Aufgabe 1

33.Bundeswettbewerb Informatik 2014/'15

Der Script-Tim

Inhaltsverzeichnis

1 Allgemeines Problem **2**

2 Lösungsidee **2**

3 Umsetzung in ein Programm **3**

3.1 Wahl der Programmiersprache 3

3.2 Grober Programmablauf 3

3.3 Programmablauf 3

3.3.1 Allgemeine Variablen, Klassen, Strukturen 3

3.3.2 Eingabe der Behälterdaten durch den Benutzer 4

3.3.3 Ermitteln der Inhaltssumme 4

3.3.4 Initiieren der Umfüllschleife 5

3.3.5 Die Umfüllschleife 5

3.3.6 *umfuellen* 6

3.3.7 setZustand 7

3.3.8 checkGleichVerteilt 8

3.3.9 Ausgabe: Kürzesten gefundenen Lösungsweg ermitteln 8

3.3.10 Ausgabe: Alle Lösungswege ausgeben 9

4 Anwendung auf gegebene Beispiele **9**

1 Allgemeines Problem

Wie kann man den kürzesten Lösungsweg ermitteln?

- **Mit einem Algorithmus?**
  Zuerst kam mir der Gedanke, einen statischen Algorithmus zu entwerfen, den man solange ausführt, bis man zu einem Lösungsweg gekommen ist. Dieser Gedanke scheiterte an seiner Umsetzung(siehe Abb.1) und auch daran, dass die Anzahl der Behälter beliebig sein sollte und deshalb für jede Behälteranzahl ein neuer Algorithmus entworfen werden müsste.

- **Mit Brute Force?**
  Wenn man alle (logisch) möglichen Umfüllungsmöglichkeiten durchgehen würde, muss man doch auch zum kürzesten Lösungsweg kommen! Der Ansatz hat etwas - nur kommt man neben dem einzig wahren Lösungsweg zu unendlich vielen anderen. Man sollte deshalb mit Bedingungen arbeiten.

2 Lösungsidee

Meine schlussendliche Lösungsidee basiert auf der Brute-Force Idee. Bei jeder Umfüllung muss gelten:

1. Quelle != Ziel
2. !Quelle.leer
3. !Ziel.voll
4. Der durch die Umfüllung erreichte Zustand darf nicht durch eine kürzere Anzahl an Umfüllungen erreicht werden können
5. (Die vorhergegangene Umfüllung darf nicht rückgängig gemacht werden; fällt unter 4., da der vorherige Zustand ja bereits vorhanden war und somit mit einer geringeren Zahl an Umfüllungen erreicht werden kann).

Die Kernidee besteht darin, dass aus der endlosen Menge an Daten - man kann ja theoretisch unendlich oft umfüllen - eine durch den Computer bearbeitbare endliche Menge wird, da man immer an einen Punkt kommen wird, an dem jede Umfüllung zu einem bereits erreichten Zustand führen würde. Eine Aneinanderreihung solcher kürzester Zustände kann - so mein Gedanke - nur zu einem kürzesten Lösungsweg führen, denn würde man zu einem bereits erreichten Zustand kommen, sind die dazwischenliegenden Umfüllungen unnötig und im Sinne des Zieles - den kürzesten Lösungsweg zu finden - nicht hilfreich.

3 Umsetzung in ein Programm

3.1 Wahl der Programmiersprache

Als Programmiersprache habe ich C++ gewählt. Da ich kein Freund von CLR bin, wird es dieses mal wieder eine Konsolenanwendung.

3.2 Grober Programmablauf

Grob aufgeteilt besteht der Programmablauf aus drei Schritten:

1. Eingabe der Behälterdaten durch den Benutzer
2. Kürzesten Lösungsweg ermitteln
3. Ausgabe aller ermittelten Lösungswege

Von Grundprinzip ist es eine Unendliche Umfüllung jedes Behälters in jeden anderen - nur mit bestimmten Umfüllbedingungen. Wesentlicher Bestandteil des Programmes ist die Speicherung der bereits erreichten Behälterzustände.

