35. Bundeswettbewerb Informatik

Runde 1

01.09.2016 - 28.11.2016

Aufgabe 3

Rotation

23. November 2016

Eingereicht von: *Wouldn't IT be nice...* (Team-ID: 00007)

Tim Hollmann (6753)
ich@tim-hollmann.de

Anike Heikrodt (6841)
anikeheikrodt@online.de

Wir versichern hiermit, die vorliegende Arbeit ohne unerlaubte fremde Hilfe entsprechend der Wettbewerbsregeln des Bundeswettbewerb Informatik angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Tim Hollmann & *Anike Heikrodt*, den 23. November 2016

1 Abstraktion

Gegeben ist ein mechanisches Puzzle im Startzustand;

1.1 Puzzle-Zustand

Der Zustand eines mechanischen Puzzles ist eindeutig durch (die Anzahl und) Lage der Stäbe, den Rahmen (, die Position der Lücke) sowie die Richtung der Gravitationswirkung beschreibbar. Im Folgenden wird ein Puzzle analog zum Eingabeformat durch eine Matrix (mit Ursprung in der unteren linken Ecke) repräsentiert. Die Stelle (a, b) beschreibt die Art des Feldes in der a-ten Spalte und b-ten Reihe entweder als Rahmen-Feld, leeres Feld oder zugehörig zu einem Stab $[0 \dots 9]$[1]. Die Gravitation wirkt stets nach unten - in Richtung $(a, 0)$.
Das Eingabeformat beschreibt den initialen Zustand des Puzzles, das es durch eine Abfolge von Rotationen in einen Zielzustand zu versetzen gilt;

1.2 Zielzustand

Ein Zielzustand sei im folgenden ein Puzzle-Zustand, bei dem ein beliebiges Stäbchen aus dem Rahmen (bzw. durch die Lücke) gefallen ist.[2] Es existieren meist viele verschiedene Zielzustände für ein mechanisches Puzzle, die sich in der Anzahl der vorangegangenen Rotationen unterscheiden oder sogar teilweise in dieser Hinsicht equivalent sind. Ziel ist es nicht nur, einen beliebigen dieser Zustände zu erreichen (falls existent), sondern zudem einen (beliebigen) derjenigen, die mit der geringsten Anzahl an Rotationen aus dem Startzustand aus errechbar sind.

2 Lösungsidee

Die Lösung der Aufgabe bestand im Wesentlichen in zwei Teilen

1. Implementierung der Puzzle-Rotationen[3]; Simulation von Drehung eines Puzzles und Gravitationswirkung

2. Suchen einer kürzesten Abfolge von Rotationen hin zu einem Zielzustand

[1] Das Eingabeformat von `www.bundeswettbewerb-informatik.de` setzt bei der Verwendung einstelliger Ziffern zur Beschreibung der Stäbchen stillschweigend voraus, dass deren Anzahl nicht größer ist als 10 (0 bis 9). Diese Begrenzung wurde in die Implementierung übernommen; die folgende theoretische Lösung ist aber von der Stäbchenanzahl unabhängig.

[2] Erweiternd hätte man hier das betreffende Stäbchen spezifizieren können (Wie viele Drehungen sind nötig, um Stäbchen xy aus dem Rahmen zu befördern?).

[3] Im Folgenden sei die „Rotation eines Puzzles“ die aufeinander folgende Drehung *und* Anwendung von Gravitationssumulation auf ein Puzzle; bei einer schlichten „Drehung“ findet keine Gravitation statt.

2.1 Transitionen

Bei der Rotation eines Puzzles kommt es zunächst zur Drehung der Zustandsmatrix um 90° und dann zur Gravitationssimulation;

2.1.1 Drehung der Zustandsmatrix

Nach der Drehung einer quadratischen Matrix $N \times N$ um 90° im Uhrzeigersinn befindet sich jede originale Stelle (a, b) nun an der Stelle $(b, N - a - 1)$. Eine Drehung *gegen* den Uhrzeigersinn erfolgt hier (etwas ineffizient) durch drei Drehungen *im* Uhrzeigersinn.[4]. Die entsprechende Drehungsvorschrift wird auf jedes Feld der Matrix eines Zustandes angewandt, um diese in Summe in die entsprechende Richtung zu drehen.

