Lösungshinweise und Bewertungskriterien

Allgemeines

Das Wichtigste zuerst: Wir haben uns sehr darüber gefreut, dass wieder besonders viele sich die Mühe gemacht und die Zeit zur Bearbeitung der Aufgaben genommen haben! Die Bewerterinnen und Bewerter haben sich ebenfalls Mühe gegeben und versucht, die Leistungen der Teilnehmerinnen und Teilnehmer so gut wie möglich zu würdigen. Dies wird ihnen aber nicht immer leicht gemacht, insbesondere wenn die Dokumentationen nicht die im Aufgabenblatt genannten Anforderungen erfüllen. Bevor Lösungsideen zu den einzelnen Aufgaben beschrieben werden, soll deshalb auf das Thema „Dokumentation“ näher eingegangen werden. Außerdem werden Einzelheiten zur Bewertung erläutert.

Zu allererst aber etwas Organisatorisches: Sollte der Name auf Urkunde oder Teilnahmebescheinigung falsch geschrieben sein, ist er auch im PMS falsch eingetragen. Die Teilnahmeunterlagen können gerne neu angefordert werden; dann aber bitte auch den Eintrag im PMS korrigieren.

Dokumentation

Die Zeit für die Bewertung ist begrenzt. Folglich ist es nicht möglich, alle eingesandten Programme gründlich zu testen. Die Grundlage der Bewertung ist deshalb die Dokumentation, die, wie im Aufgabenblatt beschrieben, für jede bearbeitete Aufgabe aus Lösungsidee, Umsetzung, Beispielen und Quellcode besteht. Leider sind die Dokumentationen bei vielen Einsendungen sehr knapp ausgefallen, und oft hat das zu den Punktabzügen geführt, die das Erreichen der zweiten Runde verhindert haben.

Ganz besonders wichtig sind *Beispiele*. Wenn Beispiele, insbesondere vorgegebene Beispiele, und die dazu gehörigen Ergebnisse in der Dokumentation fehlen, führt das zu Punktabzug. Es ist nicht ausreichend, Beispiele nur in gesonderten Dateien abzugeben, ins Programm einzubauen oder den Bewertern das Erfinden und Testen von Beispielen zu überlassen.

Auch *Quellcode*, zumindest die für die Lösung wichtigen Teile, gehört in die Dokumentation. Es ist nicht ausreichend, Quellcode nur als Code-Dateien (als Teil der Implementierung) der Einsendung beizufügen.

Zu einer Einsendung gehören als zweiter Teil der Implementierung *Programme*, die möglichst eigenständig lauffähig sind. Für die gängigsten Skript-Sprachen stehen Interpreter zur Verfügung. Einige Entwicklungsumgebungen (z. B. BlueJ), bei denen die erstellten Programme ohne Weiteres nur in der Umgebung selbst laufen, stehen bei der Bewertung zur Verfügung, aber sicher nicht alle. Kompilierung von Quellcode ist während der Bewertung nicht möglich. Deshalb können Programme nur dann mit unterschiedlichen Eingaben getestet werden, wenn diese

vom Programm eingelesen oder über eine (nicht zwingend grafische) Schnittstelle eingegeben werden können.

Auch die folgenden eher inhaltlichen Dinge sind zu beachten:

- *Lösungsideen* sollten keine Bedienungsanleitungen oder Wiederholungen der Aufgabenstellung sein. Es soll beschrieben werden, welches Problem hinter der Aufgabe steckt und wie dieses Problem grundsätzlich angegangen wird. Ein einfacher Grundsatz: Bezeichner von Programmelementen wie Variablen, Prozeduren etc. werden nicht verwendet. Eine Lösungsidee ist nämlich unabhängig von solchen Realisierungsdetails.
- Die *Beispiele* sollen die Korrektheit der Lösung belegen. Es sollten auch Sonderfälle gezeigt werden, die die Lösung behandeln kann. Die Konstruktion solcher Testfälle ist eine ganz wesentliche Tätigkeit des Programmentwurfs.

Vielleicht helfen diese Anmerkungen, wenn du (hoffentlich) im nächsten Jahr wieder mitmachst.

Bewertung

Nun einige Erläuterungen zur Bewertung:

- Pro Aufgabe werden maximal fünf Punkte vergeben. Zu jeder Aufgabe gibt es eine Reihe von Bewertungskriterien. Sie sind in der Bewertung, die man im PMS einsehen kann, aufgelistet. In der ersten Runde gibt es in der Regel nur Punktabzüge; deshalb sind die Kriterien meist negativ formuliert. Hat man bei einem Kriterium 0 Punkte, ist die Einsendung in Bezug auf dieses Kriterium einwandfrei. Wenn das Kriterium nicht erfüllt ist, gibt es in der Regel einen Punkt Abzug (-1), manchmal auch zwei. Wurde die Aufgabe nur ansatzweise oder insgesamt zu schwach bearbeitet, wird ein spezielles Kriterium angewandt, bei dem es bis zu fünf Punkte Abzug geben kann. Im schlechtesten Fall wird die Aufgabenbearbeitung mit null Punkten bewertet.
- In der Hauptliga sind für die Gesamtpunktzahl die drei am besten bewerteten Aufgabenbearbeitungen maßgeblich. Es lassen sich also maximal 15 Punkte erreichen. Einen 1. Preis erreicht man mit 14 oder 15 Punkten, einen 2. Preis mit 12 oder 13 Punkten und einen 3. Preis mit 9 bis 11 Punkten. Mit einem 1. oder 2. Preis ist man für die zweite Runde qualifiziert.
- In der Juniorliga wird ein 1. Preis für 9 oder 10 Punkte, ein 2. Preis für 7 oder 8 Punkte und ein 3. Preis für 5 oder 6 Punkte vergeben. Leider gibt es in der Juniorliga keine zweite Runde.
- Leider wurde in einigen Fällen die Regelung zur Bearbeitung von Junioraufgaben nicht beachtet. Zitat aus dem Mantelbogen des Aufgabenblatts: „Ausgeschlossen von der Bearbeitung der Junioraufgaben sind Schülerinnen und Schüler aus der Qualifikationsphase der gymnasialen Oberstufe sowie aus der Berufsschule.“ Nur wenn ein Team mindestens ein Mitglied hat, das Junioraufgaben bearbeiten darf, darf dessen Einsendung auch Bearbeitungen von Junioraufgaben enthalten.

- Eine Einsendung wird in der Juniorliga gewertet, wenn alle Gruppenmitglieder die Bedingung für Junioraufgaben erfüllen (damit kommt die Einsendung für die Juniorliga in Frage) und in der Einsendung Junioraufgaben bearbeitet wurden. Eine solche Einsendung wird zusätzlich in der Hauptliga gewertet, wenn auch „normale“ Aufgaben bearbeitet wurden.
- Grundsätzlich kann die Dokumentation als „schlecht / nicht vollständig“ bewertet werden, wenn Teile wie Umsetzung, Beispiele oder Quellcode(auszüge) fehlen. Außerdem wird in der Regel gefordert, dass die gewählten Verfahren gut nachvollziehbar beschrieben sind und dass begründet wird, warum sie das gegebene Problem lösen.
- Leider ließ sich nicht verhindern, dass etliche Teilnehmer nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden. Das ist ziemlich riskant, da Fehler sich leicht einschleichen.
- Es ist verständlich, wenn jemand, der nicht weitergekommen ist, über eine Reklamation nachdenkt. Kritische Fälle, insbesondere die mit 11 Punkten, haben wir allerdings schon sehr gründlich und mit viel Wohlwollen geprüft.

Danksagung

An der Erstellung der Lösungsideen haben mitgewirkt: Karl Schrader (Junioraufgaben 1 und 2), Thomas Leineweber (Aufgaben 1, 2 und 4), Florian Behrens (Aufgabe 3) und Nikolai Wyderka (Aufgabe 5).

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

J1 Luftballons

Lösungsidee

In 10 Fächern einer Maschine befinden sich unterschiedliche Anzahlen von Luftballons. Nun soll man einige Fächer so auswählen, dass sich daraus insgesamt möglichst genau 20 Luftballons (und auf keinen Fall weniger als 20 Luftballons) entnehmen lassen. Diese Aufgabenstellung lässt sich in ein Grundproblem und ein paar einfachere Nebenprobleme zerlegen. Als Grundproblem betrachten wir folgende Aufgabenstellung:

Eingabe: Ein Array der Länge 10 mit positiven, ganzzahligen Werten.
Gesucht: Array-Elemente (und deren Positionen), die in der Summe mindestens 20 und möglichst genau 20 ergeben.

Bei näherem Nachdenken und wenn das Nachfüllen der Fächer mit berücksichtigt wird, ergeben sich außerdem die folgenden Fragen:

1. Wenn es zwei verschiedene Möglichkeiten gibt, die gleiche Summe zu erhalten, welche soll dann gewählt werden?
2. Wenn selbst bei Verwendung aller 10 Werte nur eine Summe kleiner 20 möglich ist, was ist dann zu tun?
3. Mit den aktuell vorhandenen Werten lässt sich 20 nicht exakt erreichen. Kann man mit den Ballons, die nachgefüllt werden, genau 20 erhalten?

Auf diese Fragen gute Antworten zu finden, stellt jeweils ein eigenes „Nebenproblem“ dar, das unabhängig vom Grundproblem betrachtet werden kann.

Grundproblem

Bei der Lösung des Grundproblems geht es darum, unter den vielen Möglichkeiten, die Array-Elemente miteinander zu kombinieren, die beste(n) auf geeignete Weise zu bestimmen. Dafür wollen wir zwei Strategien vorstellen, eine leicht zu findende und eine etwas komplexere.

Strategie 1: Brute Force Da es nur 10 Speicherfächer gibt, gibt es auch nur $2^{10} = 1024$ verschiedene Teilmengen von Fächern. Daher kann man ohne praktische Laufzeitprobleme für jede mögliche Teilmenge an Fächern die Anzahl an Ballons ausrechnen und dann die besten Möglichkeiten betrachten.

Strategie 2: Bottom-Up Idee: Wann lässt sich die Summe S durch Nutzung der ersten i Fächer erreichen?

- Wenn in Fach i genau S Ballons liegen.
- Wenn ich die Summe S durch Nutzung der ersten $i-1$ Fächer erhalten kann.
- Wenn ich die Summe $S - Fach[i]$ durch Nutzung der ersten $i-1$ Fächer erhalten kann.

Mit diesem Ansatz lässt sich eine 2-dimensionale Tabelle von Wahrheitswerten füllen: In dieser wird gespeichert, ob es mit den ersten i Fächern möglich ist, die Summe S zu erreichen.

Die Werte der Tabelle erhält man „Bottom-Up“, also indem man mit einfach zu berechnenden Werten beginnt und dann nach und nach weitere Werte aus schon bekannten Werten berechnet: Die Werte der ersten Spalte hängen von keiner anderen Spalte ab und werden zuerst berechnet. Die Werte der zweiten Spalte hängen nur von der ersten ab und können jetzt berechnet werden. Entsprechend wird für die anderen Spalten vorgegangen.

Falls man zu einem Zeitpunkt schon eine gültige Lösung gefunden hat, muss man alle schlechteren S nicht mehr betrachten.

Um am Ende die besten Möglichkeiten abzulesen, betrachtet man die Spalte für $i = 10$. Dort sucht man nach den Feldern $S \geq 20$, in denen `Wahr` steht. Die Fächer, die für diese Summe benötigt werden, kann man aus der Tabelle ablesen. Dazu muss man sich wiederholt fragen: *Warum wurde dieses Feld auf* `Wahr` *gesetzt?*

Nebenprobleme

Zur Lösung der Nebenprobleme lassen sich verschiedene Strategien überlegen. Die unten genannten Strategien liefern gute Ergebnisse, jedoch lassen sich immer Füllfolgen finden, für die sie nicht optimal funktionieren.

1. Wenn es zwei verschiedene Möglichkeiten gibt, die gleiche Summe zu erhalten, welche soll dann gewählt werden? Eine einfache Strategie ist es, die Kombination zu wählen, die die kleinere Anzahl an Fächern verwendet. Dadurch bleiben meist mehr Fächer mit kleineren Werten übrig. Mit diesen ist es später leichter, auf exakt 20 aufzufüllen. Mit größeren Werten schießt man leichter über die 20 hinaus.

2. Wenn selbst bei Verwendung aller 10 Elemente die Summe 20 nicht erreicht wird, was ist dann zu tun? Es empfiehlt sich, die kleinste Anzahl an Ballons auszukippen. Dadurch ist die Summe, die noch gebildet werden muss, am größten, und es gibt daher mehr Möglichkeiten, diese zu bilden.

3. Mit den aktuell vorhandenen Werten lässt sich 20 nicht exakt erreichen. Kann man mit den Ballons, die nachgefüllt werden, genau 20 erhalten? Gleiche Idee wie bei der zweiten Frage: Man nimmt die kleinste Anzahl an Ballons, die in der (im Moment) besten Summe vorkommt. Falls sich durch die nachgefüllten Ballons noch eine bessere Kombination ergibt, wird diese gewählt. Falls nicht, hat man trotzdem die bestmögliche Lösung.

Wer das in der Aufgabenstellung verlinkte Video zur LVM in der Sendung mit der Maus noch nicht gesehen hat, sollte das wirklich noch nachholen. Es ist zur Lösungsfindung nützlich und ruft Erinnerungen an einen jüngeren Ralph Caspers wach: `http://bit.ly/29PWZhx`

Implementierung

Bei der Implementierung gibt es keine besonderen Hürden. Die Datenstrukturen sind durch den gewählten Ansatz schon klar gegeben. Es sollte nur darauf geachtet werden, dass man nicht „schummelt“ und Kenntnisse über die Füllfolge mit in den Algorithmus einfließen lässt. Falls man eine objektorientierte Sprache verwendet, lässt sich das zum Beispiel realisieren, indem

man die LVM in eine eigene Klasse packt, und nur die aktuellen Füllstände nach außen sichtbar macht. Die Füllfolge lässt sich intern z.B. als Queue modellieren.

Beispiele

Die hier vorgestellte Lösung liefert für die sieben Beispiele insgesamt folgende Ergebnisse:

```
1:    (20) x 4             Verpackt gesamt:  80    Rest: 9
2:    (20) x 4             Verpackt gesamt:  80    Rest: 0
3:    (20) x24             Verpackt gesamt: 480    Rest: 11
4:    (20) x22             Verpackt gesamt: 440    Rest: 18
5:    (20) x 9 (30) x 8    Verpackt gesamt: 420    Rest: 15
6:    (20) x 9 (30) x 8    Verpackt gesamt: 420    Rest: 15
7:    (20) x 5             Verpackt gesamt: 100    Rest: 0
```

Nun zu den Beispielen im Einzelnen: Da sich die Beispiele 1 - 4 nur in der Länge der Füllfolge unterscheiden, sei hier nur für das erste die komplette Ausgabe abgedruckt. Auch Beispiele 5 und 6 testen den gleichen Aspekt und sind daher ebenfalls zusammengefasst.