3.3 Programmablauf

3.3.1 Allgemeine Variablen, Klassen, Strukturen

Um ermitteln zu können, ob ein Zustand bereits erreicht wurde, definiere ich einen vektor-Container *zustaende*, in dem bei jeder Umfüllung der erreichte Zustand abgelegt wird.

Um die Behälter, Pfade, ermittelte Lösungswege und Zustände leichter verwalten zu können, kommen noch verschiedene Klassen(*Behaelter*) und Strukturen(*SctructPfad*, *StructZustaende*) zum Einsatz. Die vom Benutzer eingegebenen Behälterdaten liegen als Container im Vektor *behaelter* des Types *Behaelter* vor.

Allgemein wird versucht, jeden Behälter in jeden anderen Behälter umzufüllen, sofern bestimmte logische Bedingungen - siehe oben - erfüllt sind. Dies findet in der Funktion *loop* statt, die sich bei jeder Umfüllung selbst aufruft und dann wiederum jeden Behälter in jeden anderen umfüllt. Die Initiierung dieser Schleifen-Funktion erfolgt in der *main*-Funktion.

Zusätzlich wird vor jeder Umfüllung (in der Funktion *loop*) überprüft, ob der durch die Umfüllung erreichte Zustand bereits erreicht worden wäre (= in *zustaende* vorhanden ist): wurde der Zustand bereits erreicht und wurden dabei weniger Umfüllungen benötigt, wird die Umfüllung übersprungen. Andernfalls wird der (falls vorhandene) Zustand überschrieben.

3.3.2 Eingabe der Behälterdaten durch den Benutzer

Am Anfang des Programmes wird der Benutzer aufgefordert, die Behälterdaten einzugeben;

```
184     vector<Behaelter> behaelter;
185     char c = 'n';
186     do{
187         //Wenn c != '', eingabe
188         if (c == 'j'){
189             unsigned int besitzer = 0, kapazitaet = 0, inhalt = 0;
190             cout << "\n\n***** Neuen Behaelter erstellen *****";
191             do{
192                 cout << "\nBesitzer des neuen Behaelters: [0/1]:";
193                 cin >> besitzer;
194             }while(besitzer < 0 || besitzer > 1);
195 
196             do{
197                 cout << "\nKapazitaet des Behaelters [Ma(ss)e Wein]:";
198                 cin >> kapazitaet;
199             }while(kapazitaet < 0);
200 
201             do{
202                 cout << "\nWein im Behaelter (Inhalt):";
203                 cin >> inhalt;
204             }while(inhalt < 0 || inhalt > kapazitaet);
205 
206             behaelter.push_back( * new Behaelter( besitzer, kapazitaet, inhalt));
207         }
208         //Liste der Behälter anzeigen
209         cout << "\n====== Behaelter: " << behaelter.size() << " ========";
210         for (unsigned int x = 0; x < behaelter.size(); x++){
211             cout << "\n [" << x << "] ( Besitzer: " << behaelter[x].besitzer << ",
                    Kapazitaet: " << behaelter[x].kapazitaet << ", Inhalt: " << behaelter[
                    x].inhalt << ")";;
212         }
213 
214         cout << "\n\nEinen neuen Behaelter hinzufuegen?\n";
215         cout << " +----- Optionen: ----------+\n";
216         cout << " | j: Behaelter hinzufuegen |\n";
217         cout << " | n: Fortfahren            |\n";
218         cout << " +--------------------------+\n";
219         cin >> c;
220 
221     }while(c != 'n');
```

Schlussendlich sind die Behälter im Vektor *behaelter* gespeichert.

3.3.3 Ermitteln der Inhaltssumme

Um während des Programmablaufes feststellen zu können, ob der Wein auf beide Personen gerecht aufgeteilt wurde, muss zunächst einmal die Insgesamt-summe des ein-

gegebenen Weines ermittelt werden. Dies geschieht unmittelbar nach der Eingabe der Behälter:

```
223     /* Summe der Inhalte insgesamt ermitteln */
224     for (unsigned int x = 0; x < behaelter.size(); x++){ SummeInhalt += behaelter[x].
            inhalt; }
```