2.1.2 Gravitation

Nach der Drehung der Zustandsmatrix kommt es zur Gravitationssimulation; Der Algorithmus zur (schrittweisen) Gravitationssimulation geht folgendermaßen vor:

1. Solange es Stäbe gibt, bei denen alle direkt darunter liegenden Felder leere Felder sind (der Stab also die Möglichkeit besitzt, zu fallen):
   a) nehme den untersten von diesen Stäben (denjenigen mit der kleinsten y-Koordinate) und
   b) senke alle Felder dieses Stabes gleichmäßig so lange herab (in y-Richtung gegen 0), bis nicht mehr alle direkt darunterliegenden Felder leere Felder sind, der Stab also auf einem anderen Stab oder dem Rahmen aufliegt.

In der Realität würden alle Stäbe gleichzeitig herunterfallen; da dies aber zu aufwängig zu implementieren wäre, ist es bei dieser schrittweisen Simulation der Gravitation wichtig, die Gravitation von unten an zu simulieren und immer denjenigen Stab mit Fallmöglichkeit zu betrachten, der die geringste y-Koordinate besitzt. (Man könnte zwar auch immer einen beliebigen Stab derer mit Fallmöglichkeit betrachten und diesen so weit wie möglich herabsenken, allerdings wäre dies nicht so effizient, weil dann Stäbe mehrfach betrachtet werden müssen, wenn zufällig darunterliegende Stäbe erst zeitlich danach herabgesenkt werden und sich somit neue Fallmöglichkeiten ergeben.)
Kann ein Stäbchen den Rahmen verlassen, ist dessen schrittweise Absenkung theoretisch endlos; wenn die y-Koordinate negativ wird, bricht der Algorithmus ab und gibt eine Erfolgsmeldung zurück.

2.2 Suche des Lösungsweges

Wir können nun die Rotation eines mechanischen Puzzles simulieren. Um zu ermitteln, ob es eine möglichkeit gibt, einen beliebigen Stab durch Rotationen durch die Lücke zu bekommen, und wenn ja, mit welcher Abfolge am schnellsten, ist es hilfreich, das mechanische Puzzle als ein endliches Transitionssystem zu interpretieren[5]. Die Rotation ei-

[4] Wobei dazwischen keine Anwendung von Gravitationssimulation stattfinden darf (Begriff „Drehung“).
[5] bzw. einen endlichen Automaten

nes mechanischen Puzzles stellt eine diskrete und deterministische Zustandsveränderung (Transition) dar; der vorherige Puzzle-Zustand wird in einen neuen überführt. Die Transitionen sind zudem gerichtet und der Zustandsraum endlich. Dieses Transitionssystem kann durch einen Graphen $G = (V, E)$ modelliert werden. In diesem Graphen entspricht jeder Knoten $\in V$ einem möglichen Zustand des gegebenen mechanischen Puzzles und jede Kante $\in E$ einer möglichen Transition zwischen zwei spezifischen Zuständen[6].[7]
Jeder Knoten in $v \in V$ hat folglich einen Ausgangsgrad von $d^+_G(v) = 2$; es sind stets Links- und Rechtsdrehung möglich. Gesucht ist nun der kürzeste Pfad $p = (S_0, \ldots, S_Z)$ vom Knoten des Startzustands S_0 hin zu einem Knoten S_Z, der die Zielbedingung erfüllt, dass ein beliebiger Stab aus dem Rahmen gefallen ist.
Hierzu wird eine Breitensuche verwendet. Der Algorithmus wird als Bekannt vorausgesetzt; dieser wird so implementiert, dass jeder Knoten nur ein mal besucht wird. Die Breitensuche überprüft ausgehend von Startknoten (dem initialen Zustand des Puzzles) zunächst dessen direkte Nachbarn auf Erfüllung des Zielzustands. Erfüllt keiner dieser Nachbarn die Zielbedingung, wird jeder der Nachbarn der Nachbarn überprüft und falls davon keiner die Zielbedingung erfüllt, deren Nachbarn und so weiter und so fort. Dadurch ist garantiert, dass stets der kürzeste Pfad zurückgegeben wird (im Gegensatz zur z.B. Tiefensuche). Ist kein Pfad von Startzustand zu einem Zielzustand vorhanden, führt die Breitensuche eine Suche über den kompletten Suchraum aus und gibt dann schließlich zurück, dass keine möglichkeit existiert, einen Zielzustand zu erreichen, sprich, jeder Stab egal bei welcher Ritationsabfolge im Rahmen bleibt.

