Schritt nach Norden

[illegible]


```

Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Westen
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Norden
Ein Schritt nach Westen
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Ein Schritt nach Süden
Die Startstelle der Maus war [3,3]
Die Maus frisst

```

Wie man sieht, ist das Verhalten der Maus hier sehr ineffizient und sie kann sich letztlich durch das finden des Käses orientieren. Das die Maus sich so ineffizient verhält liegt an der fehlenden Orientierungshilfe in Form der Mauern und das Fehlen eines Algorithmuses, der ausrechnet, welcher Schritt mit hoher Wahrscheinlichkeit möglichst viele potentielle Startfelder ausschließt.

2.4. Programm-Text

```

1. /**
2.  * Lösung zu:
3.  * BWINF 20, Runde 1, Aufgabe 2
4.  **
5.  * Von Florian Eisele
6.  **/
7.
8. #include <iostream>
9. #include <string>
10. #include <fstream>
11. #include <vector>
12. #include <list>
13. #include <cstdlib>
14. #include <algorithm>
15.
16. using namespace std;
17.
18. enum move { NORTH = 0, SOUTH = 1, WEST = 2, EAST = 3, EAT = 4, LOCATED = 5 };
19.
20. struct Move { // Beschreibt eine Tätigkeit der Maus;
21.     move mov;
22.     int argX, argY; // Koordinaten bei mov = LOCATED;
23.     Move() { }
24.     Move(move _mov, int _argX, int _argY)
25.     { mov = _mov; argX = _argX; argY = _argY; }
26. };
27.

```

```

28.struct Pos { // Eine Position im Labyrinth;
29.    int x, y;
30.    Pos() {}
31.    Pos(int _x, int _y) { x = _x; y = _y; }
32.};
33.
34.class Labyrinth { // Die Representation des Labyrinths;
35.public:
36.    Labyrinth(const char* fd);
37.    vector<Move> simulateMouse(int startX, int startY, int eatX, int eatY);
38.private:
39.    char labData[15][15]; /** labData[x][y] = Binär[????NSWO], wobei N,S,W und
40.        0 (Himmelsrichtungen)
41.        die Existenz der Mauern repräsentieren. */
42.};
43.Labyrinth::Labyrinth(const char* fd) // Liest das Labyrinth aus der Datei fd
    in labData;
44.{
45.    ifstream in(fd);
46.    char field[31][32]; /** Sortiert nach [Zeile, Spalte] */
47.
48.    for (int line = 0; line < 31; ++line)
49.        in.getline(field[line], 32, '\n');
50.
51.    for (int x = 0; x < 15; ++x)
52.        for (int y = 0; y < 15; ++y) {
53.            int l = 29 - 2 * y;
54.            int r = 2 * x + 1;
55.            labData[x][y] = (((field[l - 1][r] == '#') ? 1 : 0) << 3) ^
56.                (((field[l + 1][r] == '#') ? 1 : 0) << 2) ^
57.                (((field[l][r - 1] == '#') ? 1 : 0) << 1) ^ ((field[l][r
58.                + 1] == '#') ? 1 : 0));
59.        }
60.}
61.vector<Move> Labyrinth::simulateMouse(int startX, int startY, int eatX, int
    eatY)
62.    /* Simuliert das Verhalten der Maus.
63.     * Intern werden Koordinaten von 0 - 14 statt von 1 - 15 verwendet;
64.     */
65.{
66.    // Variablen:
67.    list<Move> moves;
68.    vector<Move> ret;
69.    int x = startX - 1, y = startY - 1;
70.    int ox = 0, oy = 0; // Die relativen Positionen (als Offsets);
71.    vector<Pos> possible; // Vektor aller Möglicher Positionen im Labyrinth
    (siehe Doku);
72.    for (int c = 0; c < 15; ++c) // Initialisiere possible;
73.        for (int d = 0; d < 15; ++d)
74.            if (labData[c][d] == labData[x][y]) {
75.                possible.resize(possible.size() + 1);
76.                possible[possible.size() - 1] = Pos(c, d);
77.            }
78.    bool yetVisited[30][30]; // Alle im Rahmen der Suche schon besuchten
    Felder werden hier markiert. Im Array wird mit
79.    // [ox + 15][oy + 15] indiziert, um das Wissen der Maus (Offsets),
    möglichst von dem, was sie nicht weiß (absolute Koord.)
80.    // zu trennen. Prinzipiell wäre nur ein 15*15 Array notwendig.
81.    for (int c = 0; c < 30; ++c) // Initialisiere yetVisited
82.        for (int d = 0; d < 30; ++d)
83.            yetVisited[c][d] = false;
84.
85.    while (possible.size() != 1) {
86.        if ((x + ox == eatX - 1) && (y + oy == eatY - 1)) { // Wenn die Maus
    ihr Essen gefunden hat, weiß sie auch, wo sie ist und es kann
87.            // abgebrochen werden;
88.            moves.push_front(Move(move(LOCATED), eatX, eatY));
89.            moves.push_front(Move(move(EAT), 0, 0));
90.            ret.resize(moves.size());

```

```

90.         reverse_copy(moves.begin(), moves.end(), ret.begin());
91.         return ret;
92.     }
93.     yetVisited[ox + 15][oy + 15] = true; // Markiere das jetzige Feld
    als besucht;
94.     vector<Pos> newPossible(possible.size());
95.     int newPossibleSize = 0;
96.     char notYet = ((yetVisited[15 + ox][15 + oy + 1] ? 0 : 1) << 3) |
    ((yetVisited[15 + ox][15 + oy - 1] ? 0 : 1) << 2) |
97.     ((yetVisited[15 + ox - 1][15 + oy] ? 0 : 1) << 1) |
    (yetVisited[15 + ox + 1][15 + oy] ? 0 : 1); // Maske aller bevorzugten
    Richtungen
98.     notYet &= labData[x + ox][y + oy] ^ 0xf; // Und aller Möglichen
    (invertierte LabData, da in labData die Mauern, jetzt die nicht-Mauern
99.     // gefragt sind.)
100.    if (notYet == 0) // Sollte die Maus kein Feld zur Verfügung haben,
    dass sie noch nicht besucht hat;
101.        notYet = labData[x + ox][y + oy] ^ 0xf;
102.        int mask = 8; move dir = move(NORTH); // Wähle eine Richtung;
103.        while (!(notYet & mask)) {
104.            mask >>= 1; dir = move(int(dir) + 1);
105.        }
106.        moves.push_front(Move(dir, 0, 0)); // Füge die Richtung zum
    Protokoll der Mausebewegungen hinzu;
107.        switch (dir) { // Update die relativen Koordinaten;
108.            case 0: oy += 1; break;
109.            case 1: oy -= 1; break;
110.            case 2: ox -= 1; break;
111.            case 3: ox += 1; break;
112.            default: break; // Nur, um die Compiler-Warnungen zu
    vermeiden...
113.        }
114.        for (unsigned int c = 0; c < possible.size(); ++c) // Generiere die
    Liste der noch möglichen Startfelder;
115.            if (labData[possible[c].x + ox][possible[c].y + oy] ==
    labData[x + ox][y + oy])
116.                newPossible[newPossibleSize++] = possible[c];
117.            newPossible.resize(newPossibleSize);
118.            possible = newPossible;
119.        }
120.        moves.push_front(Move(move(LOCATED), possible[0].x + 1, possible[0].y +
    1)); // Wenn die nicht zwischendurch den Käse gefunden hat, dann
121.        // hat sie sich jetzt, da es nur noch ein mögliches Startfeld gibt,
    geortet.
122.
123.    ret.resize(moves.size());
124.    reverse_copy(moves.begin(), moves.end(), ret.begin());
125.    return ret;
126.}
127.int main()
128.{
129.    string sourceFile;
130.    int startX, startY, cheeseX, cheeseY;
131.    cout << "Labyrinth-Quelldatei: "; cin >> sourceFile;
132.    cout << "Startkoordinaten:\n\tx: "; cin >> startX; cout << "\ty: "; cin >>
    startY;
133.    cout << "Käsekoordinaten:\n\tx: "; cin >> cheeseX; cout << "\ty: "; cin >>
    cheeseY;
134.
135.    Labyrinth lab(sourceFile.c_str());
136.    vector<Move> moves = lab.simulateMouse(startX, startY, cheeseX, cheeseY);
137.
138.    cout << "Die Maus verhält sich wie folgt:\n";
139.    for (unsigned int c = 0; c < moves.size(); ++c) {
140.        string o = "Ein Schritt nach ";
141.        switch (moves[c].mov) {
142.            case 0: cout << (o += "Norden\n"); break;
143.            case 1: cout << (o += "Süden\n"); break;
144.            case 2: cout << (o += "Westen\n"); break;
145.            case 3: cout << (o += "Osten\n"); break;
146.            case 4: cout << "Die Maus frisst\n"; break;
147.            case 5: cout << "Die Startstelle der Maus war [" <<

```

```

    moves[c].argX << ' ' << moves[c].argY << "]\n";
148.     }
149. }
150.
151. return EXIT_SUCCESS;
152.}

```

3. Aufgabe 3

3.1. Lösungsidee

Die Lösung dieser Aufgabe gliedert sich in drei Schritte. Der erste Schritt ist es, die Sätze zu analysieren und in eine geeignete Form zu bringen. Die Analyse der Sätze ist einfach: Der Satz wird eingelesen, alle Großbuchstaben in Kleinbuchstaben verwandelt, Sonderzeichen sowie Umlaute werden gelöscht. Dann werden zwei Objektbezeichner und ein Relationsbezeichner aus dem Satz herausgesucht. Weitere Bezeichner werden ignoriert, sollten zu wenige Bezeichner im Text sein wird der ganze Satz ignoriert. Nun werden die eingelesenen Objektbezeichner als Objekte erzeugt und in eine Namens-Object-Pointer-Hash-Map und eine Array von Pointern auf alle Objekte eingetragen, das alles natürlich nur für den Fall, das die genannten Objekt nicht schon vorher genannt wurden. Die Objekte haben für jede Bekannten Relationsbezeichner ein Array von Pointern auf Objekte, mit denen sie diese Beziehung innehaben. Diesem Array wird jetzt die gefundene Beziehung bei den Objekten zugefügt. Am Ende werden bei jeder Beziehung in jedem Objekt jeweils die Inverse Beziehung in das verknüpfte Objekt eingetragen, also von z. B. „Fahrrad#1 rechts von Fahrrad#2“ wird erst in Fahrrad#1 in das Array aller „rechts von“-Beziehungen ein Pointer auf Fahrrad#2 eingefügt und nun, am Ende, auch in Fahrrad#2 ein Pointer auf Fahrrad#1 in der Kategorie „links von“ eingetragen. Nun hat man Eine Art „Netz von Objekten“, die jeweils von Pointern verbunden sind. Damit kann man schon Feststellen, wie viele vorgegebene Relationen in einem Bestimmten Raum nicht erfüllt sind, d. h. wie gut ein Raum die Vorgaben erfüllt. Ein Raum sei hier definiert als ein dreidimensionales Array von Pointern auf Objekte, bei dem ein leeres Raumsegment = 0 ist.