Beispiel 1 - 4

Bei diesem Beispiel gibt es keine weiteren Besonderheiten; es gibt meist mehrere Varianten, exakt 20 zu erreichen. Daher greift an dieser Stelle noch keines der Nebenprobleme.

```
Bitte ID der Eingabe angeben:
1
[ 5,  3,  8,  3,  6,  2,  8,  4,  2,  2]
  Kippe  3 Ballons aus  2 aus =>  3 in der Schale.
[ 5,  9,  8,  3,  6,  2,  8,  4,  2,  2]
  Kippe  8 Ballons aus  3 aus => 11 in der Schale.
[ 5,  9,  1,  3,  6,  2,  8,  4,  2,  2]
  Kippe  9 Ballons aus  2 aus => 20 in der Schale.
[ 5,  3,  1,  3,  6,  2,  8,  4,  2,  2]
  Verpacke 20 Luftballons.
[ 5,  3,  1,  3,  6,  2,  8,  4,  2,  2]
  Kippe  1 Ballons aus  3 aus =>  1 in der Schale.
[ 5,  3, 11,  3,  6,  2,  8,  4,  2,  2]
  Kippe  3 Ballons aus  2 aus =>  4 in der Schale.
[ 5,  6, 11,  3,  6,  2,  8,  4,  2,  2]
  Kippe  5 Ballons aus  1 aus =>  9 in der Schale.
[ 4,  6, 11,  3,  6,  2,  8,  4,  2,  2]
  Kippe 11 Ballons aus  3 aus => 20 in der Schale.
[ 4,  6,  7,  3,  6,  2,  8,  4,  2,  2]
  Verpacke 20 Luftballons.
[ 4,  6,  7,  3,  6,  2,  8,  4,  2,  2]
  Kippe  3 Ballons aus  4 aus =>  3 in der Schale.
[ 4,  6,  7,  5,  6,  2,  8,  4,  2,  2]
  Kippe  4 Ballons aus  1 aus =>  7 in der Schale.
[ 0,  6,  7,  5,  6,  2,  8,  4,  2,  2]
  Kippe  6 Ballons aus  2 aus => 13 in der Schale.
[ 0,  0,  7,  5,  6,  2,  8,  4,  2,  2]
  Kippe  7 Ballons aus  3 aus => 20 in der Schale.
```

```
[ 0,  0,  0,  5,  6,  2,  8,  4,  2,  2]
  Verpacke 20 Luftballons.
[ 0,  0,  0,  5,  6,  2,  8,  4,  2,  2]
  Kippe  2 Ballons aus  6 aus =>  2 in der Schale.
[ 0,  0,  0,  5,  6,  0,  8,  4,  2,  2]
  Kippe  4 Ballons aus  8 aus =>  6 in der Schale.
[ 0,  0,  0,  5,  6,  0,  8,  0,  2,  2]
  Kippe  6 Ballons aus  5 aus => 12 in der Schale.
[ 0,  0,  0,  5,  0,  0,  8,  0,  2,  2]
  Kippe  8 Ballons aus  7 aus => 20 in der Schale.
[ 0,  0,  0,  5,  0,  0,  0,  0,  2,  2]
  Verpacke 20 Luftballons.
```

Beispiele 5 und 6

Diese Beispiele testen die Behandlung der Nebenprobleme 1 und 3. Hier sollte das Programm zuerst möglichst viele 20er Beutel produzieren und anschließend 30er. Die Reihenfolge der 5er- und 15er-Pakete in der Füllfolge (in 5 und 6 unterschiedlich) soll bei einer schlechten Lösung der Nebenprobleme dazu führen, dass weniger 20er Beutel als möglich produziert werden. Bei Beispiel 5 könnte ein schlechterer Algorithmus am Anfang einen Beutel mit vier mal 5 Ballons packen. Dadurch währen drei perfekte Beutel weniger möglich.

```
Bitte ID der Eingabe angeben:
5
[ 5,  5,  5,  5, 15, 15, 15, 15, 15, 15]
  Kippe  5 Ballons aus  1 aus =>  5 in der Schale.
[15,  5,  5,  5, 15, 15, 15, 15, 15, 15]
  Kippe 15 Ballons aus  1 aus => 20 in der Schale.
[15,  5,  5,  5, 15, 15, 15, 15, 15, 15]
  Verpacke 20 Luftballons.
[7 mal 5 + 15 = 20]
[15, 15, 15,  5, 15, 15, 15, 15, 15, 15]
  Kippe  5 Ballons aus  4 aus =>  5 in der Schale.
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
  Kippe 15 Ballons aus  1 aus => 20 in der Schale.
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
  Verpacke 20 Luftballons.
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
  Kippe 15 Ballons aus  1 aus => 15 in der Schale.
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
  Kippe 15 Ballons aus  1 aus => 30 in der Schale.
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
  Verpacke 30 Luftballons.
[6 mal 15 + 15 = 30]
[ 0,  0,  0,  0,  0,  0,  0, 15, 15, 15]
  Kippe 15 Ballons aus  8 aus => 15 in der Schale.
[ 0,  0,  0,  0,  0,  0,  0,  0, 15, 15]
  Kippe 15 Ballons aus  9 aus => 30 in der Schale.
[ 0,  0,  0,  0,  0,  0,  0,  0,  0, 15]
  Verpacke 30 Luftballons.
```

Bei Beispiel 6 werden Beutel der gleichen Größe und in der gleichen Reihenfolge wie bei 5 produziert.

Beispiel 7

Dieses Beispiel testet Nebenproblem 2.

```
Bitte ID der Eingabe angeben:
7
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1]
  Kippe  1 Ballons aus  1 aus =>  1 in der Schale.
...
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1]
  Kippe  1 Ballons aus  1 aus => 20 in der Schale.
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1]
  Verpacke 20 Luftballons.
...
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
  Verpacke 20 Luftballons.
```

Laufzeit

Eine Laufzeitbetrachtung ist eigentlich unnötig, da nur eine LVM mit 10 Speicherfächern betrachtet wird. Bei größeren LVMs würden die Unterschiede zwischen den Lösungsansätzen zum Grundproblem jedoch deutlich werden. Daher soll an dieser Stelle eine LVM mit n Speicherfächern betrachtet werden.

Brute Force Bei n Fächern gibt es 2^n verschiedene Teilmengen, die untersucht werden müssen. Für $n = 10$ sind das nur 1024, für $n = 20$ schon 1.048.576.

Bottom-Up Hier hängt die Laufzeit von der Anzahl der Elemente in der Menge der möglichen Kombinationen aus Fächern und Ballon-Zahl ab. Die betrachteten Ballon-Zahlen lassen sich nach oben beschränken: In der hier betrachteten Implementierung werden Speicherfächer aufsummiert, bis 20 überschritten wird. Dadurch liegt die Grenze hier recht niedrig, in allen betrachteten Beispielen nie über 30. Für $n = 10$ ergeben sich $10 \cdot 30 = 300$, für $n = 20$ nur $20 \cdot 30 = 600$ betrachtete Zellen.

Man erkennt, dass der zweite Algorithmus auch für größere LVMs noch schnell arbeitet, während Brute Force deutlich länger braucht. In der Landau-Notation: Brute Force wächst mit $\mathcal{O}(2^n)$, Bottom Up mit (Pseudo-) $\mathcal{O}(n)$.

Quellcode

Der folgende Quellcode zeigt den Inhalt der Main-Methode eines C#-Programmes.
Die Berechnung der möglichen Kombinationen benötigt nur 3 Zeilen, der ganze Rest wird von der Betrachtung der Nebenfragen eingenommen.

```
//Einlesen und initialisieren der LVM
Console.WriteLine("Bitte ID der Eingabe angeben:");
string inputPath = @"luftballons#.txt".Replace("#", Console.ReadLine());
LVM lvm = new LVM(new Queue<int>(Array.ConvertAll(File.ReadAllLines(inputPath), int.Parse)));
int goal = 20;

for(int i = 0; i<10; lvm.fach(i++)) { } //Anfängliches Füllen aller Fächer
//Solange noch die Chance besteht, einen Beutel zu füllen:
while(lvm.speicherfaecher.Sum()>=goal||!lvm.speicherfaecher.Contains(0)) {
    bool[,] possible = new bool[10, lvm.speicherfaecher.Aggregate((a, b) => a>=goal ? a : a+b)+1];
    if(possible.GetLength(1)-goal<=0) { //Falls mit den aktuellen Mengen das Ziel nicht erreicht wird
      //Fach mit dem kleinsten Inhalt auskippen
        goal-=lvm.speicherfaecher.Min();
        lvm.fach(Array.IndexOf(lvm.speicherfaecher, lvm.speicherfaecher.Min()));
    } else {
        //Berechnen des Hauptproblems
        for(int i = 0; i<10; i++)
            for(int j = 0; j<possible.GetLength(1); j++)
                possible[i, j]=lvm[i]==j||i>0&&(possible[i-1, j]||j-lvm[i]>=0&&possible[i-1, j-lvm[i]]);
        //Prüfen, was die bestmögliche Anzahl ist:
        int found = Enumerable.Range(goal, possible.GetLength(1)-goal).FirstOrDefault(a => possible[9, a]);
        //Benutzte Fächer rekonstruieren
        List<int> used = new List<int>();
        for(int i = 9; i>=0; i--) {
            if(lvm[i]==found) { used.Add(i); break; }
            if(found-lvm[i]>=0&&possible[i-1, found-lvm[i]]&&(i==0||!possible[i-1, found])) { used.Add(i);
                found-=lvm[i]; }
        }
        //Fach mit dem kleinsten Inhalt auskippen
        goal-=used.Select(i => lvm.speicherfaecher[i]).Min();
        lvm.fach(Array.IndexOf(lvm.speicherfaecher, used.Select(i => lvm.speicherfaecher[i]).Min()));
    }
    if(goal<=0) { //Falls Beutel voll, Ballons verpacken
        goal=20;
        lvm.verpacken();
    }
}
```

Bewertungskriterien

- Die Laufzeit des Algorithmus ist nicht wichtig: Auch eine Brute-Force-Lösung, die systematisch alle Fächer-Kombinationen durchgeht, ist bei dieser Aufgabe in Ordnung.
- Die Mindestanforderung an eine Verpackungsstrategie: Auf keinen Fall dürfen Verpackungen mit weniger als 20 Luftballons entstehen. Akzeptabel ist aber, wenn die am Schluss übrig gebliebenen Luftballons noch verpackt werden und so eine einzige Verpackung mit weniger als 20 Luftballons entstehen kann.
- Die Nebenprobleme sollten (ggf. implizit) erkannt und begründet behandelt werden. Das ist insbesondere nicht der Fall, wenn keine besondere Strategie eingesetzt, sondern die erstbeste passende Fächerkombination genutzt wird. Die gewählten Strategien sollten zu brauchbaren Ergebnissen führen.
- Natürlich muss auch das Grundproblem gut gelöst werden. Allzu einfache Verfahren zur Bestimmung von Fächerkombinationen können leicht gute Kombinationen auslassen.
- Ein Detail: Es ist in Ordnung, wenn leere Fächer nicht weiter berücksichtigt werden, also davon ausgegangen wird, dass die Füllfolge keine 0 enthält und Fächer nur leer werden können, wenn die Füllfolge zu Ende ist. Es ist andererseits auch in Ordnung, wenn leere Fächer als Fächer mit 0 Luftballons in die Berechnungen mit einbezogen werden (das vereinfacht die Implementierung minimal), solange das nicht zu schlechteren Ergebnissen führt.
- In der Dokumentation müssen die Lösungen für zumindest einige der vorgegebenen Beispiele enthalten sein. Wenn die implementierte Strategie in manchen Fällen erkennbar nicht ideal ist, ist das nicht schlimm. In einer prerfekten Bearbeitung würden solche Nachteile angesprochen; das zeugt von einem besonders gutem Verständnis des Problems.

J2 LAMA

Bei dieser Aufgabe geht es nicht darum, sich einen komplizierten Algorithmus auszudenken, sondern die richtigen Werkzeuge zu wählen. Daher soll hier ein kurzer und einfacher Weg vorgestellt werden, ein funktionierendes und leicht erweiterbares Programm zu erhalten.

Zielsetzung und Plan

Warum das Rad neu erfinden, wenn alle benötigten Werkzeuge schon zur Verfügung stehen?

Diese Lösung soll insbesondere Folgendes demonstrieren:

- Die in eigentlich allen üblichen Programmiersprachen vorhandenen Standard-Elemente für grafische Benutzungsschnittstellen (engl.: graphical user interfaces, kurz GUI) können schon alles, was man für diese Aufgabe braucht.
- Ein gut strukturiertes Programm macht es einfach, Regeln für weitere Spielvarianten hinzuzufügen.

Die Lösung in dieser Aufgabe wird in C# 6.0 mit Windows Forms umgesetzt. Das Ganze funktioniert auch in jeder anderen Sprache mit entsprechenden GUI-Elementen. Natürlich lässt sich ein LAMA auch als Konsolenprogramm realisieren, bei denen etwa die Tastenzustände mit Textzeichen dargestellt werden und die zu drückende Taste per Angabe der Tastenposition im LED-Raster ausgewählt wird. Aber netter ist eine grafische Schnittstelle schon.

Um die Tasten bzw. Buttons eines LAMA darzustellen, nehmen wir - Überraschung! - Buttons. Diese bieten alles, was wir brauchen:

- Sie können ihre Farbe ändern (Licht an/aus).
- Es lässt sich im Programm leicht darauf reagieren, wenn ein Button gedrückt wird (LEDs umschalten).
- Sie können zur Laufzeit des Programms erzeugt werden (verschiedene Feldgrößen).

Umsetzung

Abbildung 1 zeigt ein Programmfenster für ein LAMA mit 16×16 Tasten. Oben links sieht man die Eingabemöglichkeiten für Feldgröße und Spielmodus. Mit diesen ist keine weitere Programm-Logik verbunden, und ihre Realisierung hängt nur von der gewählten UI-Bibliothek ab.