2.2.1 Terminierung und Laufzeitkomplexität

Nimmt man die Anzahl der Zustände eines mechanischen Puzzles (zu Recht) als endlich an, ist die Menge der Knoten V im korrespondierenden Graphen ebenfalls endlich. Da der Breitensuchen-Algorithmus gemeinhin so implementiert wird, dass er jeden Knoten nur ein mal besucht, terminiert dieser bei einer endlichen Anzahl an Knoten stets. Die Laufzeitkomplexität der Breitensuche steigt bekanntlich im worst-case (kein Lösungspfad vorhanden → vollständige Suche) linear mit der größe des Zustandsraumes; $O(|V|+|E|)$. Es ist schwierig, einen direkten und allgemeinen Zusammenhang zwischen der Größe und Form eines mechanischen Puzzles und der Größe dessen Zustandsraumes $|V|$ herzustellen.

2.2.2 Mögliche Schwächen des Algorithmus

Die Schwäche dieses Algorithmus liegt darin, dass die größe des Zustandsraumes teilweise so groß werden kann, dass der zur Verfügung stehende Speicherplatz nicht mehr ausreicht oder der Algorithmus sehr lange Zeit benötigt. (Was aber ein generelles Problem von Algorithmen mit linearer Laufzeitkomplexität ist).

[6] → deterministische Transition
[7] Da nicht jede Transition instantan rückgängig gemacht werden kann, sind die Kanten zusätzlich gerichtet.

2.3 Korrektheitsargument

Unter der Annahme, dass die Transitionen zwischen den Puzzlezuständen durch Rotations- und Gravitationssimulation korrekt implementiert wurden, ist durch die „Anatomie“ des verwendeten Breitensuchen-Algorithmus garantiert, dass dieser 1. stets terminiert und 2. (im Gegensatz zur Tiefensuche) stets einen der kürzesten Lösungspfade zurückliefert oder mit Sicherheit feststellt, dass kein solcher existieren kann.

3 Umsetzung

Die Umsetzung erfolgte in C++ 11.

3.1 Puzzle-Repräsentation; Datentyp `puzzle`

Die Repräsentation eines Puzzle-Zustandes erfolgt programmintern als zweidimensionale Matrix;

```
22 typedef vector< vector< int > > puzzle;
```

Dabei beschreibt jeder integer-Wert eines Feldes dessen Typ; ein Wert -200 einen Rahmen, -100 ein leeres Feld sowie die Zahlen 0 bis 9 die Zugehörigkeit zu eben diesem Stab.

3.2 Funktionen und ihre Aufgaben

3.2.1 `puzzle turnPuzzle ( const puzzle& p, const bool clockwise = true )`

Helferfunktion; dreht ein Puzzle um 90° in die angegebene Richtung. Der Algorithmus entspricht dem in **2.1.1**.

3.2.2 `bool gforcePuzzle( puzzle& p )`

Herferfunktion; wendet Gravitationssimulation auf ein gegebenes Puzzle an (Algorithmus siehe **2.1.2**). Gibt per Boolean zurück, ob dabei ein Stab den Rahmen verlassen hat.

3.2.3 `puzzle rotatePuzzle ( const puzzle& p, const bool clockwise = true )`

Rotiert ein gegebenes Puzzle; dazu wird dieses zunächst per `turnPuzzle` in die geforderte Richtung gedreht und anschließend der Gravitation per `gforcePuzzle` ausgesetzt. Das roterte Puzzle wird zurückgegeben.

3.2.4 `puzzle readFromFile ( const string& filename )`

Liest die per Kommandozeilenparameter (siehe **3.4**) übergebene Datendatei aus, erstellt daraus den initialen Puzzle-Zustand und gibt diesen dann zurück.

3.2.5 `vector<bool> bfs ( const puzzle& initialState )`

Ausführung der Breitensuche; ausgehend von einem gegebenen initialen Zustand wird per Breitensuche nach einem Zustand gesucht, für den `gforcePuzzle` ein `true` zurück liefert. Wird ein solcher Zustand gefunden, wird der Lösungsweg in Form einer Kette von Booleans zurückgeliefert, wobei `true` für eine Drehung im Uhrzeigersinn und `false` entsprechend gegen den Uhrzeigersinn steht.

3.2.6 `int main(int argc, char* argv[])`

Hauptfunktion; liest die Kommandozeilenparameter ein, lädt den Puzzle-Startzustand per `readFromFile` und wendet auf diesen die Breitensuche `bfs` an, die schließlich den Lösungspfad zurückliefert. Es folgt die Auswertung des Lösungspfades; dessen Ausgabe (oder Meldung der Nicht-Existenz) sowie, falls dies per kommandozeilenparameter `-p` angefordert wurde, die schrittweise Ausgabe des Puzzles jeweils nach einer Drehung.