Der zweite Schritt ist es, durch eine rekursive Funktion den Raum herauszufinden, der die Vorgaben am besten erfüllt. Bei Räumen ohne Widersprüche geht das einfach, sind aber Widersprüche vorhanden, so kann das etwas Rechenintensiver werden, da das Programm sich von unten nach oben Hocharbeitet, also zuerst davon ausgeht, dass Null Widersprüche da sind, dann davon, dass einer da ist usw. Auf meinem Rechner dauerte das Erstellen des Bildes des in der Aufgabe gegebenen Raums etwa 3 Sekunden. Das liegt auch daran, das man wirklich alle Möglichkeiten ausprobieren muss, schließlich Gibt es Konstrukte wie:

Objekt#1 rechts von Objekt#2

Objekt#3 links von Objekt#1

Objekt#3 rechts von Objekt#2

Bei denen man zwischen Objekt#1 und Objekt#2 einen Leerraum lassen muss, damit Objekt#3 die Bedingungen erfüllt. Um die Rechenzeit möglichst gering zu halten platziert mein Programm Objekte immer nur so, dass sie möglichst viele gegebene Bedingungen erfüllen (alle gleichguten Möglichkeiten werden durchprobiert) und ruft sich dann selbst auf, um das nächste Objekt zu platzieren. Das sollte auch keinen Unterschied im Ergebnis machen. Das Ergebnis ist ein Array von Pointern auf Objekte, das bei mir die Größe 6*5*5 hat. Das ist das Maximum, das bei mir auf ein 400*300-Bitmap passt. Das Bild wird nun Schicht für Schicht, von hinten nach vorne wegen der Überlagerung, durch Überlagerung von 50*50-Bitmaps der einzelnen Objekte aufgebaut. Die Formel der Position der linken oberen Ecke des Bildes des Objekts im Gesamtbild lautet dabei:

$[x * 50 - 4 * 20 + z * 20, y * 50 - 4 * 10 + z * 10]$

Das ist zwar nicht wirklich dreidimensional gezeichnet, wirkt aber in etwa so.

Die Definition der Relationen ist bei mir im engsten Sinne gefasst: Links von, Rechts von, Unter, Über, Vor und Hinter sind bei mir nur, wenn das Objekt sich z. B. Genau über dem anderen befindet, also nicht nach rechts, links, vorne oder hinten verschoben ist. Neben bedeutet bei mir direkt angrenzend, aber nicht über oder unter dem anderen Objekt, sondern nur auf derselben Höhe. Mehrdeutigkeiten und Unvollständigkeiten werden rein zufällig entschieden, bei Widersprüchen wird versucht, möglichst viele Sätze einzuhalten.

Aufgrund meiner Auslegung der Aufgabe (Künstlerische Freiheit!?) ist die Lösung sehr rechenintensiv für den Fall dass es Widersprüche gibt. Da die Aufgabe aber explizit sagt, das man die Vorgaben frei auslegen darf hoffe ich, das das Programm nicht „zu langsam“ ist.

3.2.Programm–Dokumentation

Der Typ Object ist definiert als ein String name (der Name), ein Pixmap pix (die Bildrepresentation) und eine Hash-Map, in der Relations zu Arrays von Object-Pointern zugeordnet werden.

Der Typ Relation ist definiert als ein String Name und eine Relative Position, z. B. (0, 0, -1) (eine Ausnahme bildet hier „neben“, welches die relative Position 0, 0, 0 hat, was rein willkürlich ist). Alle Relationen werden im Array relations[] definiert.

Der Typ Complex ist im Grunde die Beschreibung des Raumes als 3-dimensionales Array von Object-Pointern (room[6][5][5]) und einer Hash-Map, in der Object-Pointern Positionen zugewiesen werden, an denen sie im Komplex zu finden sind (position) sowie einer Integer Variable namens bad, die die Anzahl der nicht erfüllten Bedingungen beinhalten soll (natürlich erst nach Zusammensetzen des Raumes).

Zuerst werden in der Funktion process() die Sätze eingelesen und in einem Array von Object-Pointer gespeichert, die in der Hauptfunktion „allObjects“ heißt. Die Objekte wurden vorher mit der Funktion loadObjects() aus einem Verzeichnis von XPM-Bilddateien geladen (Name der Datei ohne Endung = Objektdatei). XPM steht für Unix Pixmap. Die Grafik-Funktionen habe ich selbst geschrieben, sie implementieren aber keineswegs den ganzen XPM-Standard, genauso wie die Ausgabe nicht alle Kompressionsmethoden des BMP-Formates nutzt. Der Vorgang in process() ist trivial.

Beim Einlesen der Sätze werden Umlaute zur 'x' ersetzt und Sonderzeichen weggelassen.

Nun werden alle Elemente des eben genannten Arrays allObjects mit der Funktion resort() wie in der Lösungsidee beschrieben neu verkettet, sodass auch die durch eine bestimmte Relation mit dem jeweiligen Objekt verknüpften Objekte jeweils die Inverse verknüpfung mit dem jeweiligen Objekt haben. Auch dieser Vorgang ist trivial.

Nun wird in einer while-Schleife so lange die unten beschriebene Funktion recConstruct() aufgerufen, bis sie eine Lösung gefunden hat. Dabei wird die maximale Anzahl von Fehlern (hier maxRec) immer um eins erhöht.

Die Funktion recConstruct() nimmt als Argumente einen Complex (in main() ein leerer Raum, Name: c), ein Array von Pointern auf Objekte (in main() noch alle Objekte, Name: objects), ein Array aller Lösungen (in main() noch leer) und die Anzahl der bisher unerfüllten Bedingungen (Name: bad) sowie die maximale Anzahl unerfüllter Bedingungen (Name: maxBad). Falls bad > maxBad ist kehrt recConstruct sofort zur aufrufenden Funktion zurück. Zuerst sucht sich die Funktion den Index des ersten Elements in objects, das nicht gleich 0 ist (d. h. schon positioniert). Der Index wird in der Variable i gespeichert. Sollte es kein solches Objekt geben ist eine Lösung gefunden, die in das Lösungsarray eingetragen wird worauf die Funktion zu main() zurückkehrt (per throw). Sollte keine Lösung gefunden worden sein so werden alle Positionen in c gesucht, an

denen objects[i] positioniert werden und dabei möglichst wenige Widersprüche verursacht. Für alle diese Möglichkeiten wird dann recConstruct mit entsprechenden Argumenten rekursiv aufgerufen.

Am Ende wird das Bild vom Lösungsraum erzeugt (Formel siehe Lösungsidee) und in eine Datei geschrieben.

3.3.Programm–Ablaufprotokoll

Als erstes Beispiel die in der Aufgabe vorgegebene Satzfolge:

```
INITIALISIERE...
Lade Objekte aus Verzeichnis ["default" wird ersetzt durchx "bilder/"]: default
Lade Objekte...
    12 Dateien gefunden.
    Objekt "zimmerpflanze" geladen.
    Objekt "sessel" geladen.
    Objekt "teddybxxr" geladen.
    Objekt "sitzkissen" geladen.
    Objekt "haustier" geladen.
    Objekt "fahrrad" geladen.
    Objekt "lagerfeuer" geladen.
    Objekt "couch" geladen.
    Objekt "stehlampe" geladen.
    Objekt "kunstobjekt" geladen.
10 Objekte geladen.
FERTIG.
ekai> Ich hätte gerne das Fahrrad#1 links von Stehlampe#1.
ekai> Der Sessel#1 steht links von Sessel#2.
ekai> Couch#1 befindet sich vor Zimmerpflanze#2.
ekai> Sessel#1 ist neben dem Lagerfeuer#1.
ekai> Ein Sessel#2 steht hinter Lagerfeuer#1.
ekai> Sitzkissen#1 unter Haustier#1.
ekai> Kunstobjekt#1 schwebt über Lagerfeuer#1.
ekai> Die Zimmerpflanze#1 hinter Teddybär#1.
ekai> Fahrrad#1 ist irgendwo rechts von Sessel#2.
ekai> Stehlampe#1 sieht man links von Sessel#1.
ekai> Doch die Zimmerpflanze#2 steht hinter dem Fahrrad#1.
ekai> ;
```

Schreiben nach: bsp1.bmp

Geschrieben!

Die Bedienung ist selbsterklärend. äbsp1.bmp sieht jetzt so aus:



Bedingung 1 wurde hier wegen eines Widerspruches ausgelassen. Das merkwürdige bunte Objekt über dem Lagerfeuer ist ein Kunstobjekt.

Ein weiteres Beispiel:

```
INITIALISIERE...
Lade Objekte aus Verzeichnis ["default" wird ersetzt durchx "bilder/"]: default
Lade Objekte...
    12 Dateien gefunden.
    Objekt "zimmerpflanze" geladen.
```

```

Objekt "sessel" geladen.
Objekt "teddybxxr" geladen.
Objekt "sitzkissen" geladen.
Objekt "haustier" geladen.
Objekt "fahrrad" geladen.
Objekt "lagerfeuer" geladen.
Objekt "couch" geladen.
Objekt "stehlampe" geladen.
Objekt "kunstobjekt" geladen.
10 Objekte geladen.
FERTIG.
ekai> sitzkissen#1 neben lagerfeuer#1
ekai> sitzkissen#2 neben lagerfeuer#1
ekai> sitzkissen#3 neben lagerfeuer#1
ekai> sessel#1 neben lagerfeuer#1
ekai> sessel#2 neben lagerfeuer#1
ekai> sessel#3 neben lagerfeuer#1
ekai> couch#1 neben lagerfeuer#1
ekai> couch#2 neben lagerfeuer#1
ekai> kunstobjekt#1 über lagerfeuer#1
ekai> ;

```

Schreiben nach: bsp2.bmp

Geschrieben!