3.3.4 Initiieren der Umfüllschleife

Ist dies erfolgreich erledigt, startet das Programm die Ermittlung des kürzesten Lösungsweges; mit zwei ineinander verschachtelten *for*-Schleifen initiiert es die Umfüllschleife *loop*(Z.241). Hierbei wird zunächst eine Umfüllung simuliert(Z.231 - 238) und dann ermittelt, ob der durch die simulierte Umfüllung erreichte Zustand bereits erreicht worden sein würde (= in *zustaende* enthalten)(Z.240).

```
227     /* Schleifen initiieren */
228     for (unsigned int sourceID = 0; sourceID < behaelter.size(); sourceID++){
229         for(unsigned int targetID = 0; targetID < behaelter.size(); targetID++){
230             if (sourceID != targetID && !behaelter[sourceID].leer() && !behaelter[
                    targetID].voll()){ //Umfüllung formal/logisch korrekt?
231                 //Umfüllung simulieren
232                 vector<Behaelter> tempBehaelter = umfuellen(behaelter, sourceID,
                        targetID);
233                 //Pfad erweitern
234                 vector<StructPfad> tempPfad;
235                 StructPfad temp;
236                 temp.source = sourceID;
237                 temp.target = targetID;
238                 tempPfad.push_back (temp);
239
240                 if (setZustand (tempBehaelter, tempPfad)){ //Wenn Zustand nicht
                        gesetzt oder überschrieben, Schleife starten
241                     loop(tempBehaelter, tempPfad);
242                 }
243             }
244         }
245         //cout << "\n" << 100/behaelter.size()*sourceID << "%";
246     }
```

3.3.5 Die Umfüllschleife

Die Umfüllschleife - die Funktion *loop* - prüft zunächst, ob der Wein - nach der Umfüllung vor dem Aufruf der Funktion - gerecht auf die Personen aufgeteilt ist (Z.145 - 149). Ist dies der Fall, ist es nicht weiter nötig, die Schleife fortzusetzen. Andernfalls startet die Funktion eine neue 'Umfüllinstanz', d,h, sie versucht, jeden Behälter in jeden anderen zu füllen, sofern die logischen Bedingungen erfüllt sind (Z.150 - 170); genau wie in Zeile 228-244 wird zunächst eine Umfüllung simuliert (Z.154-161) und die Schleife aber nur dann um eine Instanz weiter vertieft, wenn der simulierte Zustand nicht bereits erreicht wurde (Z.162f.).

```
142 /* Schleifen- Funktion */
143 void loop (vector<Behaelter> behaelter, vector<StructPfad> pfad ){
144
145     //Gleich verteilt?
146     if (checkGleichVerteilt(behaelter)){
147         loesungen.push_back (pfad);//Lösungen hinzufügen
148         return; //und Ende
149     }
150     //Sonst wird die Schleife weiter vertieft
151     for(unsigned int sourceID = 0; sourceID < behaelter.size(); sourceID++){
152         for(unsigned int targetID = 0; targetID < behaelter.size(); targetID++){
153             if (sourceID != targetID && !behaelter[sourceID].leer() && !behaelter[
                    targetID].voll()){
154                 //Umfüllung simulieren
155                 vector<Behaelter> tempBehaelter = umfuellen(behaelter, sourceID,
                        targetID);
156                 //Pfad vervollständigen
157                 vector<StructPfad> tempPfad = pfad;
158                 StructPfad temp;
159                 temp.source = sourceID;
160                 temp.target = targetID;
161                 tempPfad.push_back (temp);
162                 //Wenn der Zustand der nächsten Umfüllung noch nicht erreicht war oder
                        den alten überschrieben hat, wird die Schleife fortgesetzt
163                 if (setZustand(tempBehaelter, tempPfad)){
164                     loop(tempBehaelter, tempPfad);
165
166                 }
167
168             }
169         }
170     }
171
172     return;
173 }
```