3.3 Genauere Details zur Implementierung der Breitensuche

Die Funktion `bfs` (vgl. **3.2.5**) übernimmt einen initialen Puzzlezustand und startet die Breitensuche; diese verwendet eine Warteschlange `q` (Z.205), die aus Paaren von Zuständen und einer Menge von Entscheidungen, die getroffen wurden, um den jeweiligen Zustand zu erreichen, besteht. Parallel dazu werden die bereits erreichten Zustände in `reachedStates` gespeichert (Z.206).[8] Wurde ein Zustand bereits erreicht, wurde er in die Queue hinzugefügt und ist entweder noch darin oder wurde bereits besucht; beides macht das erneute Betrachten des Zustandes überflüssig und er kann ignoriert werden. Der initiale Status wird mit leerem Entscheidungspfad in die Queue hinzugefügt (Z.208). Solange die Warteschlange nun nicht leer ist, wird das erste Element aus der Schlange genommen (Z.212). Wenn dieses nicht die Zielbedingung erfüllt (Z.215), werden Drehungen in beide Richtungen simuliert (Z.219 + 220) und die daraus resultierenden Zustände (sofern noch nicht in `reachedStates`) (mit erweitertem Entscheidungspfad) zur Queue und `reachedStates` hinzugefügt (Z.222-225 und 228-231). Erfüllt das Element die Zielbedingung, wird dessen bisheriger Entscheidungspfad zurückgegeben und die Funktion beendet (Z.216). Wird die Schleife `while( !q.empty())` (Z.211) nicht vorzeitig beendet, wurden alle möglichen Zustände abgearbeitet und kein Zielzustand gefunden; es existiert also kein Lösungspfad und ein Leerer Lösungspfad wird zurückgegeben (Z.235).

```
203 vector<bool> bfs ( const puzzle& initialState )
204 { // Breitensuche
205     queue< pair< puzzle, vector<bool> > > q; // Warteschlange; ↩
        [(Puzzle, decisionPath)]
206     set< puzzle > reachedStates;
207
208     reachedStates.insert( initialState );
```

[8] Da `puzzle` in Wirklichkeit keine eigene Klassendefinition, sondern ein `typedef` eines Templates ist, kann hier ohne zusätzliche manuelle Überladung des „<“-Operators ein binärer Baum (`set`) mit $O(1)$ verwendet werden, was sehr vorteilhaft ist, da sehr oft in `reachedStates` gesucht wird.

```
209     q.push( { initialState, vector<bool>() } ); // Warteschlange mit ↩
            Startzustand initialisieren; leerer Entscheidungspfad
210
211     while( !q.empty() ) {
212         pair< puzzle, vector< bool > > p = q.front(); q.pop(); // ↩
                ersten Zustand aus der Liste herausnehmen
213
214         // Ziel erreicht?
215         if ( stickReleased(p.first) )
216             return p.second; // gib den Entscheidungspfad zu diesem ↩
                    Zustand zurück
217
218         // Zustandsveränderung
219         puzzle a = rotatePuzzle( p.first, true ); // Drehung im ↩
                Uhrzeigersinn
220         puzzle b = rotatePuzzle( p.first, false );
221
222         if ( reachedStates.find( a ) == reachedStates.end() ) {
223             reachedStates.insert( a ); // mark node as reached
224             vector<bool> tempPath = p.second; tempPath.push_back( true );
225             q.push( make_pair( a, tempPath ));
226         }
227
228         if ( reachedStates.find( b ) == reachedStates.end() ) {
229             reachedStates.insert( b ); // mark node as reached
230             vector<bool> tempPath = p.second; tempPath.push_back( ↩
                    false );
231             q.push( make_pair( b, tempPath ));
232         }
233
234     }
235     return *( new vector<bool>() ); // Leeren Lösungspfad zurückgeben, ↩
            wenn keine Lösungsmöglichkeit gefunden
236
237 }
```

Listing 1: 🗀 3‣src‣main.cpp - Breitensuche-Funktion `bfs`.