Das Ergebnis ist:



Hier erkennt man sehr gut, das neben auch „links hinter“ heißen kann, sodass beim letzten Beispiel Sessel#1 sehr wohl neben Lagerfeuer#1 steht, was nicht unbedingt offensichtlich ist.

Und hier noch das dritte Beispiel mit etwas komplizierteren Beziehungen:

```

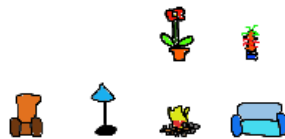
INITIALISIERE...
Lade Objekte aus Verzeichnis ["default" wird ersetzt durchx "bilder/"]: default
Lade Objekte...
12 Dateien gefunden.
Objekt "zimmerpflanze" geladen.
Objekt "sessel" geladen.
Objekt "teddybxxr" geladen.
Objekt "sitzkissen" geladen.
Objekt "haustier" geladen.
Objekt "fahrrad" geladen.
Objekt "lagerfeuer" geladen.
Objekt "couch" geladen.
Objekt "stehlampe" geladen.
Objekt "kunstobjekt" geladen.
10 Objekte geladen.
FERTIG.
ekai> Es soll ein Sessel#1 links von Couch#1.
ekai> Und wie wär's mit 'nem Lagerfeuer#1 rechts von Sessel#1?
ekai> Oh, und natürlich soll Couch#1 rechts von Lagerfeuer#1 sein!
ekai> Und nun noch ein Kunstobjekt#1 über Couch#1.
ekai> Ne Stehlampe#1 rechts von Sessel#1 käm auch ganz gut...
ekai> Und natürlich nicht vergessen: Zimmerpflanze#1 neben Kunstobjekt#1.
ekai> Und zu guter letzt sei noch gesagt dass Zimmerpflanze#1 über Lagerfeuer#1
soll.

```

ekai> ;

Schreiben nach: bsp3.bmp
Geschrieben!

äbsp3.bmp sieht jetzt so aus:



Wie man sieht sind hier alle Bedingungen erfüllt.

3.4. Programm-Text

Das eigentliche Programm (ekai.cpp):

```

1. /**
2.  * Lösung zu:
3.  * BWINF 20, Runde 1, Aufgabe 3
4.  **
5.  * Von Florian Eisele
6.  */
7.
8. #include "pixmap.h"
9.
10. #include <fstream>
11. #include <sstream>
12. #include <string>
13. #include <list>
14. #include <algorithm>
15. #include <vector>
16. #include <map>
17.
18. #include <cstdlib>
19. #include <cctype>
20.
21. using namespace std;
22.
23. #include <dirent.h> // POSIX Standard, weiß nicht, ob das unter Windoze
    funzt...
24. #include <unistd.h>
25.
26. struct Object;
27. struct Complex;
28.
29. struct Relation {
30.     Relation(int x, int y, int z, const char* n)
31.     {
32.         relPos[0] = x; relPos[1] = y; relPos[2] = z;
33.         name = n;
34.     }
35.     bool operator ==(const Relation & r)
36.     {
37.         return ((relPos[0] == r.relPos[0]) && (relPos[1] ==
r.relPos[1]) && (relPos[2] == r.relPos[2]));
38.     }
39.     Relation operator +(const Relation & a)

```



```

40.         {
41.             return Relation(a.relPos[0] + relPos[0], a.relPos[1] +
relPos[1], a.relPos[2] + relPos[2], "");
42.         }
43.     int relPos[3];
44.     string name;
45. };
46.
47. const int numberOfRelations = 7;
48.
49. Relation relations[] = {
50.     Relation(-1, 0, 0, "links von"),
51.     Relation( 1, 0, 0, "rechts von"),
52.     Relation( 0, 0, 0, "neben"), // undefiniert
53.     Relation( 0, 0, 1, "vor"),
54.     Relation( 0, 0,-1, "hinten"),
55.     Relation( 0,-1, 0, "xber"),
56.     Relation( 0, 1, 0, "unter")
57. };
58.
59. Relation* getInverse(Relation* rel) // Suche Inverses in relations[]
60. {
61.     Relation r(-rel->relPos[0], -rel->relPos[1], -rel->relPos[2], "");
62.     for (int c = 0; c < numberOfRelations; ++c)
63.         if (relations[c] == r)
64.             return &relations[c];
65.     return 0;
66. }
67.
68. struct Object {
69.     Object(const string& n, const Pixmap& p) { name = n; pix = p; }
70.     Object() : name("nothing"), pix(50, 50, 0xffffffff)
71.     {
72.     }
72.     Object(const Object & o)
73.     {
74.         name = o.name;
75.         pix = o.pix;
76.     }
77.     string name;
78.     Pixmap pix;
79.     map<Relation*, vector<Object*> > next;
80. };
81.
82. vector<Object> loadObjects(string loadPath)
83. {
84.     if (loadPath[loadPath.size() - 1] != '/')
85.         loadPath += '/';
86.     char* tmp = new char[PATH_MAX];
87.     if (loadPath[0] != '/')
88.         loadPath = string(getcwd(tmp, PATH_MAX)) + '/' + loadPath;
89.     delete tmp;
90.
91.     list<string> fileList;
92.     list<Object> objList;
93.     vector<Object> objects;
94.     // Der folgende Teil benutzt nicht-standard-Funktionen (POSIX):
95.     cout << "Lade Objekte...\n";
96.     DIR* loadDir = opendir(loadPath.c_str());
97.     dirent* entry;
98.     while ((entry = readdir(loadDir)) != 0)
99.         fileList.push_front(string(entry->d_name));
100.    closedir(loadDir);
101.    // Ende non-standard
102.    cout << '\t' << fileList.size() << " Dateien gefunden.\n";
103.    while (fileList.size()) {
104.        if (fileList.front().size() > 4)
105.            if (fileList.front().substr(fileList.front().size() - 4, 4) ==
".xpm") {
106.                bool success = true;
107.                ifstream in(string(loadPath +
fileList.front().c_str()));
108.                string objName = fileList.front().substr(0,

```

```

    fileList.front().length() - 4);
109.         if (in) {
110.             try {
111.                 objList.push_front(Object(objName,
    Pixmap(in)));
112.             } catch (string s) {
113.                 cerr << "\tFehler beim Laden von Objekt \""
    << objName << "\":\n\t#->" << s << '\n';
114.                 success = false;
115.             }
116.             if (success)
117.                 cout << "\tObjekt \"" << objName << "\"
    geladen.\n";
118.             in.close();
119.         }
120.     }
121.     fileList.pop_front();
122. }
123. objects.resize(objList.size());
124. reverse_copy(objList.begin(), objList.end(), objects.begin());
125. cout << objects.size() << " Objekte geladen.\n";
126. return objects;
127.}
128.
129.void process(vector<Object> loadedObjects, map<string, Object*> & nameMap,
    vector<Object*> & objects, const char * sentence)
130.{
131.    vector<Object*> oldObjects = objects;
132.    map<string, Object*> oldNameMap = nameMap;
133.    istringstream from(sentence, strlen(sentence));
134.    Object* obj[2] = { 0, 0 };
135.    Relation* rel = 0;
136.    string prev = "";
137.    while (from) {
138.        string s;
139.        from >> s;
140.        for (unsigned int x = 0; x < s.length(); ++x) {
141.            if ((s[x] == 'ä') || (s[x] == 'ö') || (s[x] == 'ü'))
142.                s[x] = 'x';
143.            s[x] = (isalnum(s[x]) || (s[x] == '#'))
144.                ? tolower(s[x]) : '\000';
145.        }
146.        if (s.length() > 0)
147.            if (s[s.length() - 1] == '\000')
148.                s.resize(s.length() - 1);
149.        int c = 0;
150.        while (c < numberOfRelations) {
151.            if ((s == relations[c].name) || (prev + ' ' + s ==
    relations[c].name)) {
152.                if (rel != 0)
153.                    cerr << "Weiterer Relationidentifizier.
    Ignoriert.\n";
154.                else
155.                    rel = &(relations[c]);
156.            }
157.            ++c;
158.        }
159.        Object** thisObject = &obj[0];
160.        if (*thisObject)
161.            thisObject = &obj[1];
162.        if (*thisObject)
163.            thisObject = 0;
164.        if (nameMap[s]) {
165.            if (thisObject) {
166.                *thisObject = nameMap[s];
167.            } else
168.                cerr << "Ein weiterer Objektidentifizier wurde gefunden.
    Ignoriert.\n";
169.        } else {
170.            for (unsigned int q = 0; q < loadedObjects.size(); ++q) {
171.                if (loadedObjects[q].name + '#' == s.substr(0,
    loadedObjects[q].name.size() + 1)) {

```

```

172.             if (thisObject) {
173.                 *thisObject = new Object(loadedObjects[q]);
174.                 (*thisObject)->name = s;
175.                 objects.resize(objects.size() + 1);
176.                 nameMap[s] = *thisObject;
177.                 objects[objects.size() - 1] = *thisObject;
178.             } else
179.                 cerr << "Ein weiterer Objektidentifizier wurde
gefunden. Ignoriert.\n";
180.             }
181.         }
182.     }
183.     prev = s;
184. }
185.
186. if ((rel == 0) || (obj[0] == 0) || (obj[1] == 0)) {
187.     if (obj[0]) delete obj[0];
188.     if (obj[1]) delete obj[1];
189.     objects = oldObjects;
190.     nameMap = oldNameMap;
191.     cerr << "Dieser Satz ist nicht vollständig. Ignoriert.\n";
192.     return;
193. }
194.
195. vector<Object*>* obj0next = &obj[0]->next[rel];
196. for (unsigned int c = 0; c < obj0next->size(); ++c)
197.     if ((*obj0next)[c] == obj[1])
198.         return;
199. obj0next->resize(obj0next->size() + 1);
200. (*obj0next)[obj0next->size() - 1] = obj[1];
201.}
202.
203.void resort(Object* obj)
204.{
205.    for (int c = 0; c < numberOfRelations; ++c)
206.        for (unsigned int d = 0; d < obj->next[relations[c]].size(); ++d) {
207.            Object* obj2 = obj->next[relations[c]][d];
208.            obj2->next[getInverse(&relations[c])].resize(obj2->next[getInv
erse(&relations[c])].size() + 1);
209.            obj2->next[getInverse(&relations[c])][obj2->next[getInverse(&r
elations[c])].size() - 1] = obj;
210.        }
211.}
212.
213.struct Pos {
214.    Pos () { x = -1; y = -1; z = -1; }
215.    Pos (int _x, int _y, int _z) : x(_x), y(_y), z(_z) { }
216.    bool operator ==(const Pos& p) const
217.    { return ((x == p.x) && (y == p.y) && (z == p.z)); }
218.    int x, y, z;
219.};
220.
221.struct Complex {
222.    Object* room[6][5][5];
223.    map<Object*, Pos> position;
224.    Complex operator =(const Complex& b)
225.    {
226.        for (int x = 0; x < 6; ++x)
227.            for (int y = 0; y < 5; ++y)
228.                for (int z = 0; z < 5; ++z)
229.                    room[x][y][z] = b.room[x][y][z];
230.        position = b.position;
231.        bad = b.bad;
232.        return *this;
233.    }
234.    void putObject(Pos p, Object* obj)
235.    {
236.        room[p.x][p.y][p.z] = obj;
237.        position[obj] = p;
238.    }
239.    void eraseObject(Pos p, Object* obj)
240.    {