Für die Knöpfe brauchen wir zwei Farben für An und Aus:

```
Color OnColor = Color.Yellow;
Color OffColor = Color.Black;
```

Um später auf die Buttons einfach zugreifen zu können, speichern wir sie in einem 2D-Array:

```
Button[,] grid;
```

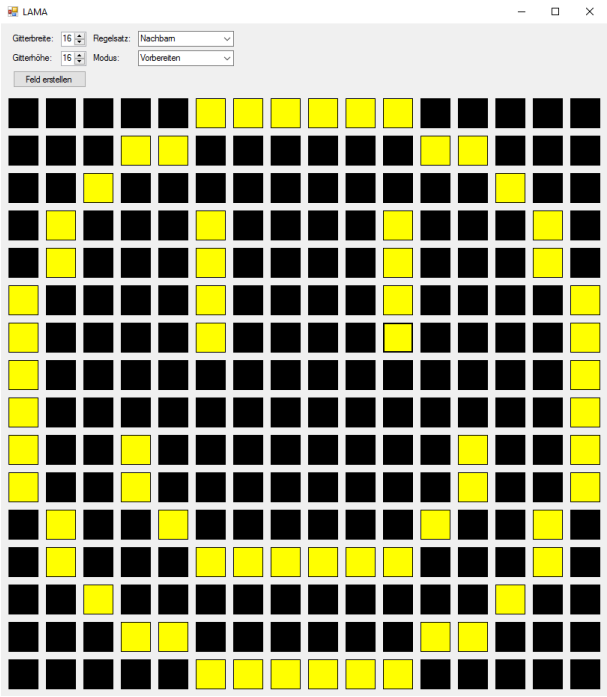


Abbildung 1: Fertiges Fenster mit einem 16×16-LAMA

Des weiteren müssen wir noch unterscheiden, ob das Spielfeld gerade vorbereitet oder ob schon gespielt wird. Wenn gespielt wird, welche Regeln werden dann genutzt? Die Auswahlmöglichkeiten realisieren wir als Aufzählungstypen (Enums), die wir als Datenquelle für die Auswahl des Regelsatzes im UI und auch später im Programm verwenden.

```
enum RuleSet { Nachbarn, ObenLinks }
enum Mode { Vorbereiten, Spielen }
```

Damit haben wir jetzt alles, um die Buttons zu erzeugen:

```
private void createGrid_Click(object sender, EventArgs e) {
  //Alte Buttons löschen
  Controls.OfType<Button>().Where(a => a !=
      sender).ToList().ForEach(Controls.Remove);
  sizeX = (int)sizeXSelect.Value; //Numeric UpDown für Größe in x-Richtung
  sizeY = (int)sizeYSelect.Value; //Numeric UpDown für Größe in y-Richtung
  grid = new Button[sizeX, sizeY];
  //Für jede Position im Gitter
  for(int x = 0; x < sizeX; x++) {
    for(int y = 0; y < sizeY; y++) {
      //Neuen Button erzeugen und konfigurieren
      Button b = new Button();
      b.Tag = new int[] { x, y }; //Tag = Position im Gitter
      b.Location = new Point(10 + x * 50, 100 + y * 50);
      b.Size = new Size(40, 40);
      b.Click += ButtonClicked;
      b.BackColor = OffColor;
      b.FlatStyle = FlatStyle.Flat;
      Controls.Add(b);
      grid[x, y] = b;
    }
  }
  //Fenstergröße anpassen
```

```
  Size = new Size(Math.Max(sizeX * 50 + 25, 339), sizeY * 50 + 138);
}
```

Jetzt müssen wir noch darauf reagieren können, wenn auf eine Taste (im Programm also auf einen Button) gedrückt wird. Dazu haben alle Buttons zu dem Ereignis Button.Click den Event-Handler ButtonClicked zugewiesen bekommen. Um feststellen zu können, welcher Knopf denn nun gedrückt worden ist, nutzen wir den Tag des Buttons: Dort haben wir beim Erstellen die Position im Gitter gespeichert.

Wenn das Spiel später gespielt wird, muss oft der Zustand einer Taste umgekehrt werden. Dies lagern wir in eine kleine Hilfsfunktion aus, die den richtigen Button ebenfalls über die Koordinaten x und y seiner Gitterposition identifiziert.

```
private void toggle(int x, int y) =>
    grid[x, y].BackColor = grid[x, y].BackColor == OffColor ? OnColor :
       OffColor;
```

Damit lässt sich jetzt der EventHandler implementieren:

```
private void ButtonClicked(object sender, EventArgs e) {
  int buttonX = ((int[])((Button)sender).Tag)[0];
  int buttonY = ((int[])((Button)sender).Tag)[1];
  toggle(btnX, btnY);
  if((Mode)mode.SelectedIndex == Mode.Vorbereiten) return;
  switch((RuleSet)ruleset.SelectedIndex) {
    case RuleSet.Nachbarn:
      //Die Klassenvariable "offsets" ist so definiert:
      //int[,] offsets = { { -1, 0 }, { 0, 1 }, { 1, 0 }, { 0, -1 } };
      for(int direction = 0; direction < 4; direction++) {
        int x = offsets[direction, 0] + buttonX;
        int y = offsets[direction, 1] + buttonY;
        if(0 <= x && x < sizeX && 0 <= y && y < sizeY)
          toggle(x, y);
      }
      break;
    case RuleSet.ObenLinks:
      for(int x = 0; x <= buttonX; x++)
        for(int y = 0; y <= buttonY; y++)
          if(x != buttonX || y != buttonY)
            toggle(x, y);
      break;
  }
  //Spiel gewonnen, wenn keine Lampe an.
  if(grid.Cast<Button>().Where(b => b.BackColor == OnColor).Count() == 0)
    MessageBox.Show("Gewonnen!");
}
```

Hier sieht man, wie einfach es ist, eine Spielvariante hinzuzufügen: Im Switch-Statement implementiert man die gewünschte Reaktion auf das Drücken einer Taste als weiteren Fall.

Am Ende der Funktion wird geprüft, ob das Spiel gewonnen wurde, nämlich genau dann, wenn kein Licht mehr leuchtet.

Bewertungskriterien

- Die Realisierung des LAMA-Spiels, egal ob grafisch oder in der Konsole, sollte die Anforderungen der Aufgabenstellung erfüllen:
  - Ein Anfangszustand soll festgelegt werden. Das kann sehr unterschiedlich realisiert sein. Entscheidend ist, dass die Zahl der möglichen Anfangszustände zumindest potenziell nicht zu klein ist. Eine zufällige Wahl des Anfangszustands ist in Ordnung, solange der (zugegeben unwahrscheinliche) Fall vermieden wird, dass alle Leuchten aus sind — eine solche Spielrunde wäre doch allzu schnell zu Ende. Ein einziger, im Programm fest codierter Zustand ist zu wenig. Das Einlesen des Anfangszustandes aus einer Datei ist in Ordnung, auch wenn nur sehr wenige Dateien als Beispiele beiliegen. Auch eine Eingabe durch den Benutzer ist eine akzeptable Variante.
  - Umschalten von Feldern durch Klicken ist gut, durch Eingabe von Koordinaten in der Konsole akzeptabel. Einlesen der Züge aus einer Text-Datei reicht nicht.
  - Der Zustand der LEDs muss erkennbar angezeigt werden: Eine grafische Darstellung ist nett, eine gute ASCII-Darstellung in der Konsole genügt aber auch.
- Das Spiel sollte, einschließlich der Regeln der jeweiligen Spielvariante, korrekt realisiert sein. Das Umschalten der Leuchten muss regelgerecht funktionieren, und das Ende des Spiels muss erkannt werden.
- Für Teilaufgabe 2 muss die Spielfeldgröße (in Grenzen) flexibel wählbar sein.
- Die in Teilaufgabe 3 beschriebene Spielvariante muss realisiert sein.
- Anhand von Beispielen sollte erkennbar werden, dass die Wahl der gedrückten Taste und insbesondere das Umschalten funktioniert. Es reicht aus, pro Spielvariante einige wenige Spielschritte darzustellen. Eine rein textuelle Darstellung der Spielschritte genügt.

Aufgabe 1: Sprichwort

1.1 Lösungsidee

Diese Aufgabe ist eine echte Recherche- und Fleißaufgabe. Es geht um vier Daten, nämlich westliches und östliches Ostern sowie westliches und östliches Weihnachten[1], die in zwei verschiedenen Kalendern (gregorianischer Kalender und julianischer Kalender) gesehen werden.

Umrechnung zwischen den Kalendern

Zuerst soll es um die Unterschiede zwischen dem gregorianischen und dem julianischen Kalender gehen. Seit der Einführung des gregorianischen Kalenders im 16. Jahrhundert läuft es auseinander. Der Unterschied ist, dass es im gregorianischen Kalender in vier Jahrhunderten drei Schalttage weniger als im julianischen Kalender gibt. So war das Jahr 1900 ein julianisches Schaltjahr, aber kein gregorianisches Schaltjahr.

Der Unterschied zwischen den Kalendern in Tagen kann wie folgt berechnet werden (dies gilt für jedes Jahr, für Jahre mit ausfallendem Schaltjahr nur ab März):

$$\Delta_{\text{Kalender}} = (jahr \div 100) - (jahr \div 400) - 2$$

Für das Jahr 2016 ergibt sich mit der Formel ein Unterschied von 13 Tagen. Auch laut `https://de.wikipedia.org/wiki/Julianischer_Kalender`[2] ist zwischen den beiden Kalendersystemen momentan einen Unterschied von 13 Tagen: Wenn nach julianischem Kalender der 25.12.2016 (Weihnachten) ist, dann ist nach gregorianischem Kalender schon der 07.01.2017.

Eine kleine Überschlagsrechnung zeigt, mit welchen Ergebnissen wir bei der Aufgabe ungefährt zu rechnen haben: Der Abstand zwischen den Kalendersystemen wird sich immer weiter erhöhen, so dass in einigen Jahren der 25.12. eines (julianischen) Jahres auf das (nach gregorianischem Kalender festgelegte) westliche Ostern fällt. Vom 7. Januar bis zum Frühlingsanfang (als frühestes Datum für Ostern) fehlen noch circa 75 Tage, um die der Unterschied zwischen den Kalendern sich erhöhen muss. Damit müssen noch ca. $75 \times 400/3 = 10000$ Jahre vergehen, bis also der julianische 25.12. auf das westliche Ostern des nachfolgenden Jahres trifft.

Für die andere Fragestellung (wann das östliche Ostern auf das westliche Weihnachten fällt), müssen wir ein bischen weiter schauen. Überschlagsmäßig fehlen noch circa 8 Monate, also circa 240 Tage. Das macht dann ca. $240 \times 400/3 = 32000$ Jahre. Die Rechnungen zeigen, dass Christian dies vermutlich nicht mehr in diesem Leben erleben wird.

[1] In der Aufgabenstellung heißt es *katholisch und protestantisch* statt westlich beziehungsweise *orthodox* statt östlich. Hier wird die etwas kürzere Schreibweise genutzt.

[2] Abgerufen am 5. Oktober 2016 (gregorianischer Kalender).

Genauere Rechnungen

Für eine genauere Rechnung kann die Aufgabe auf zwei Berechnungen zurückgeführt werden: Die Umrechnung von Daten zwischen den beiden Kalendersystemen (siehe oben) und die Bestimmung des westlichen und des östlichen Osterdatums. Für das Osterdatum ist die Gaußsche Osterformel (siehe `https://de.wikipedia.org/wiki/Gaußsche_Osterformel`) die erste Anlaufstelle.

Leider hat die Gaußsche Osterformel in der klassischen Form ein paar Ausnahmeregeln. Dafür gibt es die ergänzte Osterformel nach Kinkelin und Lichtenberg (ebenda). Diese kann für ein beliebiges Jahr berechnen, wann das westliche Ostern (mit Ergebnis als gregorianisches Datum) und wann das östliche Ostern (mit Ergebnis als julianisches Datum) in jedem zukünftigen Jahr ist. Eine formale Darstellung der Formel ist in der Tabelle 1 zu sehen. Dabei ist *jahr* das Jahr, für das die Berechnung durchgeführt werden soll. Die Divisionen in der Formel sind ganzzahlig durchzuführen. Die Formel gibt Ostern im gregorianischen und julianischen Kalender wieder. Für die Unterscheidung zwischen den Kalendern muss an zwei Stellen in der Berechnungsvorschrift unterschieden werden. Das Ergebnis *os* sagt, am wievielten März im jeweiligen Kalender Ostern ist. Dabei kann beim Monatsüberlauf einfach weiter gezählt werden: Aus dem 32. März wird der 1. April und so weiter.

Wert	**westlich**	**östlich**
Säkularzahl k	$jahr \div 100$	
säkulare Mondschaltung m	$15 + ((3 \times k + 3) \div 4) - ((8 \times k + 13) \div 25)$	15
säkulare Sonnenschaltung s	$2 - ((3 \times k + 3) \div 4)$	0
Mondparameter a	$jahr \mod 19$	
Keim für den ersten Vollmond im Frühling d	$(19 \times a + m) \mod 30$	
Korrekturgröße r	$(d + (a \div 11)) \div 29$	
Ostergrenze og	$21 + d - r$	
erster Sonntag im März sz	$7 - (jahr + (jahr \div 4) + s) \mod 7$	
Osterentfernung in Tagen oe	$7 - (og - sz) \mod 7$	
Ostersonntag os	$og + oe$	

Tabelle 1: Berechnungsvorschrift für Ostern. Das westliche Osterdatum ist ein gregorianisches Datum, das östliche Osterdatum ist ein julianisches Datum.

Um nun zu wissen, wann das östliche Ostern im gregorianischen Kalender stattfindet, wird zuerst das östliche Ostern im julianischen Kalender berechnet. Dieses Datum muss dann wieder in den gregorianischen Kalender umgerechnet werden, wie schon im vorigen Abschnitt gezeigt.

Wann ist Weihnachten gleich Ostern?

Wie oben schon gesagt, läuft der julianische Kalender dem gregorianischen Kalender mit der Zeit immer weiter voraus. So ist einerseits der julianische 25.12. momentan schon im gregorianischen Januar des darauf folgenden Jahres. Und das orthodoxe Ostern liegt in vielen Fällen auch nach dem westlichen Osterdatum. Da der Abstand mit der Zeit immer weiter zunimmt,

wird zuerst das östliche (julianische) Weihnachten in den Bereich des westlichen (gregorianischen) Ostern kommen (Abstand momentan 2-3 Monate). Später wird auch das östliche Ostern in den Bereich des westlichen Weihnachten kommen (Abstand momentan 8-9 Monate). Eine grobe Überschlagsrechnung kann (wie oben) berechnen, wann der Unterschied ungefähr ein Jahr ausmachen wird. Dies soll als Enddatum für die Suche nach übereinstimmenden Daten genommen werden. Der Unterschied zwischen den Kalendersystemen erhöht sich in vierhundert Jahren um drei Tage. Rechnen wir mit 366 zu erreichenden Tagen Unterschied, benötigen wir also $\frac{366*400}{3} = 48800$ Jahre. In der Zeit sollte es hoffentlich ein gregorianisches Ostern geben, das auf den julianischen 25.12. fällt, und ein julianisches Ostern, das auf den gregorianischen 25.12. fällt. Es sollte also reichen, in einem Programm alle Jahre von jetzt bis zum Jahr 50000 zu prüfen.

Für die Frage, wann das julianische Weihnachten auf das gregorianische Ostern fällt, wird für jedes Jahr der julianische 25.12. in ein gregorianisches Datum umgerechnet. Von dem gregorianischen Datum (es fällt ja in das darauf folgende gregorianische Jahr!) wird geschaut, wann in dem Jahr das westliche Osterdatum ist. Wenn Tag und Monat übereinstimmen, dann ist das Datum für die erste Frage gefunden.

Für die Frage, wann das gregorianische Weihnachten auf das julianische Ostern fällt, wird für jedes Jahr das orthodoxe Ostern als julianisches Datum berechnet und in ein gregorianisches Datum umgerechnet. Wenn dies der 25.12. ist, dann ist auch die zweite Frage geklärt.

Ergebnis

Für Teil 1 ist **Sonntag, der 23.03.11919** das erste gregorianische Ostern, das auf einen julianischen 25.12. fällt. Für Teil 2 ist **Sonntag, der 25.12.32839** das erste gregorianische Weihnachten, das auf einen julianischen Ostersonntag fällt. Christian wird den Fall also wohl nicht mehr erleben.

Eine Gegenrechnung zu den obigen Abschätzungen zeigt, dass bis zum Jahr 11919 noch $11919 - 2016 = 9903$ Jahre sind. In der Zeit wird der Kalender um $(9903 \times 3) \div 400 = 74$ Tage weiter abweichen. Dies sind ungefähr $2,5$ Monate und fast die geschätzen 75 Tage. Für das zweite Datum errechnet sich ein Unterschied von $32839 - 2016 = 30823$ Jahren. Dies ergibt eine Differenz von $(30823 \times 3) \div 400 = 231$ Tagen. Auch dieses Ergebnis stimmt mit der obigen Schätzung ungefähr überein.