3.4 Kompilat

Der Quelltext wurde für Linux in verschiedenen Optimierungsstufen unter 64 Bit kompiliert; Die Executables sind unter 🗀 3‣bin‣linux_64_o*.out zu finden.
Kommandozeilenparameter:

```
> ./bin/linux_64_o3.out --help
[...]
args:
-f --file Data file to be opened.
-p --printFields Print the solutions step by step.
```

4 Beispiele

4.1 Beispiellösung schittweise

Beispielhaft soll hier die Lösung eines einfachen Puzzles gezeigt werden;

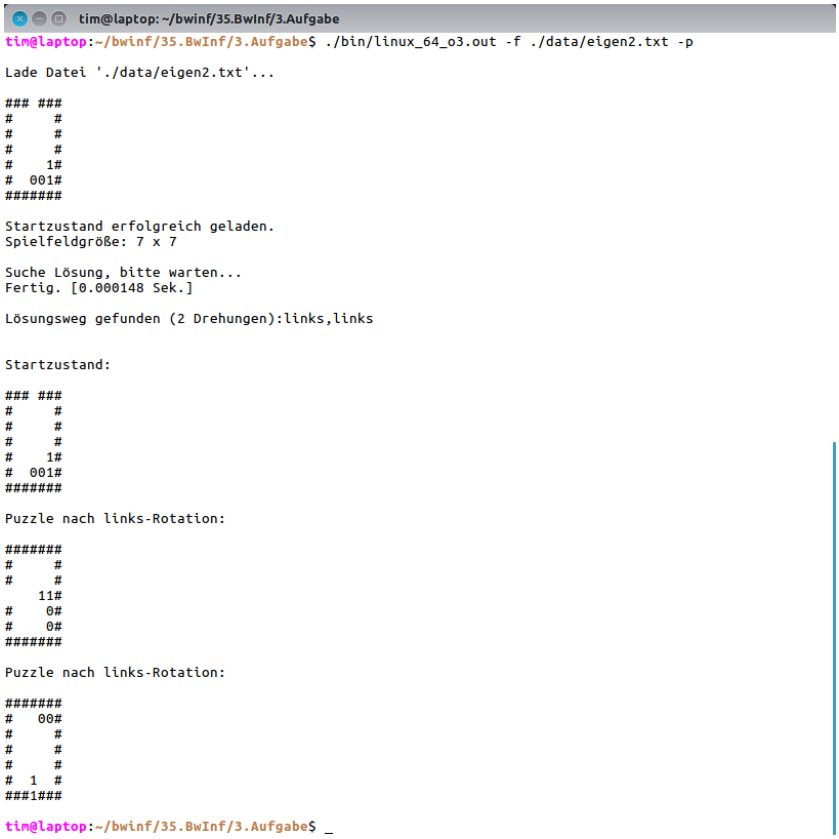

Abbildung 1: Invertierte Konsolenausgabe zur schrittweisen Lösung von 🗀 3 ▸ data ▸ eigen2.txt .

4.2 Gegebene und eigene Beispiele

Hinweis - Ausgaben
Sie finden detaillierte Ausgaben zu jeder Eingabedatei in 🗀 3▸data▸out▸*.txt .

Datei	Spielfeld	Laufzeit [Sek.]	Ausgegebener Lösungsweg
rotation1_03.txt	######## # 0# # 0# #112222# #33 4# #55 4# #666 4# ### ####	0.002173 Sek.	(6): links, links, links, links, rechts, links
rotation2_03.txt	############ # # # 01 # # 01 # # 01 # # 222222# # 34 5 # # 34 5 # # 634 5 # # 63477775 # # 63888885 # ###### #####	0.01149 Sek.	(22): rechts, rechts, rechts, rechts, links, rechts, rechts, links, links, links, links, links, links, links, rechts, rechts, rechts, rechts, rechts, rechts, links, rechts
rotation3_03.txt	########## # # # # # # # 7775# # 11 5# # 2 888# #990233 # #44066666# ##### ####	6.6692 Sek.	(90): links,rechts,rechts,rechts,rechts,links,links,links,rechts,rechts,rechts,links,links,links,links,links,links,links,rechts,rechts,links,links,rechts,rechts,rechts,rechts,rechts,rechts,rechts,rechts,rechts,rechts,links,rechts,rechts,rechts,rechts,rechts,rechts,rechts,links,links,links,rechts,rechts,rechts,rechts,rechts,links,links,links,links,links,rechts,rechts,rechts,rechts,links,links,rechts,rechts,rechts,rechts,links,links,links,links,links,links,links,links,links,links,links,links,rechts,links,links,rechts,rechts,rechts,rechts,rechts,links,rechts

eigen1.txt	##### #0 # #0 # # # ## ##	0.000139 Sek.	- Keine Lösung gefunden -
eigen2.txt	### ### # # # # # # # 1# # 001# #######	0.000154 Sek.	(2): links, links
eigen3.txt	######### # 00 1 # # 2 1 # #32 14# #32 4# #32 4# #55 66# #777 88# #### ####	0.00108 Sek.	(4): rechts,rechts,links,links