```

```

241.         room[p.x][p.y][p.z] = 0;
242.         position[obj] = Pos();
243.     }
244.     int bad;
245. };
246.
247. bool isGood(Relation rel, Pos o1, Pos o2)
248. {
249.     if (rel == Relation(0, 0, 0, "")) {
250.         if (((o1.x - o2.x)*(o1.x - o2.x) <= 1) && ((o1.z - o2.z)*(o1.z -
251.             o2.z) <= 1)) && ((o1.y - o2.y)*(o1.y - o2.y) == 0))
252.             return true;
253.         return false;
254.     }
255.     int xDiff = o1.x - o2.x;
256.     int yDiff = o1.y - o2.y;
257.     int zDiff = o1.z - o2.z;
258.     if ((rel.relPos[0] == 0) && (xDiff != 0))
259.         return false;
260.     if ((rel.relPos[1] == 0) && (yDiff != 0))
261.         return false;
262.     if ((rel.relPos[2] == 0) && (zDiff != 0))
263.         return false;
264.     if ((rel.relPos[0] == -1) && (xDiff > 0))
265.         return false;
266.     if ((rel.relPos[1] == -1) && (yDiff > 0))
267.         return false;
268.     if ((rel.relPos[2] == -1) && (zDiff > 0))
269.         return false;
270.     if ((rel.relPos[0] == 1) && (xDiff < 0))
271.         return false;
272.     if ((rel.relPos[1] == 1) && (yDiff < 0))
273.         return false;
274.     if ((rel.relPos[2] == 1) && (zDiff < 0))
275.         return false;
276.     return true;
277. }
278. int unfulfilled(Object* obj, Pos p, Complex com)
279. {
280.     com.putObject(p, obj);
281.     int retVal = 0;
282.     for (int c = 0; c < numberOfRelations; ++c) {
283.         map<Object*, bool> yetHad;
284.         for (unsigned int d = 0; d < obj->next[&relations[c]].size(); ++d)
285.             if ((!yetHad[obj->next[&relations[c]][d]]) &&
286.                 (com.position[obj->next[&relations[c]][d]] != Pos())) {
287.                 yetHad[obj->next[&relations[c]][d]] = true;
288.                 if (!isGood(relations[c], com.position[obj],
289.                     com.position[obj->next[&relations[c]][d]]))
290.                     retVal++;
291.             }
292.     }
293.     return retVal;
294. }
295. void recConstruct(Complex& c, vector<Object*> & objects, vector<Complex> &
296.     sols, int bad, int& maxBad)
297. {
298.     if (bad > maxBad)
299.         return;
300.     unsigned int i = 0;
301.     while ((!objects[i]) && (i < objects.size()))
302.         ++i;
303.     if (i == objects.size()) {
304.         sols.resize(sols.size() + 1);
305.         maxBad = bad;
306.         c.bad = bad;
307.         sols[sols.size() - 1] = c;
308.         cout << '.';
309.         throw string();
310.     }

```

```

309. int lowestBad = 1000000000; // Nur ein großer Wert, keine weitere
    Bedeutung...
310. list<Pos> tryOut;
311. for (int x = 0; x < 6; ++x)
312.     for (int y = 0; y < 5; ++y)
313.         for (int z = 0; z < 5; ++z) {
314.             if (c.room[x][y][z] == 0) {
315.                 if (unfulfilled(objects[i], Pos(x, y, z), c) <
lowestBad) {
316.                     tryOut = list<Pos>();
317.                     tryOut.push_front(Pos(x, y, z));
318.                     lowestBad = unfulfilled(objects[i], Pos(x,
y, z), c);
319.                 } else if (unfulfilled(objects[i], Pos(x, y, z),
c) == lowestBad) {
320.                     tryOut.push_front(Pos(x, y, z));
321.                 }
322.             }
323.         }
324. while (tryOut.size()) {
325.     c.putObject(tryOut.front(), objects[i]);
326.     Object* oi = objects[i];
327.     objects[i] = 0;
328.     recConstruct(c, objects, sols, bad + lowestBad, maxBad);
329.     objects[i] = oi;
330.     c.eraseObject(tryOut.front(), objects[i]);
331.     tryOut.pop_front();
332. }
333.}
334.
335.int main() {
336.    cout << "INITIALISIERE...\n";
337.    string loadPath;
338.    cout << "Lade Objekte aus Verzeichnis [\"default\" wird ersetzt durchx
\"bilder/\"]: ";
339.    cin >> loadPath;
340.    if (loadPath == "default")
341.        loadPath = "bilder/";
342.    vector<Object> objects = loadObjects(loadPath);
343.    cout << "FERTIG.\n";
344.    map<string, Object*> nameMap; // Zur beschleunigung des Auffindens eines
    bestimmten Objekts;
345.    vector<Object*> allObjects;
346.    char cmdString[1024];
347.    cin.get(); // Da is noch ein \n im Stream...
348.    do {
349.        cout << "ekai> ";
350.        cin.getline(cmdString, 1024);
351.        if (strcmp(cmdString, ";"))
352.            process(objects, nameMap, allObjects, cmdString);
353.    } while (strcmp(cmdString, ";"));
354.    for (unsigned int c = 0; c < allObjects.size(); ++c)
355.        resort(allObjects[c]);
356.    Complex complex;
357.    for (int x = 0; x < 6; ++x)
358.        for (int y = 0; y < 5; ++y)
359.            for (int z = 0; z < 5; ++z)
360.                complex.room[x][y][z] = 0;
361.    vector<Complex> sols;
362.    int bad = 0;
363.    int maxBad = 0;
364.    while (!sols.size()) {
365.        try {
366.            recConstruct(complex, allObjects, sols, bad, maxBad);
367.        } catch (string s) { }
368.        ++maxBad;
369.    }
370.    cout << '\n';
371.    complex = sols[sols.size() - 1];
372.
373.    Pixmap pix(300, 400, 0xffffffff);
374.    for (int z = 0; z < 5; ++z)

```

```

375.         for (int x = 0; x < 6; ++x)
376.             for (int y = 0; y < 5; ++y)
377.                 if (complex.room[x][y][z] != 0) {
378.                     pix.layOver(complex.room[x][y][z]->pix, y * 50 - 4
* 10 + z * 10, x * 50 - 4 * 20 + z * 20, 0xffffffff);
379.                 }
380.
381.     cout << "Schreiben nach: ";
382.     string fd;
383.     cin >> fd;
384.     pix.write(fd.c_str());
385.     cout << "Geschrieben!\n";
386.     return EXIT_SUCCESS;
387. }

```

Die Grafikunterstützung (selbstgeschrieben, daher muss das hier hin):

pixmap.h:

```

1. /**
2.  * Lösung zu:
3.  * BWINF 20, Runde 1, Aufgabe 3
4.  **
5.  * Von Florian Eisele
6.  **/
7.
8. /** Achtung:
9.  * Diese Mini-Bibliothek implementiert nur einen kleinen Teil der XPM-
10.  * Spezifikation. Sie funktioniert mit den Bildern dieser Aufgabe und mit
11.  * allen (soweit ich das beurteilen kann) aus GIMP exportierten XPMs.
12.  **
13.  * Das Ausgabeformat ist aufgrund der Aufgabenstellung BMP.
14.  */
15.
16. #ifndef PIXMAP_H
17. #define PIXMAP_H
18.
19. #include <vector>
20. #include <string>
21. #include <iostream>
22.
23. using namespace std;
24.
25. typedef unsigned int color;
26.
27. class Pixmap {
28. public:
29.     Pixmap(); // Erzeugt Dummy-Pixmap;
30.     Pixmap(int h, int w, color c); // Erzeuge ein w*h großes Pixmap mit
        Hintergrundfarbe c;
31.     Pixmap(istream& in); // Lade Pixmap aus Datei fd;
32.     Pixmap(const Pixmap& p);
33.     void write(const char* fileName); // Schreibe als Bitmap auf to;
34.     void layOver(Pixmap & pix, int topY, int leftX, color transparent); //
        "Lege" pix über das Bild
35. private:
36.     void createData(int x, int y);
37.     vector<color> usedCols; // Alle im Bild verwendeten Farben;
38.     int width, height;
39.     vector< vector<color> > data; // Referenzierung: [x][y], [0][0] = links-
        oben;
40. };
41.
42. #endif /*!(PIXMAP_H)*/

```

Und pixmap.cpp, die Implementation der Grafik:

```

1. /**
2.  * Lösung zu:
3.  * BWINF 20, Runde 1, Aufgabe 3
4.  **
5.  * Von Florian Eisele
6.  **/
7.