Alternative Osterformeln

Es gibt auch alternative Osterformeln. Einmal gibt es die originale Gaußsche Osterformel, bei der aber nachträglich ein paar Sonderfälle ergänzt worden sind. Wenn diese Formel benutzt wird, sollten die korrekten Sonderfälle beachtet werden (diese sind in die obige Formel schon komplett integriert worden).

Außerdem gibt es noch die Meeus/Jones/Butcher-Algorithmen[3]). Diese Formeln sind etwas anders strukturiert, ergeben aber für den für diese Aufgabe relevanten Zeitraum die gleichen Osterdaten (getestet für die Jahre von 1900 bis 50000, sowohl für das westliche als auch das östliche Ostern).

[3] S. z.B. `https://en.wikipedia.org/wiki/Computus#Anonymous_Gregorian_algorithm`

1.2 Ergänzende Informationen

Oster-Ei-nigkeit

Es gibt schon seit einigen Jahren Gespräche, in denen versucht wird, sich auf ein gemeinsames Osterdatum zu einigen. Gerade im Juni 2016 gab es ein panorthodoxes Konzil (das erste seit 344 bzw. 1229 Jahren – je nach Sichtweise) auf Kreta, auf dem dieses Thema wieder angegangen wurde. Es gibt aber bisher noch kein Ergebnis.

Alle möglichen Daten für Weihnachten am 25.12.

An folgenden Tagen (gregorianisch) fällt das westliche Ostern auf das östliche Weihnachten (25.12. des Vorjahres nach julianischem Kalender):

11919-03-23	11930-03-23	12014-03-23	12025-03-23	12109-03-24
12272-03-24	12356-03-25	12367-03-26	12378-03-26	12451-03-26
12462-03-26	12535-03-27	12546-03-27	12557-03-27	12630-03-28
12641-03-28	12714-03-29	12720-03-28	12725-03-29	12804-03-28
12809-03-29	12815-03-29	12888-03-28	12899-03-29	12910-03-30
12972-03-29	12983-03-30	12994-03-30	13056-03-30	13067-03-31
13078-03-31	13089-03-31	13151-04-01	13162-04-01	13173-04-01
13246-04-01	13252-03-31	13257-04-01	13330-04-02	13336-04-01
13341-04-02	13420-04-02	13425-04-03	13431-04-03	13504-04-03
13515-04-04	13526-04-04	13588-04-03	13599-04-04	13610-04-04
13672-04-03	13683-04-04	13694-04-04	13705-04-05	13767-04-05
13778-04-05	13789-04-05	13851-04-06	13862-04-06	13868-04-05
13873-04-06	13946-04-07	13952-04-06	13957-04-07	14036-04-06
14041-04-07	14120-04-07	14131-04-08	14204-04-08	14215-04-09
14226-04-09	14288-04-08	14299-04-09	14310-04-10	14321-04-10
14383-04-10	14394-04-10	14405-04-10	14467-04-10	14478-04-10
14489-04-10	14562-04-11	14568-04-10	14573-04-11	14652-04-11
14657-04-12	14736-04-12	14747-04-13	14820-04-12	14831-04-13
14842-04-13	14904-04-13	14915-04-14	14926-04-14	14999-04-14
15010-04-15	15021-04-15	15083-04-15	15094-04-15	15105-04-16
15178-04-16	15184-04-15	15189-04-16	15262-04-16	15268-04-15
15273-04-16	15352-04-16	15357-04-17	15363-04-17	15436-04-17
15447-04-18	15458-04-18	15520-04-18	15531-04-19	15542-04-19
15604-04-18	15615-04-19	15626-04-19	15637-04-19	15699-04-19
15710-04-20	15721-04-20	15794-04-20	15805-04-21	15884-04-20
15889-04-21	15968-04-21	16052-04-21	16063-04-22	16136-04-22
16147-04-23	16158-04-23	16242-04-24	16253-04-24	16500-04-25

An folgenden (gregorianischen) ersten Weihnachtstagen wird das östliche Ostern nach julianischem Kalender des gleichen Jahres gefeiert:

32839-12-25	32907-12-25	32991-12-25	33059-12-25	33127-12-25
33211-12-25	33222-12-25	33295-12-25	33363-12-25	33374-12-25
33431-12-25	33442-12-25	33504-12-25	33510-12-25	33583-12-25
33594-12-25	33605-12-25	33667-12-25	33678-12-25	33689-12-25
33735-12-25	33746-12-25	33757-12-25	33803-12-25	33814-12-25
33825-12-25	33887-12-25	33898-12-25	33955-12-25	33966-12-25
33977-12-25	33988-12-25	34039-12-25	34050-12-25	34061-12-25
34072-12-25	34107-12-25	34118-12-25	34129-12-25	34140-12-25
34208-12-25	34259-12-25	34270-12-25	34281-12-25	34292-12-25
34327-12-25	34338-12-25	34349-12-25	34360-12-25	34422-12-25
34433-12-25	34444-12-25	34501-12-25	34512-12-25	34585-12-25
34596-12-25	34642-12-25	34653-12-25	34664-12-25	34710-12-25
34721-12-25	34732-12-25	34805-12-25	34816-12-25	34895-12-25
34963-12-25	34968-12-25	35025-12-25	35031-12-25	35036-12-25
35104-12-25	35183-12-25	35188-12-25	35267-12-25	35278-12-25
35335-12-25	35340-12-25	35346-12-25	35403-12-25	35408-12-25
35414-12-25	35487-12-25	35498-12-25	35555-12-25	35566-12-25
35639-12-25	35650-12-25	35661-12-25	35707-12-25	35718-12-25
35729-12-25	35791-12-25	35859-12-25	35870-12-25	35881-12-25
35927-12-25	35938-12-25	35949-12-25	36011-12-25	36022-12-25
36033-12-25	36044-12-25	36101-12-25	36112-12-25	36163-12-25
36174-12-25	36185-12-25	36196-12-25	36231-12-25	36242-12-25
36253-12-25	36264-12-25	36310-12-25	36321-12-25	36332-12-25
36394-12-25	36405-12-25	36416-12-25	36489-12-25	36546-12-25
36557-12-25	36568-12-25	36614-12-25	36625-12-25	36636-12-25
36704-12-25	36766-12-25	36777-12-25	36788-12-25	36861-12-25
36872-12-25	36929-12-25	36940-12-25	37008-12-25	37092-12-25
37149-12-25	37160-12-25	37244-12-25	37312-12-25	

Wann ist Weihnachten?

Obwohl der 1. Weihnachtstag (25.12.) in der Aufgabenstellung explizit erwähnt wird, wollen wir andere Vorstellungen vom Datum des Weihnachtsfests tolerieren. In der folgenden Tabelle ist angegeben, wie die Ergebnisse ausfallen, wenn Weihnachten auf den 24.12., 25.12. oder gar 26.12. terminiert wird. In den Abkürzungen steht ö für östlich, w für westlich, O für Ostern und W für Weihnachten.

Termin	Fall	Gregorianisch	Julianisch
24.12.	w.O. = ö.W.	22.03.11987	24.12.11986
	ö.O. = w.W.	24.12.32744	24.04.32744
25.12.	w.O. = ö.W.	23.03.11919	25.12.11918
	ö.O. = w.W.	25.12.32839	25.04.32839
26.12.	w.O. = ö.W.	22.03.11767	26.12.11766
	ö.O. = w.W.	26.12.32934	25.04.32934

1.3 Bewertungskriterien

- Die in der Aufgabenstellung geforderte Recherche ist beschrieben; insbesondere wird die Quelle für die Berechung des Osterdatums genannt.
- Es wird (als Schlussfolgerung auf Grundlage der Recherche) begründet, wieso es positive Antworten auf die in der Aufgabe gestellten Fragen gibt.
- Die Bestimmung des Osterdatums ist korrekt. Die Verwendung der Gaußschen Osterformel ohne Beachtung der Sonderfälle wird akzeptiert, da die Antworten auf die Fragen der Aufgabenstellung davon nicht beeinflusst werden. Bei der Berechnung des Osterdatums wird bei allen bekannten Formeln die ganzzahlige Division genutzt. Wenn stattdessen mit Fließkommazahlen gerechnet wird, kann es zu falschen Ergebnissen beim Osterdatum kommen. Dies gilt als falsche Berechnung des Osterdatums.
- Die Umrechnung zwischen den Kalendersystemen ist korrekt.
- Die beiden Tage, an denen Ostern und Weihnachten zusammenfallen, werden angegeben und sind korrekt. Hierzu gibt es zwei besondere Fälle:
  - Wenn jemand bei Teilaufgabe 1 nicht mit dem 25.12. als östlichem Weihnachtsdatum gerechnet hat, sondern den 7.1. (aus der Aufgabenstellung) genommen hat, dann kommt bei der Teilaufgabe 1 heraus, dass am 24.03.10363 das erste westliche Ostern ist, an dem gleichzeitig ein julianischer 7.1. ist. Wenn aus der Einsendung klar wird, dass hier der 7.1. genommen wurde, aber sonst die Berechnungen korrekt sind, ist das akzeptabel.
  - In den Datumsbibliotheken verschiedener Sprachen ist es nicht möglich, Daten größer als das Jahr 9999 zu verwalten (z.B. C#, Python). In diesen Fällen müssen die Daten eigenständig verwaltet werden. Wenn die Daten nicht selber verwaltet werden und dann wegen dieser Unzulänglichkeiten das Ergebnis der Aufgabe nicht berechnet werden kann, gibt es einen Abzug. Dieser Abzug kann vermieden werden, wenn durch eine Überschlagsrechnung (wie oben) das jeweilige Lösungsjahr vernünftig abgeschätzt wird.

Aufgabe 2: Rhinozelfant

2.1 Lösungsidee

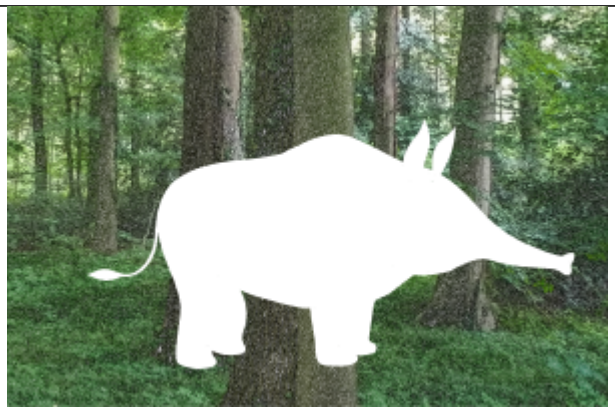
Die Aufgabenstellung verlangt nur, die Eingabebilder einzulesen und in Ausgabebilder auszugeben, bei denen die Pixel komplett weiß markiert werden, die Teil einer Schuppe eines Rhinozelfanten sein *könnten*. Ein Pixel kann Teil einer Schuppe sein, wenn mindestens eines der vier Nachbarpixel genau die gleiche Farbe hat. Deshalb reicht es, für alle Pixel des Eingabebildes die vier Nachbarpixel zu prüfen (andere Kriterien sind prinzipiell auch möglich, aber vermutlich schwerer zu begründen). Wenn eines dieser vier Pixel die genau gleiche Farbe hat, wird im Ausgabebild das Pixel auf weiß gesetzt. Die Analyse, ob auf dem Bild wohl ein Rhinozelfant zu sehen ist, bleibt dann dem menschlichen Betrachter überlassen.

2.2 Umsetzung

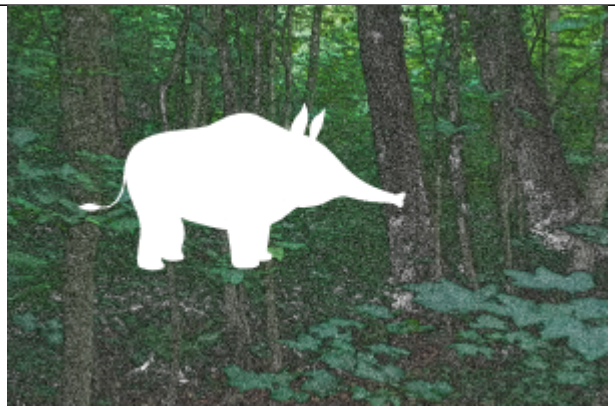
In Java gibt es die Klasse `javax.imageio.ImageIO`, mit der Bilder in den Formaten GIF, PNG, JPEG, BMP und WBMP eingelesen werden können. Dabei entsteht ein Objekt der Klasse `java.awt.image.BufferedImage`. In dem Objekt kann mit zwei verschachtelten Schleifen (für die Zeilen und die Spalten) jedes Pixel einzeln ausgelesen und mit den Werten aller vier Nachbarpixel verglichen werden. Das Ergebnisbild wird in einem neuen Pixelarray aufgesammelt. Wenn kein Nachbarpixel genau die gleiche Farbe hat, wird das Ausgangspixel in das Ergebnisbild geschrieben, sonst ein komplett weißes Pixel. Am Schluss wird das Ergebnisbild in eine neue Datei geschrieben. Die Ausgabe kann ein beliebiges Dateiformat haben. Ohne eine Bibliothek ist z.B. das PPM-Format sehr einfach schreibbar.

2.3 Ergebnisse

Für die Beispieldateien von der Webseite ergeben sich folgende Ergebnisse (mit verkleinerten Bildern). Dabei ist jeweils mit angegeben, ob bei der menschlichen Analyse ein Rhinozelfant entdeckt wurde. Insgesamt sind in den Bildern mit den Nummern 1, 2, 4, 8 und 9 Rhinozelfanten zu erkennen.

Nummer	Ergebnisbild	Rhinozelfant?
1		ja
2		ja
3		nein

4		ja
5		nein
6		nein

7		nein
8		ja
9		ja

2.4 Bewertungskriterien

- Die Bilder müssen korrekt eingelesen, die Ergebnisbilder korrekt abgespeichert werden. Die Beispieleingaben wurden in verschiedenen Formaten bereit gestellt; es ist egal, mit welchem Format die Einsendung arbeitet. Bibliotheken zur Verarbeitung von Bilddateien bzw. -daten dürfen genutzt werden.
- Es muss beschrieben sein, wann ein Pixel ein Teil einer Rhinozelfantenschuppe sein kann (und deswegen weiß eingefärbt wird). Manche Kriterien können zu Fehlern führen, etwa wenn nur immer ein bestimmtes Nachbarpixel betrachtet wird; die Ergebnisbilder weisen dann Streifen oder ähnliche Muster auf. Das Kriterium muss zumindest grob begründet sein.
- Das Rhinozelfantenschuppenpixelkriterium muss im Programm korrekt umgesetzt sein.
- Die Rhinozelfantenentscheidung muss für alle neun BWINF-Beispiele angegeben sein. Zu genügend vielen Beispielen (also einigen positiven und einigen negativen) sollten auch Ergebnisbilder in der Dokumentation enthalten sein. Wenn die Bilder den Rhinozelfanten deutlich zeigen, ist es akzeptabel, wenn die Entscheidung nicht explizit angegeben ist.