5 Quelltext

```
1  /** Aufgabe 3 **/
2  #include <iostream>
3  #include <fstream>
4  #include <sstream>
5
6  #include <string>
7  #include <vector>
8  #include <queue>
9  #include <set>
10 #include <map>
11
12 #include <algorithm>
13
14 #include <cctype>
15 #include <ctime>
16 #include <cstdlib>
17
18 using namespace std;
19
20 /** "Typen" **/
21 typedef pair< int, int > coord;
22 typedef vector< vector< int > > puzzle;
23
24 /** Flags **/
25 bool flag_logEnabled = true;
26 bool flag_printSolutionFields = false;
27
28 /** Makros **/
29 #define F_WALL  -200
30 #define F_EMPTY -100
31
32 #define LOG if(flag_logEnabled) cout << endl
33
34 /** Funktionen **/
35
36 //** Ein- und Ausgabe **//
37 void exitError( string msg = "FEHLER: Unbehandelter Fehler ↩
       aufgetreten. Beende." /* (default message) */ )
38 { // Prints an error message and exits with status FAILURE
39     cerr << /* endl <<*/  msg << endl;
40     exit(EXIT_FAILURE);
41 }
42
43 void printPuzzle( const puzzle& p )
44 { // Puzzle auf der Konsole ausgeben
45
46     cout << endl;
47     for( int y = p.size() - 1; y >= 0; y-- ) {
48         cout << endl;
49         for( int x = 0; x < p.size(); x++ ) {
```

```
50              switch( p[x][y] ) {
51                  case F_WALL:
52                      cout << "#"; break;
53                  case F_EMPTY:
54                      cout << " "; break;
55                  default:
56                      cout << p[x][y]; break;
57              }
58          }
59      }
60      cout << endl;
61  }
62
63  puzzle readFromFile ( const string& filename )
64  { // Datendatei auslesen; gibt initialen Puzzle-Status zurück
65      // Datei öffnen
66      ifstream fin ( filename.c_str() );
67      if ( !fin.good() )
68          exitError( "FEHLER: Die angegebene Datei '" + filename + "' ↩
                konnte nicht geöffnet werden. Beende." );
69
70      int N = 0; // Spielfeldgröße
71
72      string line = "";
73      getline( fin, line );
74      istringstream iss (line);
75      if (!(iss >> N ))
76          exitError( "FEHLER: Fehler beim Auslesen der ersten Zeile aus ↩
                der Datendatei (Feldgröße). Beende." );
77
78      puzzle initialState (N, vector<int>(N)); // Startzustand des Puzzles
79
80      for( int y = N-1; y >= 0; y--) {
81          string line;
82
83          if ( !getline( fin, line ) )
84              exitError("FEHLER - Konnte Zeile Nr. " + to_string(N- ↩
                    (y-1) ) + " aus Datendatei nicht auslesen.");
85
86          if ( line.length() != N ) // Nicht genau N Zeichen in Zeile?
87              exitError("FEHLER - Zeile Nr. " + to_string(N-(y-1)) + " ↩
                    enthält nicht " + to_string(N) + ", sondern " + ↩
                    to_string(line.length()) + " Zeichen!" );
88
89          for( int x = 0; x < N; x++) {
90              string s = line.substr( x, 1); //aktuelles Zeichen auslesen
91
92              if ( s == "#" ) { // Wand
93                  initialState[x][y] = F_WALL;
94              }else if( s == " " ) { // Leeres Feld
95                  initialState[x][y] = F_EMPTY;
96              }else if(isdigit(s[0])) { // Stab
97                  initialState[x][y] = atoi( s.c_str() );
```

```
98              }else // Unbekanntes Zeichen
99                  exitError( "FEHLER - Unbekanntes Zeichen '" + s + "' in Datendatei! Beende." );
100         }
101     }
102
103     return initialState;
104 }
105
106
107 //** Simulation: Drehung und Gravitation **//
108 puzzle turnPuzzle ( const puzzle& p, const bool clockwise = true )
109 { // Drehen des Puzzles (Helferfunktion) - Keine(!) Anwendung der Gravitation
110     if ( clockwise )
111     { // Drehung im Uhrzeigersinn
112         puzzle ret (p.size(), vector<int>(p.size()));
113
114         for( int x = 0; x < p.size(); x++ )
115             for( int y = 0; y < p[0].size(); y++ )
116                 ret[y][p.size() - x - 1] = p[x][y]; // (x,y) => (y, N - x - 1)
117
118         return ret;
119     }else
120     { // 1 Drehung gegen den Uhrzeigersinn = 3 Drehungen mit dem Uhrzeigersinn
121         return turnPuzzle( turnPuzzle( turnPuzzle(p, true), true), true);
122     }
123 }
124
125 bool gforcePuzzle( puzzle& p )