```

```

8. #include "pixmap.h"
9.
10. #include <fstream>
11. #include <iomanip>
12. #include <map>
13. #include <algorithm>
14.
15. Pixmap::Pixmap()
16. {
17.     width = 0; height = 0;
18. }
19.
20. Pixmap::Pixmap(int h, int w, color c)
21. {
22.     width = w; height = h;
23.     createData(width, height);
24.     usedCols.resize(1);
25.     usedCols[1] = c;
26.     for (int x = 0; x < w; ++x)
27.         for (int y = 0; y < h; ++y)
28.             data[x][y] = c;
29. }
30.
31. Pixmap::Pixmap(const Pixmap& p)
32. {
33.     width = p.width; height = p.height;
34.     usedCols.resize(p.usedCols.size());
35.     copy(p.usedCols.begin(), p.usedCols.end(), usedCols.begin());
36.     createData(width, height);
37.     for (int x = 0; x < width; ++x)
38.         for (int y = 0; y < height; ++y)
39.             data[x][y] = p.data[x][y];
40. }
41.
42. inline void streamDump(istream& in, char end) // Dumpe Stream bis
    einschließlich dem ersten Vorkommen von end
43. {
44.     char ch;
45.     do { in.get(ch); } while ((ch != end) && (in));
46. }
47.
48. Pixmap::Pixmap(istream& in)
49. {
50.     string currStr = "";
51.     char curr;
52.     int colors, cpp; // Breite, Höhe, Anzahl Farben, Chars pro Pixel
53.     int stage = 0; // 0 = Head, 1 = Colors, 2 = Data, 3 = End
54.     int counter = 0;
55.     map<string, color> colorTable;
56.     map<string, bool> defined; // Alle definierten Farben werden auf true
    gesetzt
57.     while (in && (stage != 3)) {
58.         in.get(curr);
59.         if (curr == '\\')
60.             switch (stage) {
61.                 case 0: {
62.                     in >> height >> width >> colors >> cpp;
63.                     if (colors == 0)
64.                         throw string("Too few colors! Error in
    pixmap.h#Pixmap::Pixmap(string);");
65.                     if (cpp == 0)
66.                         throw string("Too few chars per pixel! Error in
    pixmap.h#Pixmap::Pixmap(string);");
67.                     while (in && (curr != '\\'))
68.                         in.get(curr);
69.                     createData(width, height);
70.                     usedCols.resize(colors);
71.                     streamDump(in, '\\');
72.                     ++stage; }
73.                 break;
74.                 case 1: {
75.                     string colorRep;

```

```

76.         colorRep.resize(cpp);
77.         for (int x = 0; x < cpp; ++x)
78.             in.get(colorRep[x]);
79.         do {
80.             in.get(curr);
81.         } while ((curr == ' ') || (curr == '\n') || (curr == '\t'));
82.         if (curr != 'c')
83.             throw string(string("\"cRep ") + curr + string(" ...\" in
the Color-Part is not implemented. Error in
pixmap.h#Pixmap::Pixmap(string);"));
84.         do {
85.             in.get(curr);
86.         } while (curr == ' ' || (curr == '\n') || (curr == '\t'));
87.         color col;
88.         if (curr != '#') { // Alle nicht-implementierten Farben werden
als Weiß aufgefasst
89.             col = 0xFFFFFFFF;
90.             streamDump(in, '\n');
91.         } else {
92.             in >> hex >> col;
93.             streamDump(in, '\n');
94.         }
95.         defined[colorRep] = true;
96.         usedCols[counter] = colorTable[colorRep] = col;
97.         ++counter;
98.         if (counter == colors) {
99.             ++stage;
100.            counter = 0;
101.        } }
102.        break;
103.        case 2: {
104.            char thisPixel[cpp + 1];
105.            thisPixel[cpp] = 0;
106.            for (int x = 0; x < width; ++x) {
107.                for (int y = 0; y < cpp; ++y)
108.                    in.get(thisPixel[y]);
109.                data[x][counter] = colorTable[thisPixel];
110.                if (!defined[thisPixel])
111.                    throw string("Pixmap definition contains undefined
colors. Error in pixmap.h#Pixmap::Pixmap(string);");
112.            }
113.            streamDump(in, '\n');
114.            ++counter;
115.            if (counter == height)
116.                ++stage; }
117.        }
118.    }
119.    if (stage != 3)
120.        throw string("The specified file is not XPM-compilant. Error in
pixmap.h#Pixmap::Pixmap(string);");
121.    // Optimierte Farbtabelle:
122.    map<color, bool> yetUsed;
123.    int newSize = 0;
124.    int pos = 0;
125.    for (int x = 0; x < width; ++x)
126.        for (int y = 0; y < height; ++y)
127.            if (!yetUsed[data[x][y]]) {
128.                usedCols[pos] = data[x][y];
129.                yetUsed[data[x][y]] = true;
130.                pos++; newSize++;
131.            }
132.    usedCols.resize(newSize);
133.}
134.
135.const string replacers =
136.    "
. + @ # $ % & * = - ; > , ' ) ! ~ { ] ^ / ( _ : < [ ] | 1 2 3 4 5 6 7 8 9 0 a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z ' " ;
137.
138.void Pixmap::write(const char* fileName)
139.    // Long muss hierfür 32-Bit groß sein. Läuft also nur auf 32-Bit-Rechnern.
140.{

```



```

141. ofstream to(fileName, ios::out|ios::binary);
142. if (!to)
143.     throw string("Could not write file!");
144. unsigned long size = 54 + 4 * width * height;
145. unsigned long w = width, h = height, unused = 0;
146. unsigned long offset = 54;
147. // Fileheader:
148. to.write("BM", 2 * sizeof(char));
149. to.write((const char*)&size, sizeof(size));
150. to.write((const char*)&unused, sizeof(unused));
151. to.write((const char*)&offset, sizeof(offset));
152.
153. // Infoheader:
154. size = 40;
155. to.write((const char*)&size, sizeof(size));
156. to.write((const char*)&w, sizeof(w));
157. to.write((const char*)&h, sizeof(h));
158. to << char(1);
159. to << '\000'; // Bits Per Pixel
160. to << char(32);
161. to << '\000'; // Planes
162. unused = 0;
163. to.write((const char*)&unused, sizeof(unused)); // Type of Compression
164. unused = width * height * 4;
165. to.write((const char*)&unused, sizeof(unused)); // Compressed Image size
    (wird nicht benutzt)
166. unused = 5000;
167. to.write((const char*)&unused, sizeof(unused)); // X-PPM
168. to.write((const char*)&unused, sizeof(unused)); // Y-PPM
169. unused = usedCols.size();
170. to.write((const char*)&unused, sizeof(unused)); // Anzahl der
    gebrauchten Farben
171. unused = 0;
172. to.write((const char*)&unused, sizeof(unused)); // Anzahl der wichtigen
    Farben (0 = alle)
173.
174. // Raster Data:
175. for (int c = height - 1; c >= 0; c--)
176.     for (int d = 0; d < width; ++d) {
177.         to.write((const char*)&data[d][c], sizeof(color));
178.     }
179. }
180.
181. void Pixmap::createData(int x, int y)
182. {
183.     data.resize(x);
184.     for (int c = 0; c < x; ++c)
185.         data[c].resize(y);
186. }
187.
188. void Pixmap::layOver(Pixmap & pix, int topY, int leftX, color transparent)
189. {
190.     if ((topY - pix.height < 0) || (leftX + pix.width >= width) || (topY < 0)
        || (leftX < 0))
191.         throw string("Doesn't fit!#Pixmap::layOver(Pixmap&,int,
            int,color);");
192.     for (int x = 0; x < pix.width; ++x)
193.         for (int y = 0; y < pix.height; ++y)
194.             if (pix.data[x][y] != transparent)
195.                 data[x + leftX][topY + y] = pix.data[x][y];
196. }

```

4. Aufgabe 4

4.1.Lösungsidee

Ein absolutes Brute-Force, bei dem alle Möglichkeiten durchprobiert werden ist bei dieser Aufgabe zu langsam. Es gilt daher Einschränkungen zu machen:

1. Es bringt nichts Teile eines Zuges umgekehrt an einen Anderen anhängen zu wollen. Das ist

damit zu begründen dass durch das Abkoppeln von Zug 1, Umdrehen, Ankoppeln an Zug 2 und Abkoppeln von eines Teils Zug 2 nichts gegenüber dem Abkoppeln von zwei Stücken von Zug 1 und dem Ankoppeln eines an Zug 2 nichts an Kopplungen gespart wurde.

2. Die Reihenfolge der Koppelvorgänge ist egal. Die kürzeste Variante ist immer erst beide Züge in n verschiedene Teile zu Teilen und diese dann wieder zusammenzukoppeln. Daher wird immer eine gerade Anzahl von Koppelvorgängen gebraucht.
3. Es bringt nichts ein Stück aus einem ankommenden Zug herauszukoppeln, das ein Teil eines der gebrauchten Züge ist wenn sich mit der gleichen Anzahl Kopplungen ein größeres Stück herauskoppeln lässt, das auch ein Teil des gebrauchten Zuges ist.

Der Algorithmus kann also rekursiv die beiden Züge in Teile zerlegen und diese jeweils entweder an den Zug nach K oder V anhängen. Das Anhängen an die Züge nach K und V darf aber nur dann passieren wenn Regel 3 erfüllt ist, also wenn man das Stück, das man gerade von einem der Züge abgekoppelt hat, nicht noch größer hätte machen können ohne dass es nicht mehr in den für K bzw. V gebrauchten Zug passen würde. Die beiden ankommenden Züge werden so einfach der Reihe nach in Teile zerlegt und wenn diese Regel 3 erfüllen an den ausgehenden Zug nach K bzw. V gehängt und dann wird rekursiv weitergemacht. Am Ende wird die kürzeste gefundene Lösung ausgegeben.

4.2. Programm–Dokumentation

Der Algorithmus wird in der Funktion `search()` implementiert, die als Argumente die gebrauchten Züge (`o1` und `o2`) vom Typ `OutTrain` (s. u.), `frag`, ein Array aus den beiden eingehenden Zügen, `sols`, das Array aller Lösungen in „Wortlaut“, `solStr`, eine Liste von Anweisungen die bei Erfolg eines rekursiven Zweiges der Funktion an `sols` angehängt wird, `train1` und `train2`, die Strings der derzeitigen „ausgehenden Züge“, die am Ende der rekursion immer den Spezifikationen für Konsumweiler und Verberau entsprechen, am Anfang aber leer sind.

Die Klasse `OutTrain`, von der `o1` und `o2` sind ist eine Hilfsklasse, die Beschreibungsstrings für ausgehende Züge einliest, die Anzahl der Vorkommen der einzelnen Waggontypen zählt und später mit der Funktion `isOk(string)` testet, ob der übergebene String den Spezifikationen für den entsprechenden ausgehenden Zug genügt und mit der Funktion `isPart(string)` Testet, ob der übergebene String ein Teil des verlangten Zuges ist, also von keinem Waggontyp mehr als die erwartete Anzahl enthält.