Aufgabe 3: Rotation

3.1 Lösungsidee

Mit einem analytischen Ansatz lässt sich dieses Problem schlecht lösen: Es gibt keine einfache Formel, die man anwenden kann, um eine möglichst kurze Reihe von Rotationen zu erhalten, sodass ein Stäbchen aus dem Rahmen fällt. Es gibt einige Sonderfälle, bei denen man direkt ausschließen kann, dass jemals ein Stäbchen aus dem Rahmen fallen wird, zum Beispiel, wenn kein Stäbchen in die richtige Richtung zeigt. Außerdem gibt es Fälle, bei denen wir auf einen Blick sagen können, dass nur eine Rotation notwendig ist, um ein Stäbchen aus dem Rahmen fallen zu lassen.

Abbildung 2: Zwei Sonderfälle: links kann kein Stäbchen herausfallen, rechts fällt durch eine Rotation nach rechts das rote Stäbchen heraus.

Solche Sonderfälle wollen wir jedoch nicht näher betrachten, denn sie machen nur einen sehr kleinen Teil aller möglichen Eingaben aus. Im Normalfall können wir eine Lösung nicht einfach erkennen; wir müssen nach ihr suchen, indem wir jede mögliche Lösung ausprobieren.

Suchbaum

Dazu betrachten wir einen Suchbaum, dessen Knoten jeweils einem Puzzle-Zustand entsprechen; also einer Lage der Stäbchen im Puzzle, die aus dem Anfangszustand des Puzzles durch irgendeine Abfolge von Rotationen entstanden ist. Die Wurzel des Baums ist der Anfangszustand des Puzzle, den wir einlesen. Das linke Kind eines jeden Knotens ist der Zustand, den man durch eine Linksrotation erhält. Dem entsprechend ist das rechte Kind der Zustand, den man durch eine Rechtsrotation erhält. Der Suchbaum des Puzzles aus der Beispieleingabe `rotation1.txt` ist exemplarisch und ausschnittsweise in Abbildung 3 zu sehen.

Indem wir uns in diesem Baum entlang der Kanten von Zustand zu Zustand hangeln, können wir einen Zustand entdecken, bei dem ein Stäbchen herausgefallen ist. Falls ein solcher Zustand nämlich existiert, muss er irgendwann in diesem Baum auftauchen. Wir müssen jedoch aufpassen, in welcher Reihenfolge wir den Baum durchsuchen, denn wir wollen möglichst schnell eine kürzest-mögliche Lösung finden. Jedes Entlanghangeln an einer Baumkante entspricht einer Rotation. Also brauchen wir ein Suchverfahren, das die Knoten in aufsteigender Entfernung von der Wurzel durchsucht und den ersten Treffer ausgibt. Unter Entfernung eines Knotens von der Wurzel verstehen wir die Anzahl der Kanten, denen man von der Wurzel aus folgen muss, um den Knoten zu erreichen.

Eine Suche, die diese Eigenschaft erfüllt, ist in der Informatik als Breitensuche bekannt. Um auf einem Baum eine Breitensuche durchzuführen, beginnt man bei der Wurzel. Erfüllt die Wurzel die gesuchte Eigenschaft (in unserem Fall wollen wir wissen, ob ein Stäbchen aus dem Puzzle herausgefallen ist), ist man fertig. Ansonsten untersucht man der Reihe nach alle Kinder. Erfüllt

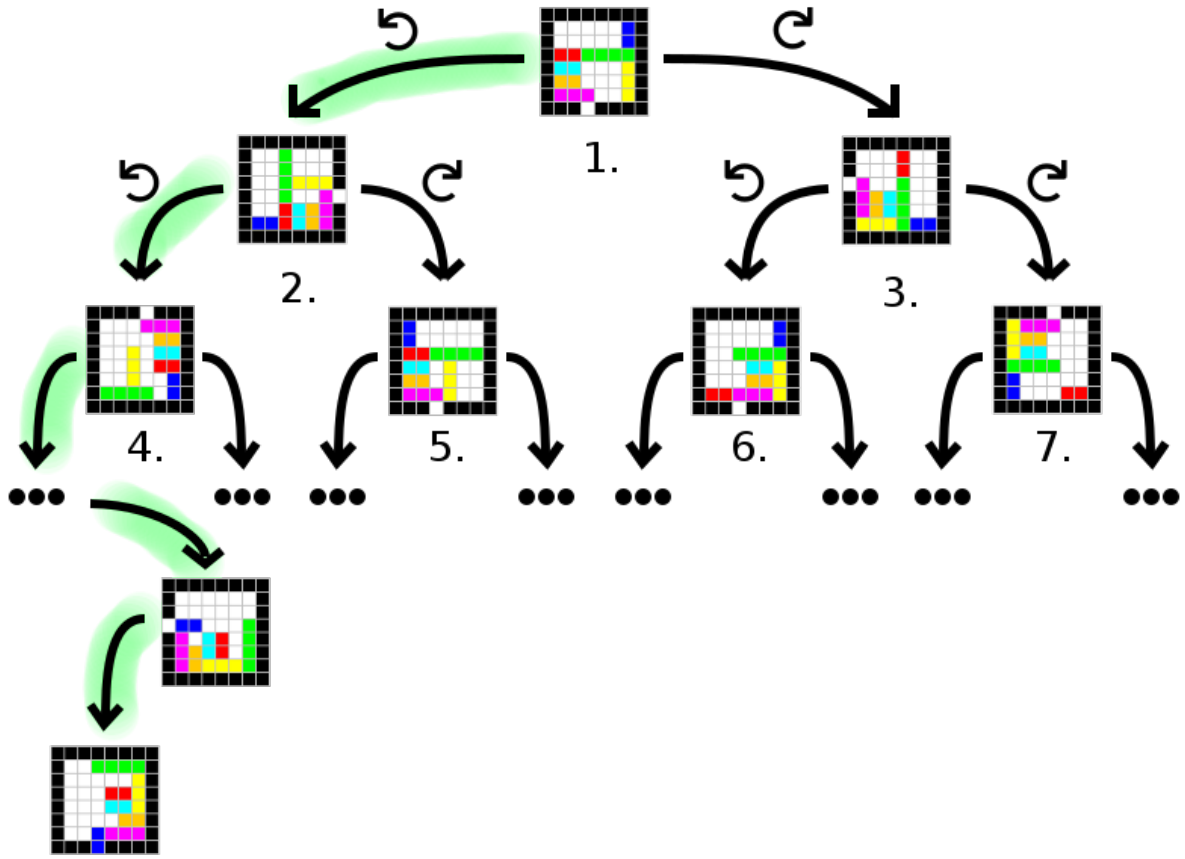


Abbildung 3: Der „obere“ Teil eines Suchbaums und der (unvollständig abgebildete) Pfad zur Lösung. Die Reihenfolge, in welcher die Knoten des Baums von der Breitensuche durchsucht werden, ist durch die Zahl unter den Knoten angegeben. Die grün unterlegten Kanten bilden den Pfad, der zu der letztlich entdeckten Lösung führt. Dieser Baum bezieht sich auf die erste Beispieleingabe.

keines der Kinder die gewünschte Eigenschaft, überprüft man nun die Kinder der Kinder, und so weiter, bis man einen Knoten gefunden hat, der die Eigenschaft erfüllt. (Die Reihenfolge, die eine Breitensuche bei der Untersuchung des Suchbaums in Abbildung 3 wählt, ist dort jeweils unter den Zuständen eingetragen.)

Es gibt keinen Knoten, der näher an der Wurzel liegt als dieser nun entdeckte Knoten und auch die gesuchte Eigenschaft erfüllt. Denn alle Knoten, die näher an der Wurzel liegen, hat die Breitensuche bereits untersucht. Es kann aber sein, dass man den gesamten Baum durchsucht, ohne einen Knoten zu finden, der die gewünschte Eigenschaft erfüllt. Auch kann es mehr als eine Lösung geben; durch Weitersuchen kann man diese auch finden, uns interessieren diese anderen Lösungen jedoch nicht.

Größe des Suchbaums

Die Größe des Suchbaums ist wichtig: je größer der Suchbaum, desto länger müssen wir möglicherweise darin suchen, um eine Lösung zu finden. Betrachten wir den Suchbaum etwas genauer. Auf den ersten Blick scheint er unendlich groß zu sein, denn das Puzzle kann man in jedem

Zustand noch einmal jeweils nach links und nach rechts drehen, um noch mehr Folgezustände und damit Knoten im Baum zu erzeugen. Für uns ist jedoch nicht der gesamte Suchbaum von Interesse. Stellt man sich nämlich einen leeren Rahmen und eine Anzahl von Stäbchen vor, die man in diesem Rahmen platzieren möchte, stellt man fest, dass es nur eine endliche Anzahl von Möglichkeiten gibt, diese Stäbchen im Rahmen zu positionieren.

Dies erkennt man anhand eines Beispiel-Puzzles: Wir haben einen Rahmen der Größe 6, der also im Inneren $4 \cdot 4 = 16$ Felder fassen kann. Jetzt wollen wir darin 8 Stäbchen der Länge 1 positionieren (also Quadrate): dann haben wir für das erste Stäbchen 16 Möglichkeiten, es zu positionieren, denn alle Felder sind noch frei. Für das zweite Stäbchen bleiben 15 Möglichkeiten, und so weiter. Also gibt es insgesamt $16 \cdot 15 \cdot 14 \cdot 13 \cdot 12 \cdot 11 \cdot 10 \cdot 9 = \frac{16!}{(16-8)!} = 518.918.400$ Möglichkeiten, die Stäbchen im Rahmen zu positionieren. Für uns sind aber nicht all diese Anordnungen relevant: die meisten der Anordnungen kann man nämlich durch Rotation gar nicht erreichen. Nach einer Rotation darf kein Stäbchen mehr in der Luft hängen, sodass viele der Möglichkeiten bereits ausscheiden. Außerdem kann man in einigen Puzzles nicht jede beliebige Anordnung der Stäbchen erreichen. Wenn das obige Puzzle zum Beispiel zusätzlich einen Stab der Länge 4 hätte, dann könnten sich die quadratischen Stäbchen niemals von der einen Seite des Stabes auf die andere Seite des Stabes bewegen.

Unser Suchbaum ist also endlich, denn irgendwann wiederholen sich die Zustände. Wenn wir dies bei der Breitensuche berücksichtigen, indem wir die Kinder eines Knotens nicht untersuchen, falls wir den Knoten selbst bereits einmal untersucht haben, dann wird die Suche terminieren. Entweder haben wir dann einen Zustand entdeckt, bei dem ein Stäbchen herausgefallen ist (einen *Lösungszustand*), oder uns sind die zu untersuchenden Knoten ausgegangen.

Ob das Durchsuchen des Baumes machbar ist, hängt stark davon ab, als wie gutartig sich das Problem herausstellt. An dieser Stelle nehmen wir vorweg: die Lösung einer der Beispieleingaben benötigt 90 Rotationen. Würde man den vollständigen Suchbaum bis zu dieser Tiefe durchsuchen, dann müsste man $2^{90} > 10^{27}$ Puzzle untersuchen. Selbst mit einem Supercomputer und hoch optimiertem Code kann man dies nicht in annehmbarer Zeit schaffen, falls man für die Untersuchung eines einzigen Puzzles 100 ns veranschlagt. Dann bräuchte man immer noch etwa 10^{20} Sekunden, um den gesamten Baum zu durchsuchen. Das entspricht etwa dem 230-fachen Alter des Universums. Da das unten beschriebene Programm aber eine Lösung gefunden hat, können wir davon ausgehen, dass sich die Puzzle-Zustände im Baum oft wiederholen und der Suchbaum deshalb tatsächlich klein bleibt.

3.2 Beispiele

Ausgabe zu rotation1.txt

Das Programm terminiert sehr schnell und gibt aus:

```
Es wurde eine Lösung mit 6 Rotationen gefunden.

########                              #55   4#
#     0#                              #666  4#
#     0#                              ### ####
#112222#
#33   4#                              Rotation nach links:
```

```
########
#  2   #
#  2   #
#  2444#
#  2  6
#  1356#
#001356#
########

Rotation nach links:
#### ###
#   666#
#    55#
#  4 33#
#  4 11#
#  4  0#
#2222 0#
########

Rotation nach links:
########
#      #
#    00#
      2#
#6 31 2#
#6531 2#
#654442#
########
```

```
Rotation nach links:
########
#  2222#
#     4#
#   114#
#   334#
# 0  55#
# 0 666#
### ####

Rotation nach rechts:
########
#      #
#      #
 00   2#
#6 31 2#
#6531 2#
#654442#
########

Rotation nach links:
########
#  2222#
#     4#
#   114#
#   334#
#    55#
#  0666#
###0####
```

Ausgabe zu rotation2.txt

Das Programm terminiert sehr schnell. Da die Ausgabe sehr lang ist, geben wir lediglich die Rotationen an, mit R für Rechtsrotation und L für Linksrotation:

```
Es wurde eine Lösung mit 22 Rotationen gefunden.

RRRRLRRLLLLLLLRRRRRRLR
```

Ausgabe zu rotation3.txt

Auch diese sehr lange Lösung terminiert schnell (< 1 Sekunde). Dies spricht dafür, dass ein Großteil des Suchbaumes übersprungen wird, weil sich die Puzzles schnell wiederholen. Wäre dies nicht so, würden wir, wie am Ende von Teil 1 erwähnt, niemals eine Lösung finden.

Da die Ausgabe sehr lang ist, geben wir wieder lediglich die Rotationen an:

```
Es wurde eine Lösung mit 90 Rotationen gefunden.
```

```
LRRRRLLLRRRLLLLLLLRRLLRRRRRRRLRRRRRRRRRRRLLLR
RRRRLLLLLRRRRLLRRRRRRRLRLLLLLLLLLLLLRLLRRRRRLR
```

Ausgabe zu einem Puzzle ohne Lösung

Das übergebene Puzzle entspricht dem Puzzle ohne Lösung aus Abbildung 2:

```
##########
#        #
#        #
#        #
#        #
#  555555#
#444     #
#  6666  #
#111 2222#
#### #####
```

Das Programm terminiert sehr schnell und gibt aus:

```
Es gibt keine Lösung! Es wurden alle Möglichkeiten ausprobiert.
```

3.3 Bewertungskriterien

- Die Rotation muss korrekt konzipiert und realisiert sein. Hier kann man sehr leicht Fehler machen und erhält dann angebliche Lösungen, die aber keine sind. Um eine Lösung zu überprüfen, vergolgt man am besten ein oder zwei Rotationsschritte. Ansonsten kann man die Korrektheit gut an der Länge der gefundenen Lösung beurteilen (siehe Lösung der Beispieleingaben).
- Einfache Probleme wie die erste Beispieleingabe kann man auch ohne Beschneiden des Baums finden; für die erste Beispieleingabe ist ein Lösungszustand unter den ersten 64 untersuchten Puzzle-Zuständen zu finden. Größere Eingaben wie insbesondere das 3. BWINF-Beispiel sind so aber nicht zu lösen. Weder das Verfahren noch die verwendeten Datenstrukturen dürfen zu ineffizient sein. Bei einer Suche muss insbesondere korrekt erkannt werden, ob ein Zustand schon behandelt wurde.
- Das Verfahren soll eine (minimale) Lösung finden können. Eine Tiefensuche ist also nicht geeignet, auch nicht eine Suche mit begrenzter Suchtiefe.
- Terminiert der Algorithmus, wenn keine Lösung existiert? Am besten ist es, wenn dies durch ein Beispiel verifiziert werden kann.
- Alle BWINF-Beispiele müssen bearbeitet worden sein. Die Lösungen sollten geeignet dokumentiert sein. Für ein Beispiel (möglicherweise in den Abschnitten Lösungsidee oder Umsetzung) sollten zumindest einige Rotationen auch bildlich dargestellt sein.