3.3.6 *umfuellen*

In *umfuellen* wird die eigentliche Umfüllung vorgenommen:

```
60 /* In dieser Funktion wird die eigentliche Umfüllung vorgenommen: von behaelter[
       sourceID] => behaelter[targetID] */
61 vector<Behaelter> umfuellen(vector<Behaelter> behaelter, unsigned int sourceID,
       unsigned int targetID){
62
63     //Umfüllen
64     if (behaelter[sourceID].inhalt == behaelter[targetID].frei()){ //Passt genau
65         behaelter[sourceID].inhalt = 0;
66         behaelter[targetID].inhalt = behaelter[targetID].kapazitaet;
67     }else if (behaelter[sourceID].inhalt > behaelter[targetID].frei()){ //Zu viel
           Inhalt in source für target
68         behaelter[sourceID].inhalt -= behaelter[targetID].frei();
```

```
69          behaelter[targetID].inhalt = behaelter[targetID].kapazitaet;
70      }else{ //Zu wenig Inhalt in source für target
71          behaelter[targetID].inhalt += behaelter[sourceID].inhalt;
72          behaelter[sourceID].inhalt = 0;
73      }
74      return behaelter;
75  }
```

3.3.7 setZustand

setZustand ist neben *loop* das Herzstück meines Programmes: es überprüft, ob und mit wie vielen Umfüllungen ein Zustand bereits erreicht wurde und überschreibt ggf. einen Zustand mit mehr Umfüllungen.

```
95  bool setZustand(vector<Behaelter> zustand, vector<StructPfad> pfad){
96      //Zustand vorhanden?
97
98      if (zustandVorhanden(zustand)){
99
100         unsigned int id = 0;
101         for(unsigned int x = 0; x < zustaende.size(); x++){
102             bool gleich = true;
103             for (unsigned int y = 0; y < zustaende[x].behaelter.size(); y++){
104                 if (zustand[y].inhalt != zustaende[x].behaelter[y].inhalt ) gleich =
                        false;
105             }
106             if (gleich == true) id = x;
107         }
108
109         if (zustaende[id].pfad.size() > pfad.size()){
110             //Zustand überschreiben
111             StructZustaende temp;
112             temp.pfad = pfad;
113             temp.behaelter = zustand;
114             zustaende[id] = temp;
115         }else{
116             return false;
117         }
118
119     }else{
120         //Zustand belegen
121         StructZustaende temp;
122         temp.pfad = pfad;
123         temp.behaelter = zustand;
124         zustaende.push_back(temp);
125         return true;
126     }
127 }
```

Diese Funktion bedient sich ihrerseits einer weiteren Funktion namens *zustandVorhanden*, die einen booleschen Wert zurückliefert, ob der Zustand in Parameter 1 bereits in *zustaende* vorhanden ist.

```
77 /* Ermittelt, ob ein Zustand 'zustand' bereits vorhanden ist */
78 bool zustandVorhanden(vector<Behaelter> zustand){
79
80     for(unsigned int x = 0; x < zustaende.size(); x++){
81         bool gleich = true;
82
83         for(unsigned int y = 0; y < zustand.size(); y++){
84             if (zustaende[x].behaelter[y].inhalt != zustand[y].inhalt) gleich = false;
85         }
86
87         if (gleich == true) return true;
88     }
89     return false;
90 }
```

3.3.8 checkGleichVerteilt

Die Funktion *checkGleichVerteilt*, tut das, was Ihr Name sagt: sie überprüft, ob der Wein gerecht auf die Personen aufgeteilt ist und gibt dementsprechend einen booleschen Wert zurück; hierfür war es nötig, die Insgesamtsumme des Weines zu ermitteln, siehe **3.3.3**

```
129 /* Überprüft, ob der Inhalt gleich auf die Personen verteilt ist */
130 bool checkGleichVerteilt(vector<Behaelter> behaelter){
131     int BesitzPerson[2] = {0, 0};
132     for (unsigned int x = 0; x < behaelter.size(); x++){
133         if (behaelter[x].besitzer == 1){
134             BesitzPerson[0] += behaelter[x].inhalt;
135         }else{
136             BesitzPerson[1] += behaelter[x].inhalt;
137         }
138     }
139     return (BesitzPerson[0] == (SummeInhalt/2) && BesitzPerson[0] == BesitzPerson[1])
          ? true : false;
140 }
```