Am Anfang liest `main()` die Züge und den Bedarf ein und ruft `search()` mit entsprechenden Argumenten auf. `search()` sucht sich zuerst einen der beiden eingehenden Züge aus (existiert der erste so wird der erste genommen, falls der erste leer, d. h. alle Waggons wurden abgekoppelt, ist wird der zweite genommen) und setzt den Pointer `currFrag` gleich der Adresse des jeweiligen Strings. Für den Fall das nicht beide Züge schon leer sind (alle Waggons abgekoppelt) folgt nun der rekursive Teil der Funktion. Ansonsten ist auf jeden Fall ein funktionierender Kuppelvorgang durchgeführt worden und die Lösung in `solStr` wird in `sols` (die Lösungsmenge) geschrieben. Der rekursive Teil der Funktion besteht daraus dass in einer `for`–Schleife mit zwei Durchläufen, in denen jeweils an `train1` oder `train2` das größtmögliche Stück von Anfang bis zu einem bestimmten Punkt aus dem Zug, auf den `currFrag` zeigt, angehängt wird und entsprechend bei der Variable auf die `currFrag` zeigt, entfernt wird. Der Vorgang wird in `solStr` geloggt (Wenn ein Zug nach dem Abkoppeln keinen Waggon mehr hat oder der Zug, an den Angekoppelt wird noch leer ist wird das natürlich nicht geloggt, da kein Kuppelvorgang erforderlich ist) und `search()` ruft sich rekursiv auf. Danach werden `solStr` und alle Zugstring wieder in ihren Ursprungszustand zurückversetzt und entweder läuft die `for`–Schleife noch einmal durch (derselbe Vorgang wird auf den anderen ausgehenden Zug angewand, s. o.) oder die Funktion kehrt zurück.

In `main()` wird nun die kürzeste Lösung in `sols` herausgesucht und ausgegeben.

4.3. Programm–Ablaufprotokoll

Der Tagesbedarf, angegeben als Folge von Waggonen, die Reihenfolge ist daher egal:

Tagesbedarf in Konsumweiler:

AAAAACCCCCDDDDDDDEEE

Tagesbedarf in Verberau:

AAAAAAABBBBBBCCCCCCCCCEEE

die Ankommenden Züge:

Zug aus Produdorf:

LAAAAAAAABBBBBBCCCCCDDDD

Zug aus Herställen:

LDDDDDEEEEEEECCCCCCCCCAAAA

..

Und hier einer der kürzesten Koppelvorgänge (es gibt vielleicht noch andere gleichlange):

Entkopple LAAAAAAAABBBBBBCCCCCDDDD zu LAAAAAAA und ABBBBBBBCCCCCDDDD.

Entkopple ABBBBBBBCCCCCDDDD zu A und BBBBBBBBCCCCCDDDD.

Entkopple BBBBBBBBCCCCCDDDD zu BBBBBBBBCCCCC und DDD.

Kopple LAAAAAAA und BBBBBBBBCCCCC zu LAAAAAAAABBBBBBCCCCC.

Kopple A und DDD zu ADDD.

Entkopple LDDDDDEEEEEEECCCCCCCCCAAAA zu LDDDDDEEE und EEEEECCCCCCCCCAAAA.

Kopple ADDD und LDDDDDEEE zu ADDDLDDDDDEEE.

Entkopple EEEEECCCCCCCCCAAAA zu EEEEC und CCCCCCAAAA.

Kopple LAAAAAAAABBBBBBCCCCC und EEEEC zu LAAAAAAAABBBBBBCCCCCCCCCEEEEC.

Kopple ADDDLDDDDDEEE und CCCCCCAAAA zu ADDDLDDDDDEEECCCCCCCCCAAAA.

ADDDLDDDDDEEECCCCCCCCCAAAA fährt nach Konsumweiler.

LAAAAAAAABBBBBBCCCCCCCCCEEEEC fährt nach Verberau.

Hier noch ein weiteres Beispiel:

Tagesbedarf in Konsumweiler:

AAAAAAAABBBBBBBBCCCCDD

Tagesbedarf in Verberau:

CCCCAAAAAABBE

Zug aus Produdorf:

LAAAAAAAABBBBBBCCDDE

Zug aus Herställen:

LEEAaaaaaaABBBBBCCCCC

..

Entkopple LAAAAAAAABBBBBBCCDDE zu LAAAAAAAABBBBBBCCDD und E.

Entkopple LEEAaaaaaaABBBBBCCCCC zu LEEAaaaaaa und ABBBBBCCCCC.

Kopple E und LEEAaaaaaa zu ELEEAAAAA.

Entkopple ABBBBBCCCCC zu AB BB und BBCCCCC.

Kopple LAAAAAAAABBBBBBCCDD und AB BB zu LAAAAAAAABBBBBBCCDDABBB.

Entkopple BBCCCCC zu BBCCCC und CC.

Kopple ELEEAAAAA und BBCCCC zu ELEEAAAAAABCCCC.

Kopple LAAAAAAAABBBBBBCCDDABBB und CC zu LAAAAAAAABBBBBBCCDDABBBCC.

LAAAAAAAABBBBBBCCDDABBBCC fährt nach Konsumweiler.

ELEEAAAAAABCCCC fährt nach Verberau.

Und noch das dritte Beispiel:

Tagesbedarf in Konsumweiler:

AAABBCDDDEEFFGGGHHI

Tagesbedarf in Verberau:

CCCBBAFFFEEDIIHHG

Zug aus Produdorf:

LAAABBCCEEEFFGGGHII

Zug aus Herställen:

LABBCDDDEFFGHHHII

..

Entkopple LAAABBCCEEEFFGGGHII zu LAAABBC und CCDEEEFFGGGHII.

Entkopple CCDEEEFFGGGHII zu CCDEE und EFFGGGHII.

Entkopple EFFGGGHII zu EF und FGGGHII.

Kopple LAAABBC und EF zu LAAABCE.

Entkopple FG GGHII zu FG und GGHII.
 Kopple CCDEE und FG zu CCDEEFG.
 Entkopple GGHII zu GGHI und I.
 Kopple LAAABBCEF und GGHI zu LAAABBCEFGGHI.
 Kopple CCDEEFG und I zu CCDEEFGI.
 Entkopple LABBCDDDEFFGHHHII zu LABBC und DDDEFFGHHHII.
 Kopple CCDEEFGI und LABBC zu CCDEEFGILABBC.
 Entkopple DDDEFFGHHHII zu DDDE und FFGHHHII.
 Kopple LAAABBCEFGGHI und DDDE zu LAAABBCEFGGHIDDDE.
 Entkopple FFGHHHII zu FF und GHHHII.
 Kopple CCDEEFGILABBC und FF zu CCDEEFGILABBCFF.
 Entkopple GHHHII zu GH und HHII.
 Kopple LAAABBCEFGGHIDDDE und GH zu LAAABBCEFGGHIDDDEGH.
 Kopple CCDEEFGILABBCFF und HHII zu CCDEEFGILABBCFFHHII.
 LAAABBCEFGGHIDDDEGH fährt nach Konsumweiler.
 CCDEEFGILABBCFFHHII fährt nach Verberau.

4.4. Programm-Text

```

1. /**
2.  * Lösung zu:
3.  * BWINF 20, Runde 1, Aufgabe 4
4.  **
5.  * Von Florian Eisele
6.  **/
7.
8. #include <iostream>
9. #include <string>
10. #include <vector>
11. #include <list>
12. #include <map>
13. #include <algorithm>
14. #include <cstdlib>
15.
16. using namespace std;
17.
18. class OutTrain {
19. public:
20.     OutTrain(string s);
21.     bool isOk(string s); // Testet, ob s den Spezifikationen genügt;
22.     bool isPart(string s); // Testet, ob s ein Teil ist;
23.     map<char, int> countOfOccurance;
24.     vector<char> occuringTypes;
25. };
26.
27. OutTrain::OutTrain(string s)
28. {
29.     occuringTypes.resize(s.length() + 1);
30.     if (s.find('L') == string::npos)
31.         s += 'L';
32.     unsigned int pos = 0;
33.     for (unsigned int c = 0; c < s.length(); ++c) {
34.         countOfOccurance[s[c]]++;
35.         if (countOfOccurance[s[c]] == 1) {
36.             occuringTypes[pos++] = s[c];
37.         }
38.     }
39.     occuringTypes.resize(pos);
40. }
41.
42. bool OutTrain::isPart(string s)
43. {
44.     OutTrain t(s);
45.     for (unsigned int x = 0; x < t.occuringTypes.size(); ++x)
46.         if (t.countOfOccurance[t.occuringTypes[x]] >
47.             countOfOccurance[t.occuringTypes[x]])
48.             return false;
49.     return true;
50. }
51.
52. bool OutTrain::isOk(string s)
53. {

```

```

53.   if (s.find('L') == string::npos)
54.       return false;
55.   OutTrain t(s);
56.   unsigned int i = 0;
57.   for (unsigned int c = 0; c < occuringTypes.size(); ++c)
58.       for (unsigned int d = 0; d < t.occuringTypes.size(); ++d)
59.           if (occuringTypes[c] == t.occuringTypes[d]) {
60.               ++i;
61.               if (countOfOccurance[occuringTypes[c]] !=
t.countOfOccurance[t.occuringTypes[d]])
62.                   return false;
63.           }
64.   if ((i != occuringTypes.size()) || (i != t.occuringTypes.size()))
65.       return false;
66.   return true;
67.}
68.
69.void search(OutTrain & o1, OutTrain & o2, vector<string> frag,
vector<list<string> > & sols,
70.   list<string> solStr, string train1, string train2)
71.{
72.   string* currFrag = 0;
73.   if (frag[0] != "") {
74.       currFrag = &frag[0];
75.   }
76.   else if (frag[1] != "") {
77.       currFrag = &frag[1];
78.   }
79.   if (currFrag) {
80.       for (int c = 0; c < 2 ; ++c) {
81.           OutTrain* oP = 0;
82.           string* trainP = 0;
83.           switch (c) {
84.               case 0:
85.                   oP = &o1;
86.                   trainP = &train1;
87.                   break;
88.               case 1:
89.                   oP = &o2;
90.                   trainP = &train2;
91.           }
92.           unsigned int x = 0;
93.           string tmpT = *trainP, tmpF = *currFrag;
94.           while ((oP->isPart(*trainP + currFrag->substr(0, x))) && (x <=
currFrag->length()))
95.               x++;
96.           x--;
97.           *trainP += currFrag->substr(0, x);
98.           *currFrag = currFrag->substr(x, currFrag->length() - x);
99.           if (*currFrag != "")
100.               solStr.push_front("Entkopple " + tmpF + " zu " +
tmpF.substr(0, x) + " und " +
101.                   *currFrag + '.');
102.           if (tmpT != "")
103.               solStr.push_front("Kopple " + tmpT + " und " +
tmpF.substr(0, x) + " zu " + *trainP + '.');
104.           if (x != 0)
105.               search(o1, o2, frag, sols, solStr, train1, train2);
106.           if (*currFrag != "")
107.               solStr.pop_front();
108.           if (tmpT != "")
109.               solStr.pop_front();
110.           *trainP = tmpT;
111.           *currFrag = tmpF;
112.       }
113.       return;
114.   }
115.   cout << '.'; cout.flush();
116.   solStr.push_front(train1 + " fährt nach Konsumweiler.");
117.   solStr.push_front(train2 + " fährt nach Verberau.");
118.   sols.resize(sols.size() + 1);
119.   sols[sols.size() - 1] = solStr;