Aufgabe 4: Radfahrspaß

4.1 Lösungsidee

Es wird davon ausgegangen, dass die Geschwindigkeit nicht nur am Ende des letzten Streckenabschnitts sondern am Ende jedes einzelnen Streckenabschnitts den Wert 0 erreichen darf. Nach diesem Moment muss auf einer Geraden oder einem Gefälleabschnitt wieder beschleunigt werden.

Teilaufgabe 1

Für die Teilaufgabe 1 betrachten wir beim Durchlauf durch den Parcours für jeden Abschnitt die jeweils potentiell möglichen nicht-negativen Fahrgeschwindigkeiten an dieser Stelle. Hier ein paar Beispiele:

davor	Zeichen	danach	nicht-negativ
$\{0\}$	\	$\{1\}$	$\{1\}$
$\{0\}$	/	$\{-1\}$	nicht befahrbar
$\{0\}$	_	$\{-1,1\}$	$\{1\}$
$\{1\}$	_	$\{0,2\}$	$\{0,2\}$
$\{0,2\}$	_	$\{-1,1,3\}$	$\{1,3\}$
$\{1,3\}$	_	$\{0,2,4\}$	$\{0,2,4\}$

In der Spalte „davor“ steht die Menge der möglichen Fahrgeschwindigkeiten vor dem Parcoursabschnitt, und unter „Zeichen“ steht das eingelesene Zeichen. Unter „danach“ folgt die Menge der nach dem Abschnitt möglichen Geschwindigkeiten, und unter „nicht-negativ“ steht die Menge aller nicht-negativen Geschwindigkeiten.

Folgende Beobachtungen können gemacht werden:

- Im Laufe des Parcours darf die höchstmögliche Geschwindigkeit auf keinen Fall kleiner als Null werden. Dann würde man mitten im Parcours stehen bleiben.
- Es darf nicht mit einer niedrigstmöglichen Geschwindigkeit kleiner als 0 gefahren werden. Auch dann würde man stehen bleiben.
- Wie direkt zu Beginn gesagt: Eine niedrigstmögliche Geschwindigkeit gleich 0 ist – nach einer (potenziellen) Steigung – möglich; es muss dann aber ein (potenzielles) Gefälle (also \ oder _) folgen.
- Wenn am Ende des Parcours die Menge der möglichen Geschwindigkeiten die Geschwindigkeit 0 enthält, dann ist der Parcours korrekt befahrbar.
- Die Geschwindigkeiten in der Menge der möglichen Geschwindigkeiten von der niedrigstmöglichen bis zur höchstmöglichen Geschwindigkeit steigen immer in Zweierschritten an. Deshalb reicht es, sich immer die zum aktuellen Streckenpunkt niedrigstmögliche nicht-negative Geschwindigkeit $v_{\min}$ und die höchstmögliche Geschwindigkeit $v_{\max}$ zu merken. Wenn am Ende des Parcours die niedrigstmögliche nicht-negative Geschwindigkeit genau gleich 0 ist, dann ist der Parcours befahrbar.

Vor dem ersten Zeichen ist nur die Fahrgeschwindigkeit 0 möglich. Wenn danach ein Gefälleabschnitt kommt, wird jedes Zeichen des Parcours der Reihe nach durchgegangen, um die zu jedem Zeitpunkt jeweils höchste bzw. niedrigste Fahrgeschwindigkeit zu bestimmen. Es wird dabei mit der möglichen Niedrigstgeschwindigkeit $v_{\min} = 0$ und der möglichen Höchstgeschwindigkeit $v_{\max} = 0$ gestartet. Je nach eingelesenem Zeichen werden die beiden Geschwindigkeiten angepasst:

- Wenn ein \ eingelesen wird(es geht bergab), werden $v_{\min}$ und $v_{\max}$ um 1 erhöht, da die Geschwindigkeit hier nur höher werden kann.
- Wenn ein / eingelesen wird (es geht bergauf), werden $v_{\min}$ und $v_{\max}$ um 1 erniedrigt, da die Geschwindigkeit hier nur niedriger werden kann.
- Wenn aber ein _ eingelesen wird, dann kann der Fahrer beschleunigen oder bremsen. Die Niedrigstgeschwindigkeit $v_{\min}$ wird dann um 1 verringert und die Höchstgeschwindigkeit $v_{\max}$ um 1 erhöht.

Wenn beim Einlesen des Parcours bei einem Element die mögliche Höchstgeschwindigkeit unter 0 fällt, bedeutet es, dass man an der Stelle spätestens stehen bleibt und den Parcours nicht befahren kann. Wenn die mögliche Niedrigstgeschwindigkeit unter 0 fällt (sie ist dann genau -1), bedeutet es, dass vorher im Parcours einmal zu viel gebremst wurde. In diesem Fall wird die mögliche Niedrigstgeschwindigkeit um 2 erhöht, damit sie wieder größer als 0 ist (sie ist dann genau $+1$). Am Ende des Parcours muss geschaut werden, ob die Endgeschwindigkeit 0 möglich ist. Dies ist genau dann der Fall, wenn die mögliche Niedrigstgeschwindigkeit genau gleich 0 ist.

Für Teil 1 reicht es also, sich im Laufe des Einlesens zwei Werte zu merken: Die jeweils aktuell mögliche Niedrigst- und Höchstgeschwindigkeit.

Alternative Herangehensweisen

Natürlich gibt es auch andere Möglichkeiten, diese Aufgabe zu lösen. Allerdings muss man genau aufpassen, dass die Herangehensweise auch wirklich den in der Aufgabenstellung angegebenen Regeln entspricht. Deshalb wollen wir zwei weitere Lösungswege, die auch in vielen Einsendungen beschrieben wurden, exemplarisch vorstellen; der eine funktioniert, der andere nicht.

Nur Niedrigstgeschwindigkeit Alternativ kann man beim Durchlaufen des Parcours nur die Niedrigstgeschwindigkeit $v_{\min}$ wie oben mitführen, wenn man gleichzeitig auch über die Anzahl der bisherigen Bremsvorgänge n_{brems} Buch führt. Sollte die Niedrigstgeschwindigkeit $v_{\min} < 0$ werden, muss ein bisheriger Bremsvorgang in eine Beschleunigung umgewandelt werden (und $v_{\min}$ um 2 erhöht). Ist kein Bremsvorgang mehr verfügbar, so lässt sich der Parcours nicht befahren. Wie oben gilt dann auch, dass am Ende die Niedrigstgeschwindigkeit $v_{\min} = 0$ sein muss, damit der Parcours regelkonform befahrbar ist. Mit diesem Verfahren lassen sich dementsprechend auch die richtigen Ergebnisse erzielen.

Nur Höchstgeschwindigkeit Dagegen reicht es *nicht* aus, sich nur die Höchstgeschwindigkeit $v_{\max}$ und die Anzahl der bisherigen Beschleunigungsvorgänge n_{beschl} zu merken. Die Idee hinter diesem Verfahren wäre, am Ende des Parcours bei einer Höchstgeschwindigkeit $v_{\max} > 0$ zu versuchen, eine entsprechende Anzahl von Beschleunigungen in Bremsvorgänge umzuwandeln. Bei dieser Herangehensweise würde allerdings nicht mehr korrekt überprüft, ob die Geschwindigkeit vorher durch das Bremsen kleiner als 0 wird. Dies lässt sich an dem folgendem Minimalbeispiel veranschaulichen:

```
_ _ _ \ \ /
```

Hier kommt man nach dreimaligem Beschleunigen mit einer höchstmöglichen Geschwindigkeit von $v_{\max} = 4$ an. Wenn man jedoch nun zwei der drei Beschleunigungen in zwei Bremsvorgänge umwandelt, so wird die Geschwindigkeit in der Mitte des Parcours auf jeden Fall negativ. Dieses Verfahren führt also nicht in allen Fällen zu einem richtigen Ergebnis.

Teilaufgabe 2

In Teil 2 geht es darum, eine Fahrtanweisung auszugeben, die für jeden flachen Streckenteil entweder ein + oder ein − ausgibt. Wenn der Fahrer immer nur beschleunigt (+), kommt er mit der möglichen Höchstgeschwindigkeit $v_{\max}$ am Ende des Parcours an. Damit er am Ende des Parcours mit der Geschwindigkeit 0 ankommt, muss er auf den letzten $n_{\text{brems}} = \frac{v_{\max}}{2}$ flachen Streckenteilen bremsen anstatt zu beschleunigen. Mit dieser Fahrstrategie kann es (bei einem befahrbaren Parcours) nicht passieren, dass die aktuelle Geschwindigkeit unter 0 fällt. Der Wert von $v_{\max}$ ist schon aus Teil 1 am Ende des Parcours bekannt. Leider sind die $\frac{v_{\max}}{2}$ flachen Streckenteile, auf denen gebremst werden muss, am Ende des Parcours, und es ist noch nicht bekannt, wie oft am Anfang auf flachen Streckenteilen beschleunigt werden muss. Deshalb wird zusätzlich zu den Werten aus Teil 1 noch die Anzahl der flachen Streckenteile gezählt (n_{flach}). Nun kann auf den ersten $n_{\text{beschl}} = n_{\text{flach}} - n_{\text{brems}}$ flachen Streckenabschnitten beschleunigt und auf den restlichen n_{brems} flachen Streckenabschnitten gebremst werden. Damit wird der Parcours korrekt befahren.

Für Teil 2 reicht es also, sich zusätzlich beim ersten Lauf die Anzahl der gefundenen flachen Stellen zu merken. Damit reichen drei gemerkte Zahlenwerte während des ersten Durchlaufs, um Teil 1 und Teil 2 zu beantworten.

Die Fahrtanweisung ist sogar unabhängig von den nicht flachen Streckenabschnitten des Parcours. Deshalb kann die Fahrtanweisung ausgegeben werden, ohne den Parcours ein zweites Mal einzulesen. Dabei fehlt dann natürlich der Radfahrspaß; dafür muss man nötigenfalls doch selber noch einmal den Parcours mit der Fahrtanweisung durchfahren.

Alternativ kann es vorkommen, dass für Teil 2 an Stelle der möglichen Höchstgeschwindigkeit $v_{\max}$ die Differenz aus der Anzahl der Steigungsabschnitte und der Anzahl der Gefälleabschnitte benutzt wird. Dann wird mit vier gemerkten Zahlen gearbeitet.

4.2 Umsetzung

Der folgende Algorithmus stellt eine Umsetzung des Verfahrens dar:

```
anzahlFlach = 0;
minSpeed = 0;
maxSpeed = 0;

für jedes Zeichen c aus dem Parcours:
  wenn
    c == '\' :
      minSpeed um eins erhöhen
      maxSpeed um eins erhöhen
    c == '/' :
      minSpeed um eins erniedrigen
      maxSpeed um eins erniedrigen
    c == '_' :
      minSpeed um eins erniedrigen
      maxSpeed um eins erhöhen
      anzahlFlach um eins erhöhen
  wenn maxSpeed kleiner 0:
    Parcours ist nicht befahrbar. Ende!
  wenn minSpeed kleiner 0:
    minSpeed um zwei erhöhen

wenn minSpeed größer 0:
  Parcours ist nicht befahrbar. Ende!
wenn minSpeed gleich 0:
  Parcours ist befahrbar!

AnzahlBremsen = maxSpeed/2
AnzahlBeschleunigen = anzahlFlach - AnzahlBremsen
AnzahlBeschleunigen mal '+' ausgeben
AnzahlBremsen mal '-' ausgeben
```

Beim Einlesen der Datei sollten die Zeichen einzeln nacheinander eingelesen werden, damit es bei besonders großen Parcoursdateien nicht zu Speicherproblemen kommt.

4.3 Beispiele

Für die Beispiele von der Webseite ergeben sich die in Tabelle 3 dargestellten Ergebnisse. Um die Ergebnisse (insbesondere bei den sehr großen Beispielen) nachvollziehen zu können, ist es von Vorteil, die Anzahl der Beschleunigungsanweisungen und die Anzahl der Bremsanweisungen zusammen mit (oder an Stelle) der ausgeschriebenen Fahrtanweisung auszugeben.

4.4 Hintergrund der Aufgabe

Diese Aufgabe ist äquivalent zu der Frage, ob aus einer Folge von Klammern und Fragezeichen eine gültige Klammerung erzeugt werden kann. Für den Parcours C aus den Materialien ist es z. B. die Frage, ob man aus `(?(??)` eine gültige Klammerung erzeugen kann. Dabei darf ein `?` entweder durch eine öffnende oder eine schließende Klammer ersetzt werden. Im Ergebnis muss es zu jeder öffnenden Klammer eine passende schließende Klammer geben.

Parcours	fahrbar?	Zeit	Anweisung
parcoursA	nein	0.109s	
parcoursB	nein	0.112s	
parcoursC	ja	0.115s	+−−
parcours0	nein	0.111s	
parcours1	nein	0.143s	
parcours2	nein	0.169s	
parcours3	nein	0.221s	
parcours4	ja	0.591s	1 001 577 +, dann 83 −
parcours5	nein	0.469s	
parcours6	nein	0.959s	
parcours7	nein	1.803s	
parcours8	nein	3.428s	
parcours9	ja	15.599s	49 992 957 +, dann 252 −

Tabelle 3: Ergebnisse für die Beispiel-Parcours. Bei fahrbahren Parcours ist die Zeit inklusive der Generierung der Fahranweisung in einer Datei.

4.5 Bewertungskriterien

- Die Bestimmung, ob der Parcours befahrbar ist, darf die Zeichen nur nacheinander einlesen und immer nur ein Zeichen betrachten. Das Einlesen in einen Buffer oder internen String ist erlaubt, wenn die Zeichen dann nur sequentiell gelesen werden.
  Das Verfahren soll mit drei, maximal vier gemerkten Zahlen funktionieren.
- Die Geschwindigkeit bei einer Fahrt darf nicht negativ werden.
- Eine Geschwindigkeit von 0 nach einem Streckenabschnitt ist möglich, wenn danach wieder beschleunigt werden kann; z. B. ist der Parcours \/_/ befahrbar. Wenn das Verfahren eine punktuelle 0-Geschwindigkeit grundsätzlich ausschließt, muss das begründet sein.
- Die Entscheidung zur Befahrbarkeit darf auch nicht aus anderen Gründen fehlerhaft sein.
- Für Teil 2 muss eine korrekte Bestimmung der Fahranweisungen (in eine Datei) ausgegeben werden. Es müssen gleich viele Zeichen + oder − sein wie _ in der Eingabe. Die Anweisungen müssen eine gültige Fahrt ergeben. Wenn die Fahranweisungen im Block vorkommen (z. B. erst +, danach −), dann reicht es auch, die jeweilige Anzahl der Zeichen auszugeben.
- Es ist zwar nicht explizit gefordert, aber zumindest einige BWINF-Beispiele sollen gelöst werden: mindestens drei der Beispiele 0 bis 9, darunter mindestens je ein befahrbarer Parcours und ein nicht befahrbarer. Die Ergebnisse von Teil 2 müssen dabei entweder als Datei in der Abgabe mitgeliefert werden (Ausdrucken ist nur bei kleinen Beispielen sinnvoll) oder in „Blocknotation“ in der Dokumentation angegeben werden. Im letzteren Fall genügt zur Not, wenn nur die Anzahl der Beschleunigungen angegeben ist; bei den übrigen geraden Abschnitten muss dann gebremst werden. Gerade bei den sehr großen Beispielen ist es nicht nötig, die Ausgaben zu Teil 2 mit einzureichen (geschweige denn abzudrucken).