3.3.9 Ausgabe: Kürzesten gefundenen Lösungsweg ermitteln

Bis hierhin sind alle gefundenen Lösungswege unsortiert in *loesungen* gespeichert. Ich möchte es dem Benutzer nicht zumuten, sich aus der Menge der gefundenen Lösungswege den kürzesten erst noch raus zu suchen. Das kann das Programm machen.

```
248     /* Lösungsweg(e) ausgeben */
249     cout << "\n\n=== Loesungswege ===";
250     if (loesungen.size() == 0){
251         cout << "\nEs wurde kein Loesungsweg gefunden.";
252     }else{
253         cout << "\nGefundene Loesungswege: " << loesungen.size();
254         //Kürzesten Lösungsweg ermitteln
255         int kuerzester = 1000000;
256         int kuerzesterID = 0;
257         for (unsigned int x = 0; x < loesungen.size(); x++){
```

```
258             if (loesungen[x].size() < kuerzester) { kuerzester = loesungen[x].size();
                    kuerzesterID = x; }
259         }
260
261         //Kürzesten Lösungsweg ausgeben
262         cout << "\nKuerzester Loesungsweg: " << loesungen[kuerzesterID].size() << "
                Umfuellungen:";
263         for (unsigned int x = 0; x < loesungen[kuerzesterID].size(); x++){
264             cout << "\n[" << loesungen[kuerzesterID][x].source << "] => [" <<
                    loesungen[kuerzesterID][x].target << "]";
265         }
```

Das Programm geht hier stillschweigend davon aus, dass es nur einen kürzesten Lösungsweg gibt; falls es zwei gibt, wird der zuerst ermittelte ausgegeben. Wenn sich der Benutzer aber eine Liste sämtlicher Lösungswege ausgeben lässt, werden selbstverständlich beide Lösungswege ausgegeben.

3.3.10 Ausgabe: Alle Lösungswege ausgeben

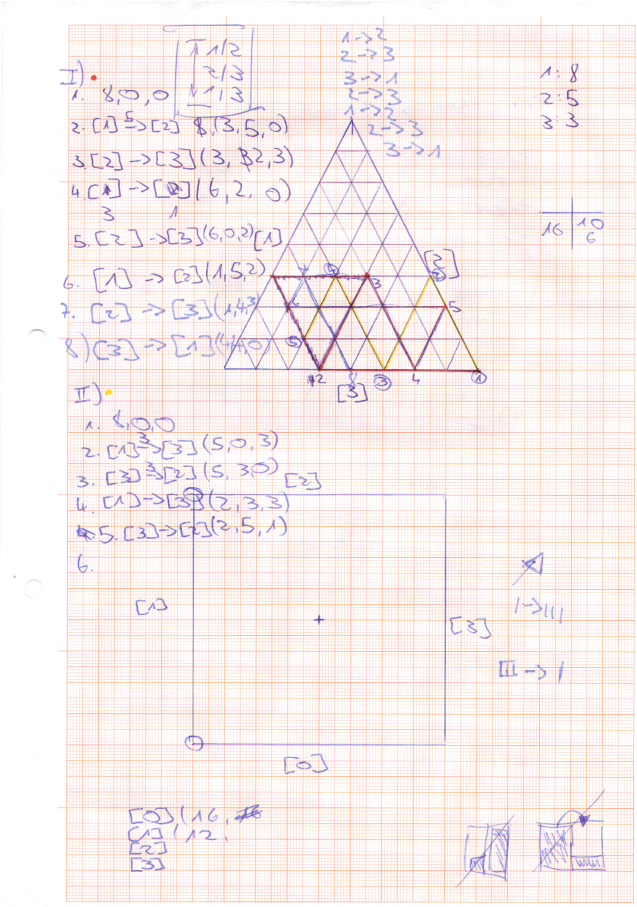
Schließlich kann sich der Benutzer noch eine Liste aller ermittelter Lösungswege ausgeben lassen

```
275             //Alle Lösungswege ausgeben
276             cout << "\n----------------- Alle Loesungswege:";
277             for (unsigned int x = 0; x < loesungen.size(); x++){
278                 cout << "\n\nLoesungsweg Nr. " << x+1 << ": " << loesungen[x].size()
                        << " Umfuellungen";
279                 for(unsigned int y = 0; y < loesungen[x].size(); y++){
280                     cout << "\n[" << loesungen[x][y].source << "] => [" << loesungen[x
                            ][y].target << "]";
281                 }
282             }
```

Information
Wenn Sie sich alle Lösungswege anzeigen lassen wollen, ist es ratsam, die Ausgabe in eine Datei umzulenken, da zumindest die Konsole unter Windows bei mehreren oder längeren Lösungswegen ab einer bestimmten Menge an ausgegebenen Zeilen diese einfach unterschlägt. Zum Lenken der Ausgabe in eine Datei verwenden Sie
> Aufgabe1.exe » ausgabe.txt.