```

```

120.}
121.
122.int main()
123.{
124.    string needed[2], incoming[2]; // Bedarf und eintreffende Züge als Folge
    von Wagontypen;
125.    cout << "Tagesbedarf in Konsumweiler:\n"; cin >> needed[0];
126.    cout << "Tagesbedarf in Verberau:      \n"; cin >> needed[1];
127.    cout << "Zug aus Produdorf:           \n"; cin >> incoming[0];
128.    cout << "Zug aus Herställen:          \n"; cin >> incoming[1];
129.    cout <<
    "_____\n";
130.    OutTrain o1(needed[0]), o2(needed[1]);
131.    vector<string> frags(2);
132.    frags[0] = incoming[0]; frags[1] = incoming[1];
133.    vector<list<string> > sols;
134.    list<string> solStr;
135.    search(o1, o2, frags, sols, solStr, "", "");
136.    cout << '\n';
137.
138.    if (!sols.size()) {
139.        cerr << "Es wurde kein passender Koppelvorgang gefunden.\n";
140.        return EXIT_FAILURE;
141.    }
142.
143.    unsigned int i = 0;
144.    for (unsigned int c = 0; c < sols.size(); ++c)
145.        if (sols[c].size() < sols[i].size())
146.            i = c;
147.
148.    reverse(sols[i].begin(), sols[i].end());
149.    while (sols[i].size()) {
150.        cout << sols[i].front() << '\n';
151.        sols[i].pop_front();
152.    }
153.
154.    return EXIT_SUCCESS;
155.}

```

5. Aufgabe 5

5.1.Lösungsidee

Endlich mal eine Aufgabe deren Lösung nicht auf Brute-Force beruht. Also, am Anfang hat man einen Winkel, hier alpha genannt, und die Daten hier x_0 , y_0 und r genannt) des Kreises (Dose, im Folgenden beschränke ich mich auf die Form). Gesucht ist/sind der/die Punkt(e) x und y (Endkoordinaten der verschobenen Dose), an dem/denen es bei Verschiebung zur Kollision mit einem zweiten Kreis kommt mit den Koordinaten x_2 , y_2 , und r_2 . Sei $\alpha \neq 90^\circ$ und $\alpha \neq 270^\circ$, dann gilt:

$y = \tan(\alpha) \cdot x + y_0 - \tan(\alpha) \cdot x_0$ (Geradengleichung der Verschiebungsbahn)

und $(x - x_2)^2 + (y - y_2)^2 = (r + r_2)^2$ (Die Entfernung vom Endpunk der Verschiebung und der zweiten Dose muss die Summe der Radien sein).

Die Herleitung ist trivial. Wenn dieses Gleichungssystem genau zwei Lösungen hat, so trifft Dose 1 Dose 2. Bei Einer Lösung geht sie gerade vorbei, streift die zweite Dose aber gerade.

Sollte die Geradengleichung eine Parallele zur y -Achse sein, so gilt $x = x_0$ sowie die zweit Gleichung von oben.

Ersetzt man $\tan(\alpha)$ durch t und $y_0 - t \cdot x_0 - y_2$ durch q und $(r + r_2)$ durch r_{gesamt} so ergibt sich für Gleichungssystem 1:

$$x = \frac{x_2 - t * q}{t^2} \pm \sqrt{\left(\frac{x_2 - t * q}{t^2}\right)^2 - \frac{x_2^2 + q^2 - r_{gesamt}^2}{t^2 + 1}}$$

Wenn der Term unter der Wurzel größer als 0 ist treffen sich die Dosen an einem der Ergebnisse. Immer das Ergebnis, das am kürzesten vom Ausgangsort entfernt ist ist das richtige. Das gilt auch für die Lösung des Gleichungssystems 2 und der Gesamtlösung. y kann man mit der ersten Gleichung ausrechnen.

Gleichungssystem zwei hat die Lösung:

$$y = y_2 \pm \sqrt{r_{gesamt}^2 - (x - x_2)^2}$$

x ist ja wie gesagt hier konstant. Auch hier gilt wenn die beiden Dosen sich treffen hat das Gleichungssystem zwei Lösungen von denen die mit dem kleineren Abstand zum Ausgangspunkt die richtige ist.

Zum Finden der Gesamtlösung setzt man einfach eine Lösungsposition (x,y) gleich dem Punkt, an dem die bewegte Dose eine Wand trifft (mit Gleichung 1 leicht zu berechnen). Dann lässt man für jede schon existierende Dose die Punkte an denen sie Dose eins trifft berechnen. Ist die Entfernung dieser Punkte bezüglich dem Ausgangspunkt kleiner als die der bisherigen Lösung, so wird diese durch die neuen Koordinaten ersetzt. Dabei ist zu berücksichtigen, dass das Gleichungssystem nicht die Bewegungsrichtung der Dose einschließt, also noch geprüft werden muss, ob sich Dose 2 überhaupt in der Bewegungsrichtung von Dose 1 befindet.

5.1.1. Teilaufgabe 2

Die offensichtlichste und vermutlich auch beste Möglichkeit verschiebe zu verwenden um eine möglichst gute Packung zu erhalten beruht auf der Idee, für alle möglichen ganzzahligen (oder auch alle in irgend einem anderen Abstand, z.B. alle Vielfachen von 0.5) Winkel (im Programm, nicht real) durchzuführen und eine PackungsdichtevARIABLE zu berechnen, was sich so bewerkstelligen lässt: Summe der Flächen aller Dosen durch die Fläche des kleinsten Rechtecks, in das sich die Dosen in der jetzigen Konstellation packen lassen würden. Der Winkel, der am Ende die größte PackungsdichtevARIABLE ergibt wird dann real als Verschiebungswinkel für die neue Dose genommen. Da die Daten der später hinzukommenden Dosen am Anfang nicht bekannt sind dürfte dies (wenn man entsprechend viele Winkel durchprobiert) eine ganz gute Approximation an die Optimallösung sein. Ein Nachteil dieser Lösungsidee ist die relativ hohe Rechenzeit (da verschiebe() aber nicht rekursiv ist dürfte die Rechenzeit immer noch deutlich unter der von Anderen Aufgaben liegen).

Eine andere Möglichkeit, die die PackungsdichtevARIABLE nicht berechnen muss, aber auch wesentlich schlechter als die eben genannte Lösung ist, ist es, die Dosen in etwa in Dreiecksform zu schieben, also eine Dose immer in einer bestimmten Ecke positionieren und dann zu verschieben, wobei die erste Dose im Winkel 45° verschoben wird und allgemein jede n-te Dose so verschoben wird, dass wenn n-1 eine Dreieckszahl ist (Von der Form (i²-i)/2) soll der Winkel gleich 45° / (Breite der Box / maximaler Durchmesser einer Dose) * i + 45° gesetzt werden, ansonsten soll der vorherige um (90° minus 2 * 45° / (Breite der Box / maximaler Durchmesser einer Dose) * i) * (n - 1) verkleinert werden, wobei i immer das i vom letzten n ist, bei dem n-1 eine Dreieckszahl war. Wenn ich mich nicht verrechnet habe sollte so ca. die Hälfte der Box gefüllt werden (in Dreieckiger Packung, was so weit ich weiß die effizienteste ist). Danach muss man die n-te-Dose zu Winkel-Relation umdrehen, also am Anfang möglichst viele Dosen in eine Reihe und am Schluss nur noch eine in eine Reihe stecken. Wohlgemerkt stimmen die Formeln nur dann, wenn ich mich nicht vertan habe, sollten aber auf jeden Fall nach geringer Anpassung stimmen.

5.2. Programm-Dokumentation

Die Box wird durch den Typ Box repräsentiert, der die Maße (xWidth und yWidth) sowie ein Array aller beinhalteten Dosen beinhaltet. Eine Dose wird durch den Typ Can repräsentiert, der die Position (x, y) und den Radius r beinhaltet.

Am Anfang werden alle schon vorhandenen Dosen eingelesen und in die Box eingeführt. Dann wird die zu Positionierende Dose eingelesen und eingefügt. Dann wird der Winkel alpha eingelesen und die Funktion verschiebe() mit entsprechenden Argumenten (Box: b, Index der zu verschiebenden Dose: i, Winkel alpha) aufgerufen und die Position der zu verschiebenden Dose wird erneut ausgegeben.

Die Funktion verschiebe() macht im Wesentlichen das, was in der Lösungsidee beschrieben ist. Zuerst werden die Variablen x0, y0 und ri (in der Lösungsidee x_0 , y_0 und r) initialisiert. Das Argument alpha, das den Verschiebungswinkel beschreibt wird in Bogenmaß ($/180^\circ \cdot \pi$) umgerechnet und der ursprüngliche Wert wird in oldAlpha gespeichert.

Nun findet die Fallunterscheidung $\text{oldAlpha} = 90^\circ$ bzw. $\text{oldAlpha} = 270^\circ$ oder oldAlpha gleich irgendwas anderes statt.