Aufgabe 5: Buhnenrennen

5.1 Einleitung

Wer nicht am Meer wohnt und das Wort Buhne im Titel dieser Aufgabe zum ersten Mal gelesen hat, kann sich mit einen Blick in den Wikipedia-Artikel zum Thema helfen. Dort erfährt man, dass sie auch als Stack, Höft, Kribbe, Schlenge oder Schlacht bezeichnet wird; diese geläufigeren Begriffe hat sicher jeder schon einmal gehört. Beim Lösen der Aufgabe hilft der Artikel aber nicht weiter, denn erstaunlicherweise scheint es sich bei der Verfolgungsjagd von Minnie und Max um ein seltenes Verhalten unter Hunden an Nordseestränden zu handeln. Tatsächlich herrscht an diesen meist eine Leinenpflicht, sodass sich der reale Nutzen einer Lösung der Aufgabe eher bescheiden ausnimmt, ganz zu schweigen von den Schwierigkeiten, der Hündin Minnie den optimalen Fluchtweg beizubringen, während diese von Max verfolgt wird. Wir wollen diese Aufgabe daher als eher akademische Aufgabe betrachten.

Gesucht ist also ein Weg, auf dem Minnie mit Sicherheit vor Max fliehen kann. Max ist zwar schneller, passt aber nicht durch alle Löcher in den Buhnen.

Ein Minnie-Weg ist sicher, wenn Minnie unterwegs an keinem Ort anzutreffen ist, an dem Max zuvor eintreffen kann, egal welchen Weg er nimmt. Zunächst sieht es also so aus, als müsste jeder mögliche krumme Weg zwischen den Buhnen getestet werden. Man kann sich aber überlegen, dass es ausreicht, nur direkte Wege zwischen den Buhnenlöchern zu untersuchen und dabei nur die Zeiten berücksichtigen muss, zu denen die Hunde an den Löchern ankommen. Zuerst begründen wir, warum es reicht, die Zeiten an den Buhnenlöchern zu prüfen: Angenommen, es gibt einen Minnie-Weg, bei dem es einen Punkt unterwegs gibt, den Max vor Minnie erreichen kann. Dann kann Max zu diesem Punkt laufen und von dort aus dem Minnie-Weg zum nächsten Buhnenloch folgen. Da Max schneller läuft als Minnie, ist er kommt er auch dort vor Minnie an, und es reicht, Max' Zeit am Buhnenloch zu prüfen. Nun begründen wir, warum gerade Wege zwischen den Buhnen ausreichen: Da es ausreicht, die Zeit an den Löchern zu prüfen, ist für Minnie der beste Weg stets der, der sie am schnellsten zu diesem Loch bringt. Dies ist ein gerader Weg zwischen den Buhnen. Gleiches gilt für Max.

Die obige Beobachtung erlaubt uns, das Problem mithilfe eines Graphen zu modellieren. Ein Graph besteht aus Knoten, die mit Kanten verbunden sind. Zwei Knoten heißen benachbart, wenn es eine Kante gibt, die sie miteinander verbindet. Für die Zwecke der Aufgabe können wir die Buhnenlöcher als Knoten interpretieren und die möglichen Laufwege der Hunde von Buhnenloch zu Buhnenloch als Kanten zwischen den Knoten.

Die Aufgabenstellung verlangt, dass wir einen beliebigen Weg finden, der für Minnie sicher ist. Dazu bietet es sich an, zunächst für jedes Buhnenloch bestimmen, in welcher Zeit Max dieses im besten Fall erreichen kann. Bei der anschließenden Suche nach einem sicheren Minnie-Weg müssen wir dann solche ausschließen, die über einen Knoten führen, an denen Max vorher eintreffen kann.

5.2 Darstellung als Graph

Schauen wir uns zunächst an, wie das Problem als Graph modelliert werden kann. Aus den gegebenen Buhnendaten können wir zwei Graphen mit identischer Knotenmenge bilden, jeweils

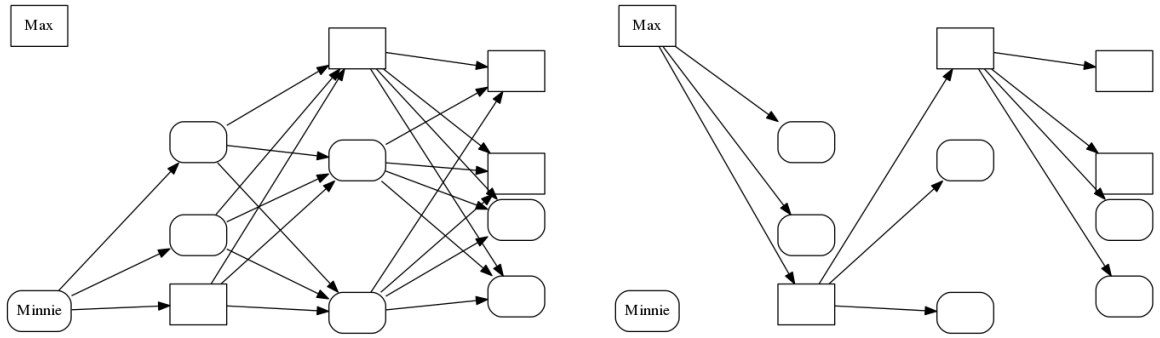


Abbildung 4: Das Beispiel aus der Aufgabenstellung als Graph interpretiert. Der linke Graph zeigt Minnies mögliche Laufwege, der rechte Graph die von Max. Abgerundete Knoten stellen kleine Minnie-Löcher dar, eckige dagegen große Max-Löcher.

einen für Max und einen für Minnie. Da die Kanten die möglichen Laufwege der Hunde darstellen sollen und Max nicht alle Wege zur Verfügung stehen, wird Max' Graph einige Kanten weniger enthalten als der von Minnie. Die Knoten stellen die Löcher in den Buhnen dar; wir haben also zwei verschiedene Arten, nämlich solche, die kleine Minnie-Löcher und solche, die große Max-Löcher darstellen.

Die Kanten verbinden dann die Knoten benachbarter Buhnen. Dabei wird in Minnies Graph jeder Knoten einer Buhne mit jedem anderen verbunden, da sie von jedem Knoten aus zu jedem anderen laufen kann. In Max' Graph lassen wir solche Kanten aus, die von einem kleinen Minnie-Loch aus starten, belassen aber solche Kanten, die in einem kleinen Loch münden. Schließlich kann Max ja durchaus bis zu dem Loch hin laufen, um Minnie dort abzufangen.

Kanten zwischen den Knoten einer Buhne brauchen wir nicht hinzufügen, ebenso solche, die rückwärts zu vorhergehenden Buhnen führen, da jeder Weg, der solche Kanten nutzt, mit Sicherheit kein schnellster Weg sein wird. Das erlaubt uns auch, die Knoten aus der ersten Buhne zu entfernen, die weder Startloch von Max, noch von Minnie sind, denn diese Löcher wird keiner der beiden Hunde im optimalen Fall nutzen.

Betrachten wir das in der Aufgabenstellung gegebene Beispiel. Abbildung 4 zeigt links den zugehörigen Minnie-Graph, rechts den Max-Graph. Hier haben wir bereits den zusätzlichen Knoten der ersten Buhne entfernt.

5.3 Bestimmung der schnellsten Max-Zeiten

Zunächst wollen wir für jeden Knoten der Graphen bestimmen, in welcher Zeit Max diesen erreichen kann. Wir kennen für jeden Knoten die Position, können also zunächst für jede Kante den Abstand berechnen, den sie überbrückt und daraus die Zeit errechnen, die Max braucht, wenn er diese Kante entlang läuft. Sei die Position des ersten Knoten gegeben durch das Tupel (p_x, p_y) und die des zweiten durch (q_x, q_y). Dann ist die Länge l der Kante, die beide verbindet, nach Pythagoras

$$l = \sqrt{(q_x - p_x)^2 + (q_y - p_y)^2}.$$

Max' Geschwindigkeit beträgt laut Aufgabenstellung $v_{\text{Max}} = 30\,\text{km/h} = 8.\bar{3}\,\text{m/s}$, also beträgt die Zeit t, die er für obige Kante der Länge s benötigt,

$$t = \frac{l}{v}.$$

Wir können nun jede Kante in Max' Graphen mit der Zeit, die Max für diese braucht, beschriften. Im Prinzip könnte man nun einen einfachen Kürzester-Weg-Algorithmus benutzen (der den kürzesten Weg von einem Startknoten zu einem bestimmten Zielknoten bestimmt), um für jeden Knoten Max' kürzeste Zeit dorthin zu berechnen. Einen solchen Algorithmus müssten wir aber mehrfach anwenden, für jeden Knoten außer Max' Startknoten. In der Regel lassen sich Kürzester-Weg-Algorithmen leicht so modifizieren, dass sie in einem Lauf die kürzesten Wege von einem Startknoten zu allen anderen Knoten berechnen. Einen solchen, auf die Bedürfnisse dieser Aufgabe noch angepassten „Single-Source Shortest Paths" Algorithmus wollen wir nun beschreiben.

Wir starten an der zweiten Buhne. Für jedes Loch in dieser Buhne, egal ob groß oder klein, bestimmen wir die Zeit, die Max braucht, um dieses zu erreichen. Dazu brauchen wir im ersten Schritt nur die Beschriftung der Kante zu lesen, die diesen Knoten mit dem Max-Startknoten verbindet. Diese Zeit schreiben wir auf den entsprechenden Knoten der zweiten Buhne. Anschließend wenden wir uns der dritten Buhne zu und suchen für jedes Loch der dritten Buhne dasjenige große Loch der zweiten Buhne, das die niedrigste Gesamtzeit hat. Diese ergibt sich aus der dem Loch der zweiten Buhne zugewiesenen Zeit plus der Wegdauer der Kante, die das Loch der zweiten Buhne mit dem der dritten verbindet. Die so ermittelte kleinste Zeit weisen wir dem Loch zu und machen anschließend bei der vierten, fünften, usw. Buhne weiter, bis wir bei der letzten Buhne angekommen sind. Abbildung 5 veranschaulicht den Algorithmus.

5.4 Finden eines Minnie-Weges

Nun müssen wir noch einen sicheren Weg für Minnie finden. Wir können fast den gleichen Algorithmus verwenden wie eben, müssen ihn aber etwas anpassen: Erstens kann Minnie durch alle Löcher schlüpfen. Zweitens müssen wir, nachdem wir für ein Loch die früheste Ankunftszeit Minnies berechnet haben, diese mit Max' Zeit vergleichen.

- Erreicht Minnie das Loch nach Max, ist es unsicher und wir markieren es rot. Bei der weiteren Suche nach schnellsten Verbindungen ignorieren wir alle unsicheren Löcher, da Minnie diese meiden muss.
- Erreicht Minnie das Loch vor Max, so ist es sicher und grün zu markieren. Zusätzlich zur frühesten Ankunftszeit merken wir uns auch, von welchem Vorgängerloch aus diese schnellste Zeit erreicht wird. Dies erlaubt uns am Ende eine Rekonstruktion von Minnies Weg.

Am Ende des Algorithmus müssen wir lediglich in der letzten Buhne nachschauen, ob es dort ein sicheres Loch gibt. Folgen wir den Verweisen auf die jeweiligen Vorgängerlöcher, können wir so einen sicheren Weg für Minnie konstruieren. Natürlich können wir so nicht alle möglichen Wege berechnen, aber laut Aufgabenstellung reicht es, einen einzigen anzugeben. Abbildung 6 veranschaulicht den Vorgang.

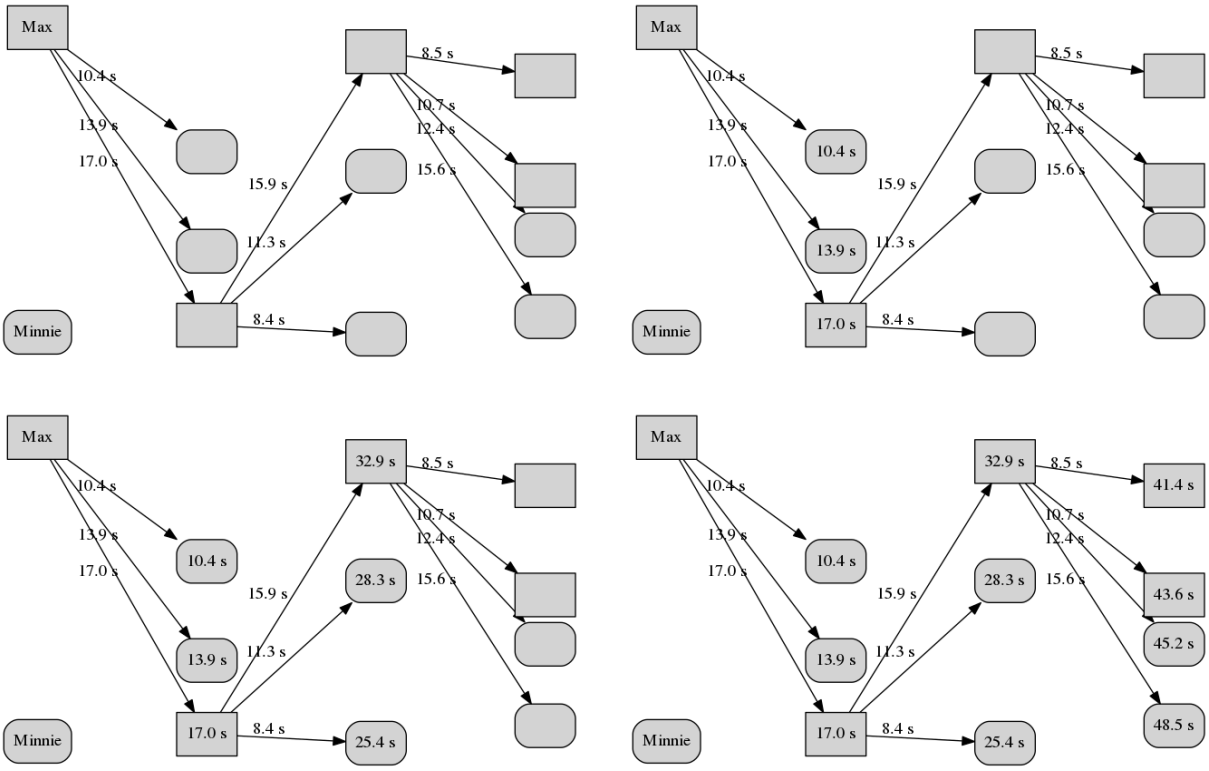


Abbildung 5: Veranschaulichung des Algorithmus, der die schnellsten Max-Zeiten berechnet. Zunächst werden die Kanten mit den jeweils benötigten Laufzeiten beschriftet (oben links). Anschließend wird für jedes Loch der zweiten Buhne die Ankunftszeit berechnet (oben rechts). Danach wird für jedes Loch der dritten Buhne dasjenige Max-Loch der zweiten Buhne gesucht, dessen Ankunftszeit plus die Zeit der verbindenden Kante am kleinsten ist. In diesem Fall gibt es jeweils nur ein Max-Loch, das damit automatisch optimal ist (unten links). Das gleiche wird für die vierte Buhne wiederholt.