4 Anwendung auf gegebene Beispiele

***Hinweis zum Programm**: Das Programm benötigt zur Ermittlung der Lösungswege z.T. längere Bearbeitungszeit, was aber auch stark von der Leistung Ihres Computers und der Anzahl der Behälter abhängig ist. Ich habe das Programm in einer VM mit wenig Leistung ausgeführt, weshalb die Bearbeitungszeiten z.T. relativ lange waren (steht an den Abbildungen).*



Versuch der Entwicklung eines allgemeingültigen Algorithmus unter Verwendung des Steinhaus'schen Dreiecks (hierfür siehe http://www.hjcaspar.de/mpart/dateien/textdateien/umfuell.htm).
Der Versuch scheiterte, als es unten galt, dieses Prinzip auf ein Viereck (→ vier Behälter) zu übertragen; unmöglich, da jede Seite mit jeder anderen verbunden sein muss, was beim Quadrat, Fünfeck usw.. nicht der Fall ist.

Abbildung 1: Steinhaus'sches Dreieck

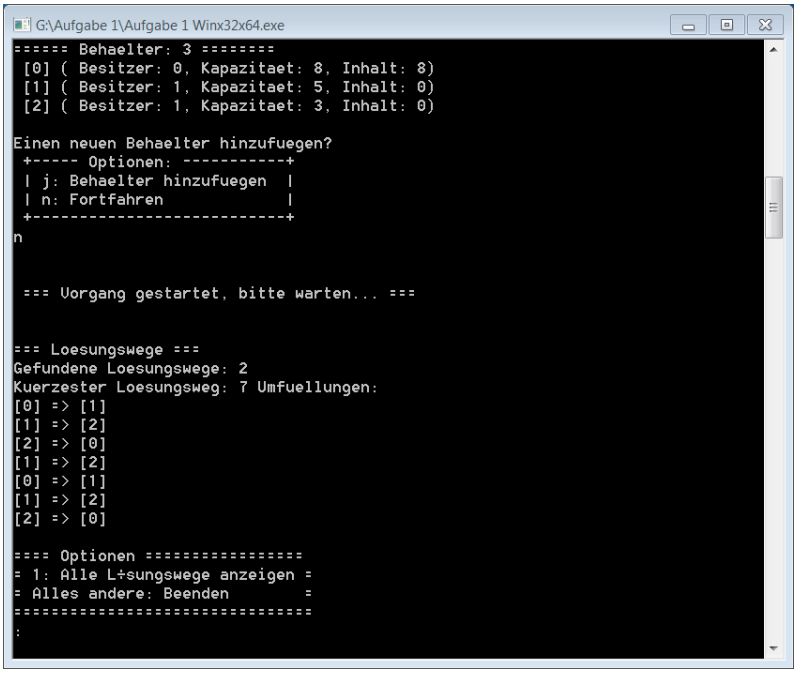


Abbildung 2: Anwendung des Programmes auf Beispiel 1 (Bearbeitungszeit: ca. 0,5 Sek.)