126 { // Wendet Gravitation auf die Stäbe eines Puzzles an und gibt zurück, ob ein Stab durch die Lücke gefallen ist
127
128     bool stickFallen = true;
129     while( stickFallen ){
130         stickFallen = false;
131
132         // Stäbe nach ihrer geringsten Höhe sortieren
133         map< int, int> stickHeights;
134
135         for( int y = p.size() - 1; y >= 0; y-- )
136             for( int x = 0; x < p[0].size(); x++ )
137                 if ( p[x][y] != F_WALL && p[x][y] != F_EMPTY ) { // Stab, da keine Wand oder Leeres Feld
138                     if ( stickHeights.find( p[x][y] ) == stickHeights.end() ) {
139                         stickHeights.insert( make_pair(p[x][y], y) );
140                     }else{
141                         if ( stickHeights.at( p[x][y] ) > y )
142                             stickHeights.at( p[x][y] ) = y;
143                     }
```

```
144                 }
145
146         // Nach Y-Wert sortieren
147         vector< pair<int,int> > heightsSorted;
148         for( auto i : stickHeights)
149             heightsSorted.push_back( make_pair(i.second, i.first) );
150
151         sort( heightsSorted.begin(), heightsSorted.end() );
152
153
154         for( auto s : heightsSorted )
155         { // for each stick
156
157             // maximales Fall-Y ermitteln
158             int maxFallY = -1;
159
160             for( int x = 0; x < p.size(); x++ )
161             {
162                 if ( p[x][s.first] != s.second ) continue;
163                 int y = s.first - 1; for( ; y >= 0 && p[x][y] == ↩
                        F_EMPTY; y-- ) {} y++;
164                 maxFallY = max( maxFallY, y );
165             }
166
167             if ( s.first > maxFallY )
168             { // Dieser Stab kann fallen
169                 for( int x = 0; x < p.size(); x++ )
170                     for( int y = 0; y < p.size(); y++ )
171                         if ( p[x][y] == s.second ) {
172                             p[x][y] = F_EMPTY;
173                             p[x][y - abs(s.first-maxFallY) ] = s.second;
174                         }
175
176                 stickFallen = true;
177                 break;
178             }
179
180             if ( maxFallY == 0 ) return true;
181
182         }
183
184     }
185
186     return false;
187 }
188
189 puzzle rotatePuzzle ( const puzzle& p, const bool clockwise = true )
190 { // Drehung des Puzzles in eine Richtung; Anwendung von Gravitation; ↩
      gibt ein neues, gedrehtes Puzzle zurück, ohne das übergebene Puzzle ↩
      zu verändern
191     puzzle temp = turnPuzzle( p, clockwise );
192     gforcePuzzle(temp);
193     return temp;
```

```
194 }
195
196 bool stickReleased( const puzzle& p )
197 { // Gibt true zurück, wenn ein Stab durch die Lücke / aus dem Rahmen ↩
        fällt
198     puzzle temp = p;
199     return gforcePuzzle(temp);
200 }
201
202 //** Breitensuche **//
203 vector<bool> bfs ( const puzzle& initialState )
204 { // Breitensuche
205     queue< pair< puzzle, vector<bool> > > q; // Warteschlange; ↩
            [(Puzzle, decisionPath)]
206     set< puzzle > reachedStates;
207
208     reachedStates.insert( initialState );
209     q.push( { initialState, vector<bool>() } ); // Warteschlange mit ↩
            Startzustand initialisieren; leerer Entscheidungspfad
210
211     while( !q.empty() ) {
212         pair< puzzle, vector< bool > > p = q.front(); q.pop(); // ↩
                ersten Zustand aus der Liste herausnehmen
213
214         // Ziel erreicht?
215         if ( stickReleased(p.first) )
216             return p.second; // gib den Entscheidungspfad zu diesem ↩
                    Zustand zurück
217
218         // Zustandsveränderung
219         puzzle a = rotatePuzzle( p.first, true ); // Drehung im ↩
                Uhrzeigersinn
220         puzzle b = rotatePuzzle( p.first, false );
221
222         if ( reachedStates.find( a ) == reachedStates.end() ) {
223             reachedStates.insert( a ); // mark node as reached
224             vector<bool> tempPath = p.second; tempPath.push_back( true );
225             q.push( make_pair( a, tempPath ));
226         }
227
228         if ( reachedStates.find( b ) == reachedStates.end() ) {