Für den Fall dass oldAlpha nicht 90 oder 270 Grad ist (was wohl in der Mehrzahl der Fälle der Fall ist) werden zuerst minX, minY, maxX und maxY, die Koordinaten, zwischen denen sich die Dose maximal bewegen kann, berechnet. Das ist im Wesentlichen die Startposition der Dose in minX|minY und die Position in der sie an die Wand der Box trifft. Die Endergebnisvariablen x und y werden auch gleich diesem Treffpunkt mit der Wand gesetzt. Danach werden in einer for-Schleife für alle Elemente aus der Box, in der sich die Dose befindet, ohne die Dose selbst nach den Formeln aus der Lösungsidee Kollisions- (=Schnitt-) Punkte berechnet und in localX[2] und localY[2] gespeichert. Dann wird für beide Koordinatenpaare (localX[0]|localY[0]) und (localX[1]|localY[1]) getestet, ob sie zwischen (minX|minY) und (maxX|maxY) liegen und ob die Entfernung zwischen Ausgangspunkt und dem derzeitigen Ergebnis (x|y) größer ist als die zwischen Ausgangspunkt und (localX[*]|localY[*]). Sollte das der Fall sein werden x und y durch die jeweiligen localX und localY Werte ersetzt.

Für den Fall dass oldAlpha = 90° oder 180° passiert essentiell das Selbe, nur das hier x immer gleich x0 ist und y am Anfang ri (Fall 270°) oder die y-Breite der Box – ri ist (Fall 90°) gesetzt wird. Außerdem werden die Formeln ersetzt und es müssen immer nur y-Werte berechnet werden da x unveränderlich ist.

5.3. Programm-Ablaufprotokoll

Beispiel 1:

```
x-Breite der Kiste in Metern: 1
y-Breite der Kiste in Metern: 1
Wieviele Dosen sind schon in der Kiste?:1
0-te Dose:
x-Position in Metern: 0.75
y-Position in Metern: 0.75
Radius in Metern: 0.25
---
Neue Dose:
x in Metern: 0.05
y in Metern: 0.05
r in Metern: 0.05
Winkel in °: 45
---
Die Dose wurde verschoben nach: (0.537868|0.537868).
```

Hier wird also in einer 1*1-Meter großen Box mit einer 0.25 Meter großen Dose an der Position (0,75|0,75) angefangen und es wird eine 0,05 Meter große Dose an der Position (0,05|0,05)

eingefügt und im Winkel 45° verschoben. Sie trifft die schon vorhandene Dose und bleibt an der Position (0,537868|0,537868) stehen.

Ein weiteres Beispiel:

```
x-Breite der Kiste in Metern: 1
y-Breite der Kiste in Metern: 1
Wieviele Dosen sind schon in der Kiste?:2
0-te Dose:
x-Position in Metern: 0.25
y-Position in Metern: 0.75
Radius in Metern: 0.25
---
1-te Dose:
x-Position in Metern: 0.75
y-Position in Metern: 0.25
Radius in Metern: 0.25
---
Neue Dose:
x in Metern: 0.05
y in Metern: 0.05
r in Metern: 0.05
Winkel in °: 45
---
Die Dose wurde verschoben nach: (0.95|0.95).
Hier wurde keine Dose getroffen.
```

Und Beispiel Nr. 3:

```
x-Breite der Kiste in Metern: 1
y-Breite der Kiste in Metern: 1
Wieviele Dosen sind schon in der Kiste?:1
0-te Dose:
x-Position in Metern: 0.5
y-Position in Metern: 0.5
Radius in Metern: 0.25
---
Neue Dose:
x in Metern: 0.5
y in Metern: 0.05
r in Metern: 0.005
Winkel in °: 89
---
Die Dose wurde verschoben nach: (0.503404|0.245023).
```

5.4.Programm-Text

```
1. /**
2.  * Lösung zu:
3.  * BWINF20, Runde 1, Aufgabe 5
4.  **
5.  * Von Florian Eisele
6.  **/
7.
8. #include <cmath>
9. #include <iostream>
10. #include <vector>
11. #include <cstdlib>
12.
13. struct Can { // Die Dose
14.     Can(double _x, double _y, double _r) : x(_x), y(_y), r(_r) { }
15.     Can(const Can & c)
16.         { x = c.x; y = c.y; r = c.r; }
17.     Can() : x(0), y(0), r(0) { }
18.     double x, y, r;
19. };
20.
21. struct Box { // Und die Box
22.     Box(double _xWidth, double _yWidth) : xWidth(_xWidth), yWidth(_yWidth) { }
23.     Box(const Box & b)
24.         {
```

```

25.             xWidth = b.xWidth; yWidth = b.yWidth;
26.             cans = b.cans;
27.         }
28.         double xWidth, yWidth;
29.         vector<Can> cans;
30. };
31.
32. void verschiebe(Box & b, unsigned int i, double alpha)
33. {
34.     double x = 0, y = 0;
35.     double oldAlpha = alpha;
36.     double x0 = b.cans[i].x, y0 = b.cans[i].y, ri = b.cans[i].r;
37.     alpha = alpha / 180 * 3.141592;
38.
39.     if ((oldAlpha != 90) && (oldAlpha != 270)) {
40.         // Berechne Zussamenprallpunkt mit Kistenwand:
41.         double t = tan(alpha);
42.         double minX = ri, minY = t * minX + y0 - t * x0, maxX = b.xWidth -
ri, maxY = t * maxX + y0 - t * x0;
43.         if (minY < ri) {
44.             minY = ri;
45.             minX = (minY + t * x0 - y0) / t;
46.         }
47.         if (maxY > b.yWidth - ri) {
48.             maxY = b.yWidth - ri;
49.             maxX = (maxY + t * x0 - y0) / t;
50.         }
51.         if ((270 > oldAlpha) && (oldAlpha > 90)) {
52.             maxX = x0; x = minX;
53.             maxY = y0; y = minY;
54.         } else {
55.             minX = x0; x = maxX;
56.             minY = y0; y = maxY;
57.         }
58.
59.         for (unsigned int c = 0; c < b.cans.size(); ++c)
60.             if (c != i) {
61.                 double x2 = b.cans[c].x, y2 = b.cans[c].y, rges =
b.cans[c].r + ri;
62.                 double q = y0 - t * x0 - y2;
63.                 double valOne = (x2 - t * q) / (t * t + 1);
64.                 double underSqrt = valOne * valOne - (x2 * x2 + q * q -
rges * rges) / (t * t + 1);
65.                 if (underSqrt > 0) { // Ansonsten geht er vorbei, bei
underSqrt = 0 berührt er, geht aber trotzdem weiter;
66.                     double localX[2], localY[2];
67.                     localX[0] = valOne + sqrt(underSqrt);
68.                     localX[1] = valOne - sqrt(underSqrt);
69.                     localY[0] = t * localX[0] + q + y2;
70.                     localY[1] = t * localX[1] + q + y2;
71.                     for (int w = 0; w < 2; ++w)
72.                         if (((localX[w] - x0) * (localX[w] - x0) +
(localY[w] - y0) * (localY[w] - y0) <
73.                             (x - x0) * (x - x0) + (y - y0) *
(y - y0))
74.                             if (((minX <= localX[w]) &&
(localX[w] <= maxX)) || ((minX >= localX[w]) && (localX[w] >= maxX)) &&
75.                                 (((minY <= localY[w]) &&
(localY[w] <= maxY) || (minY >= localY[w]) && (localY[w] >= maxY)))) {
76.                                     // Liegt auf dem Weg
77.                                     x = localX[w]; y = localY[w];
78.                                 }
79.                             }
80.                     }
81.             } else {
82.                 x = x0, y = 0;
83.                 switch ((int)(oldAlpha)) {
84.                     case 90: y = b.yWidth - ri; break;
85.                     case 270: y = ri; break;
86.                 }
87.                 for (unsigned int c = 0; c < b.cans.size(); ++c)
88.                     if (c != i) {

```

```

89.         double x2 = b.cans[c].x, y2 = b.cans[c].y, rges =
        b.cans[c].r + ri;
90.         double possY[2];
91.         if (rges * rges - (x - x2) * (x - x2) > 0) {
92.             possY[0] = y2 + sqrt(rges * rges - (x - x2) * (x -
            x2));
93.             possY[1] = y2 - sqrt(rges * rges - (x - x2) * (x -
            x2));
94.             for (int w = 0; w < 2; ++w)
95.                 if ((possY[w] - y0) * (possY[w] - y0) < (y -
            y0) * (y - y0))
96.                     if (((y0 <= possY[w]) && (possY[w] <=
            y)) || ((y0 >= possY[w]) && (possY[w] >= y))) {
97.                         // Liegt auf dem Weg
98.                         y = possY[w];
99.                     }
100.        }
101.    }
102. }
103. b.cans[i].x = x;
104. b.cans[i].y = y;
105. }
106.
107. int main()
108. {
109.     double xW, yW;
110.     cout << "x-Breite der Kiste in Metern: "; cin >> xW;
111.     cout << "y-Breite der Kiste in Metern: "; cin >> yW;
112.     Box b(xW, yW);
113.
114.     int c;
115.     cout << "Wieviele Dosen sind schon in der Kiste?: "; cin >> c;
116.     b.cans.resize(c + 1);
117.     for (int d = 0; d < c; ++d) {
118.         double x, y, r;
119.         cout << d << "-te Dose:\n";
120.         cout << "x-Position in Metern: "; cin >> x;
121.         cout << "y-Position in Metern: "; cin >> y;
122.         cout << "Radius in Metern: "; cin >> r;
123.         b.cans[d] = Can(x, y, r);
124.         cout << "---\n";
125.     }
126.
127.     cout << "Neue Dose:\n";
128.     double x, y, r, alpha;
129.     cout << "x in Metern: "; cin >> x;
130.     cout << "y in Metern: "; cin >> y;
131.     cout << "r in Metern: "; cin >> r;
132.     cout << "Winkel in °: "; cin >> alpha;
133.     b.cans[b.cans.size() - 1] = Can(x, y, r);
134.     verschiebe(b, b.cans.size() - 1, alpha);
135.     cout << "---\nDie Dose wurde verschoben nach: (" << b.cans[b.cans.size() -
        1].x << ' ' << b.cans[b.cans.size() - 1].y << ").\n";
136.
137.     return EXIT_SUCCESS;
138. }
139.

```

6. Zur CD

Alle Programme wurden unter SuSE Linux 7.1 (gcc 2.95.2, glibc 2.2.0) kompiliert und sollten auf entsprechenden Systemen lauffähig sein. Die bei Aufgabe 3 geforderten Bitmaps und Satzfolgen befinden sich unter /aufg3/*.bmp und /aufg3/satzfolge*.txt.