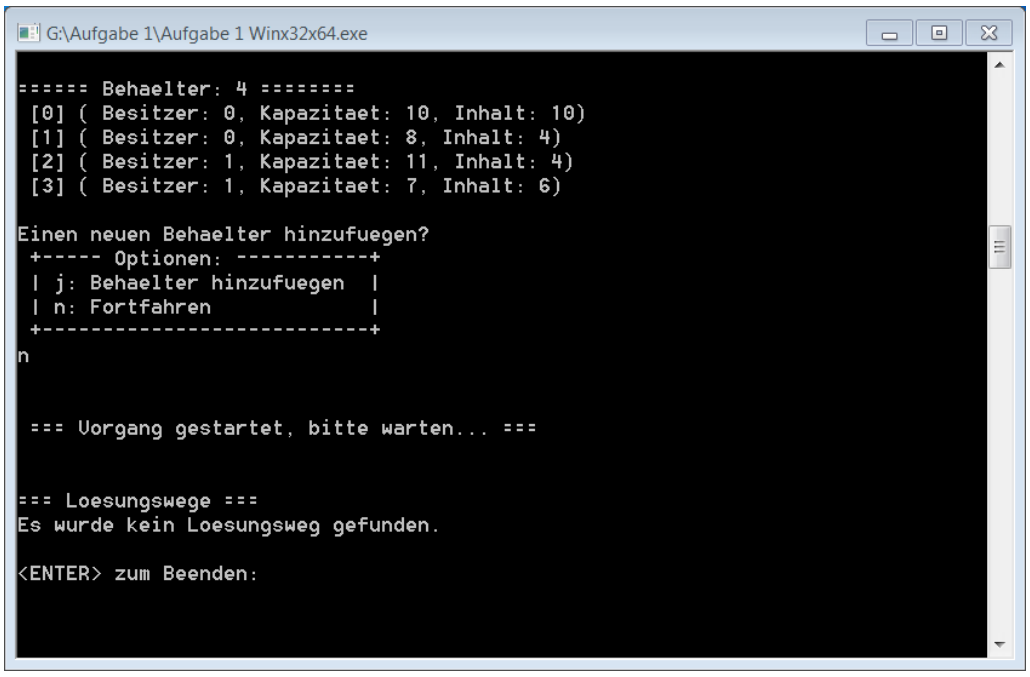


Abbildung 3: Anwendung des Programmes auf Beispiel 2 (Bearbeitungszeit: ca. 30 Sek.)

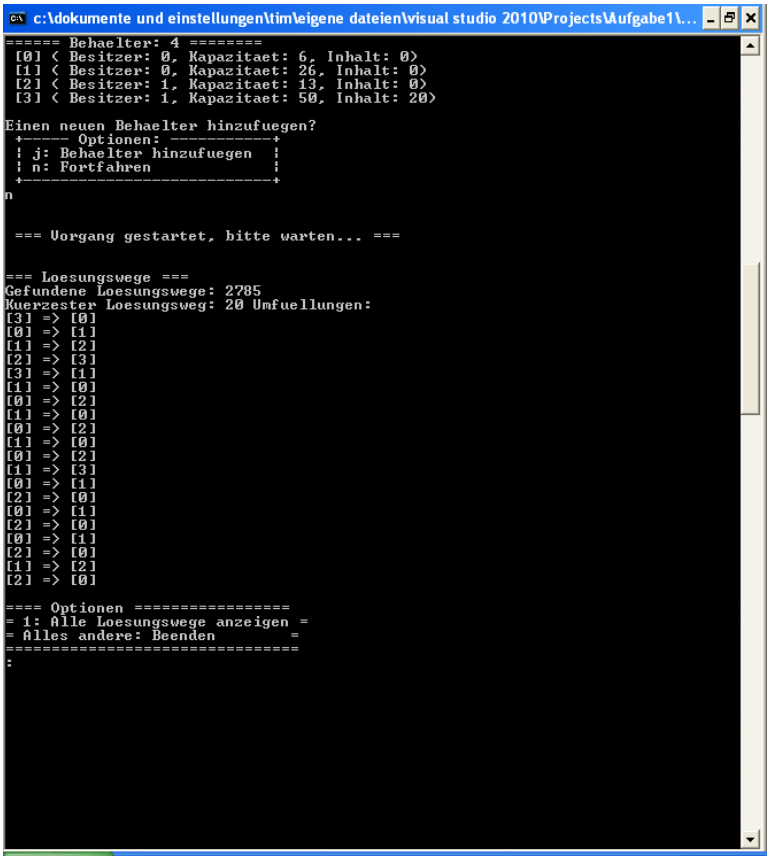


Abbildung 4: Anwendung des Programmes auf Beispiel 3 (Bearbeitungszeit: ca. 5 Min.)