229             reachedStates.insert( b ); // mark node as reached
230             vector<bool> tempPath = p.second; tempPath.push_back( ↩
                    false );
231             q.push( make_pair( b, tempPath ));
232         }
233
234     }
235     return *( new vector<bool>() ); // Leeren Lösungspfad zurückgeben, ↩
            wenn keine Lösungsmöglichkeit gefunden
236
237 }
238
```

```
239 int main(int argc, char* argv[])
240 {
241     // Kommandozeilenparameter auswerten
242     std::vector<std::string> args;
243    for (int i = 1 /*(skip first arg)*/; i < argc; ++i)
244       args.push_back(argv[i]);
245 
246     string filename;
247     for( int i = 0; i < args.size(); i++ )
248     {
249         if ( args[i] == "-?" || args[i] == "--help" ) {
250             // Hilfe-Ausgabe
251             cout << endl << "35. Bundeswettbewerb Informatik 2016/'17, 1.Runde, Aufgabe 3 - 'Rotation'.";
252             cout << endl << "Lösung von Team 'Wouldn't IT be nice...'";
253             cout << endl << endl << "args: ";
254             cout << endl << "-f --file Data file to be opened.";
255             cout << endl << "-p --printFields Print the solutions step by step.";
256             cout << endl;
257             return EXIT_SUCCESS;
258         }
259         else if ( args[i] == "-f" || args[i] == "--file" ) {
260             if ( i+1 >= args.size() )
261                 exitError( "Fehler - Kein Dateiname nach '" + args[i] + "'");
262             filename = args[i+1];
263             i++;
264         }
265         else if( args[i] == "-p" || args[i] == "--printFields" ){
266             flag_printSolutionFields ^= 1; // flip bool value
267         }
268         else {
269             cerr << endl << "Warnung: Unerkannter Kommandozeilenparameter '" << args[i] << "'. Ignoriere.";
270         }
271     }
272 
273     // Einlesen der Datei
274     if ( filename == "" )
275         exitError( "Fehler: Kein Dateiname übergeben. Verwenden Sie den Kommandozeilenparameter '-f <Dateiname>' oder '--help'. Beende." );
276 
277     cout << endl << "Lade Datei '" << filename << "'... ";
278 
279     puzzle initialState = readFromFile( filename );
280     printPuzzle( initialState );
281 
282     LOG << "Startzustand erfolgreich geladen.";
283     LOG << "Spielfeldgröße: " << initialState.size() << " x " << initialState[0].size();
284 
```

```
285     cout << endl << endl << "Suche Lösung, bitte warten... " << endl;
286
287     double start_t = clock(); // Laufzeit stoppen
288     vector<bool> solutionPath = bfs( initialState ); // Breitensuche ↩
            durchführen
289     double end_t = clock();
290
291     cout << "Fertig. [" << ( (double) (end_t - start_t) / ↩
            CLOCKS_PER_SEC ) << " Sek.]" << endl;
292
293     if ( solutionPath.size() != 0)
294     { // Lösungsweg gefunden
295         cout << endl << "Lösungsweg gefunden (" << solutionPath.size() ↩
                << " Drehung" << (solutionPath.size() > 1 ? "en" : "") << ↩
                "):";
296
297         for( int i = 0; i < solutionPath.size(); i++ )
298             cout << ( ( i == 0 ) ? "" : "," ) << ( solutionPath[i] ? ↩
                    "rechts" : "links" );
299
300         if ( flag_printSolutionFields )
301         { // Ausgabe der Felder mit Verdrehung schrittweise
302
303             puzzle temp = initialState;
304
305             cout << endl << endl << endl << "Startzustand: ";
306             printPuzzle( temp );
307
308             for( auto i : solutionPath ) {
309                 temp = rotatePuzzle( temp, i);
310                 cout << endl << "Puzzle nach " << ( i ? "rechts" : ↩
                        "links" ) << "-Rotation:";
311                 printPuzzle( temp );
312             }
313
314         }
315
316     }else if( stickReleased(initialState) )
317     { // Im Startzustand fällt ein Stab aus dem Rahmen
318         cout << endl << "HINWEIS - Bereits im Startzustand fällt ein ↩
                Stab durch die Lücke! Keine Rotation erforderlich.";
319     }else
320     { // Keine Lösung gefunden
321         cout << endl << "Kein Lösungsweg gefunden.";
322     }
323
324     cout << endl;
325     return EXIT_SUCCESS;
326 }
```

Listing 2: 🗀 3‣src‣main.cpp - Quelltext der main.cpp