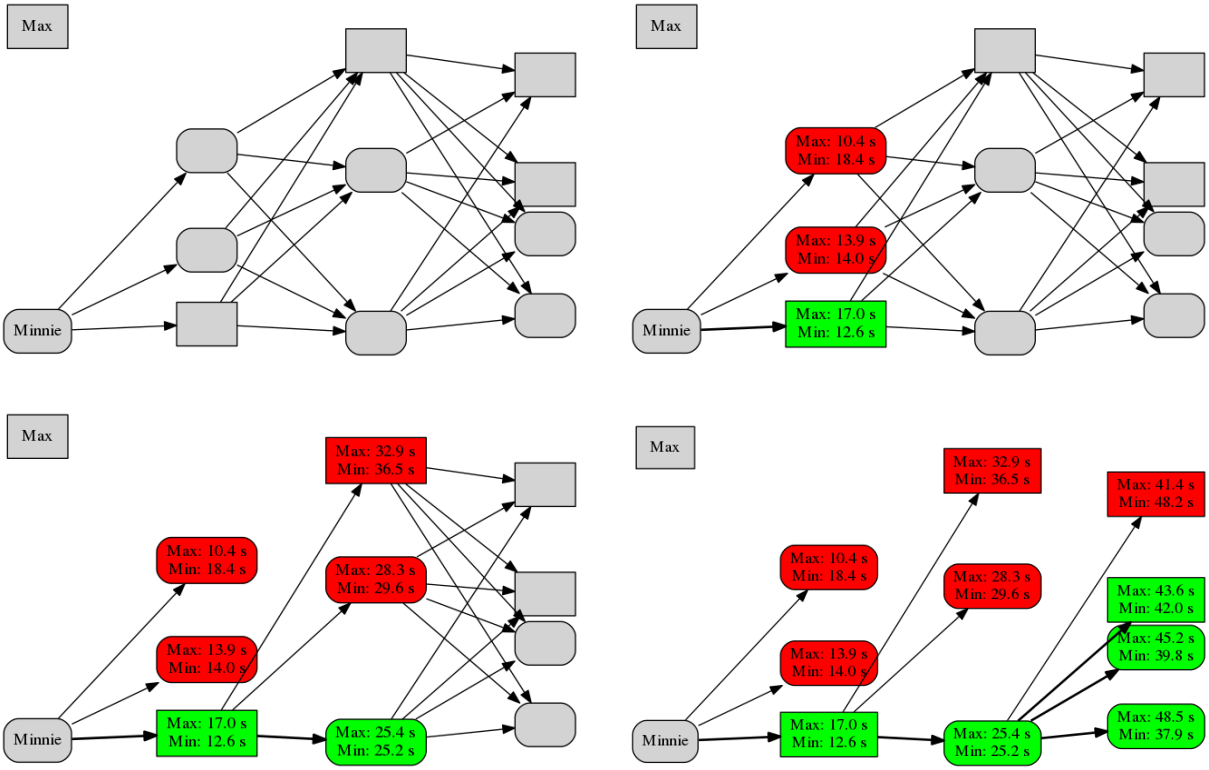


Abbildung 6: Veranschaulichung von Minnies Algorithmus. Der Übersicht halber wurde hier auf die Beschriftung der Kanten verzichtet. Zunächst wird für jedes Loch der zweiten Buhne die Zeit berechnet, die Minnie zum Erreichen braucht, und mit Max' Zeit verglichen. Ist Minnies Zeit kürzer, ist das Buhnenloch sicher (grün), sonst unsicher (rot) (oben rechts). Anschließend wird für jedes Loch der dritten Buhne dasjenige sichere Loch der zweiten Buhne gesucht, von dem aus Minnie am schnellsten das Loch erreichen kann und der Weg markiert (dicke Kanten) (unten links). Dies wird für die weiteren Buhnen wiederholt (unten rechts). Einen sicheren Weg erhält man dann, wenn man von einem sicheren Loch der letzten Buhne den Verweisen rückwärts zur ersten Buhne folgt.

5.5 Laufzeitverhalten

Der Minnie-Algorithmus ist aufgrund der zusätzlich zu betrachtenden Löcher und des zusätzlichen Aufwands durch die Rekonstruktion des Weges für die Laufzeit entscheidend. Deshalb betrachten wir nur dessen Laufzeitverhalten etwas genauer. Wir gehen davon aus, dass wir b Buhnen und durchschnittlich n Löcher pro Buhne, also insgesamt $N = n \cdot b$ Löcher haben. Für jede der b Buhnen abzüglich der ersten Buhne müssen wir für alle durchschnittlich n Löcher der Buhne die Zeit zu jedem der durchschnittlich n Vorgängerbuhnenlöcher berechnen, wir benötigen also $(b-1) \cdot n \cdot n = (b-1) \cdot n^2$ Schritte. Die Rekonstruktion des Weges benötigt anschließend b Schritte, da wir nur den Verweisen zurück zur ersten Buhne folgen müssen. Die Gesamtlaufzeit ist also beschränkt durch $\mathcal{O}((b-1) \cdot n^2 + b) = \mathcal{O}(b \cdot n^2) = \mathcal{O}(Nn)$.

Dieses gute Laufzeitverhalten zeigt sich auch bei der Lösung der Beispiele: So liegt auch im aufwendigsten, 14. Beispiel die Laufzeit bei deutlich unter einer Sekunde.

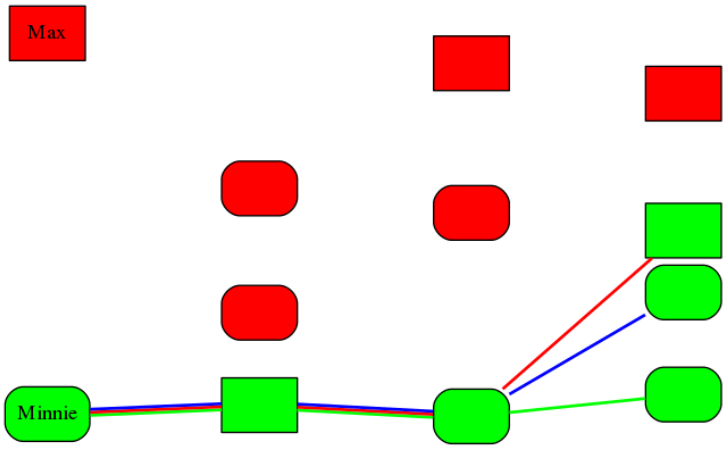
Beim Lösen der Aufgabe könnte man auch dazu neigen, alle möglichen Minnie-Pfade zu erzeugen und aus diesen einen beliebigen auszuwählen. Das ist aber ungleich aufwendiger: Wir starten mit n möglichen Pfaden von Minnies Startloch zu jedem der n Löcher in der zweiten Buhne. Gehen wir weiter zur dritten Buhne, müssen wir für jeden der n bereits vorhandenen Pfade n neue erzeugen, die zu den jeweils n Löchern der dritten Buhne führen, haben also schon n^2 Pfade erzeugt. Nach b Buhnen haben wir daher n^{b-1} Pfade erzeugt, also exponentiell viele. Zwar können wir einige der Pfade gleich verwerfen, da sie unsicher sind, am exponentiellen Wachstum der Komplexität ändert dies aber nichts. Für kleinere Beispiele mit wenigen Buhnen ist dies noch kein Problem, aber die größeren Beispiele lassen sich mit einer solchen Methode nicht lösen.

5.6 Lösungen der Beispiele

Tatsächlich gibt es in jedem BWINF-Beispiel mindestens einen sicheren Minnie-Pfad. Bei den kleineren Beispielen geben wir alle möglichen Pfade an, die für Minnie sicher sind. Bei den größeren Beispielen explodiert die Zahl der Möglichkeiten derart, dass wir stattdessen nur jeweils zwei Pfade angeben: nämlich den sichersten und den unsichersten Pfad, d.h. die Pfade, bei denen Minnie am Ende mit dem größten bzw. kleinsten Vorsprung herauskommt. Bei den sehr großen Beispielen ist sogar die Bestimmung aller Pfade so aufwendig, dass wir stattdessen nur den von unserem Algorithmus ausgegebenen Pfad angeben können.

Beispiel 1

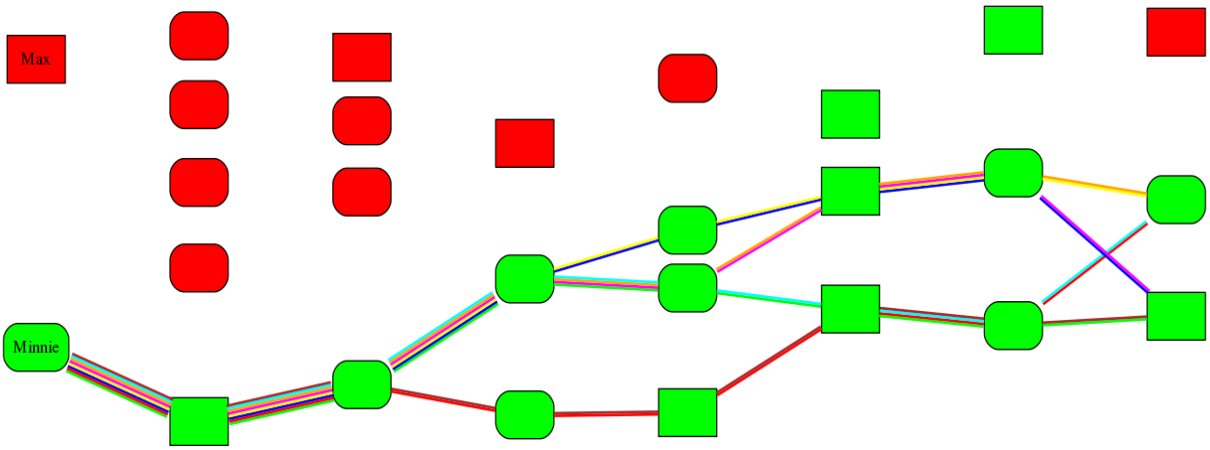
- Grafik



- Gesamtzahl Lösungswege: 3
- Sicherster Weg (grün): 10.6 s Vorsprung (Gesamtlaufzeit 37.9 s)
- Gefährlichster Weg (rot): 1.6 s Vorsprung (Gesamtlaufzeit 42.0 s)

Beispiel 2

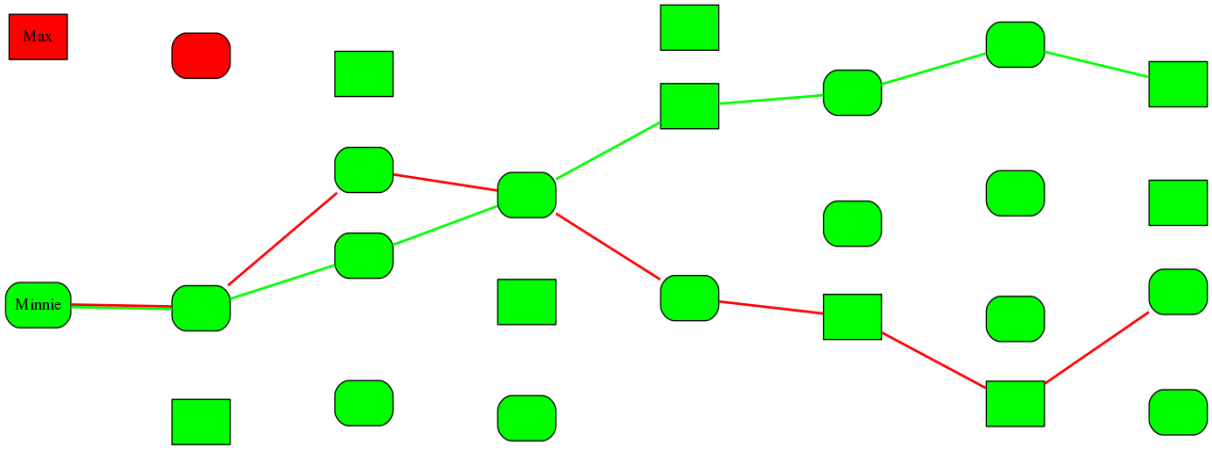
- Grafik



- Gesamtzahl Lösungswege: 8
- Sicherster Weg (grün): 9.2 s Vorsprung (Gesamtlaufzeit 91.5 s)
- Gefährlichster Weg (rot): 2.5 s Vorsprung (Gesamtlaufzeit 94.1 s)

Beispiel 3

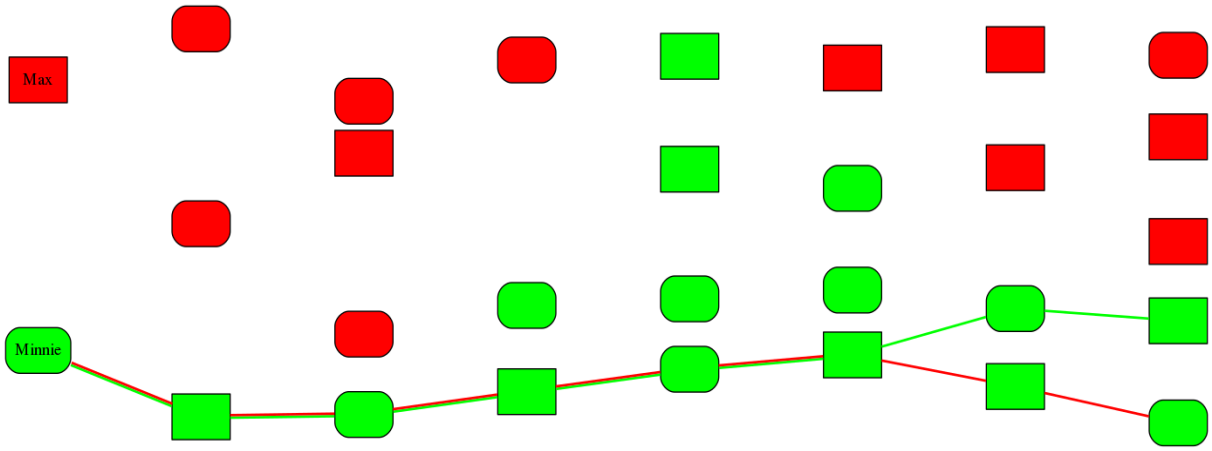
- Grafik



- Gesamtzahl Lösungswege: 192
- Sicherster Weg (grün): 13.3 s Vorsprung (Gesamtlaufzeit 91.8 s)
- Gefährlichster Weg (rot): 0.0 s Vorsprung (Gesamtlaufzeit 97.6 s)

Beispiel 4

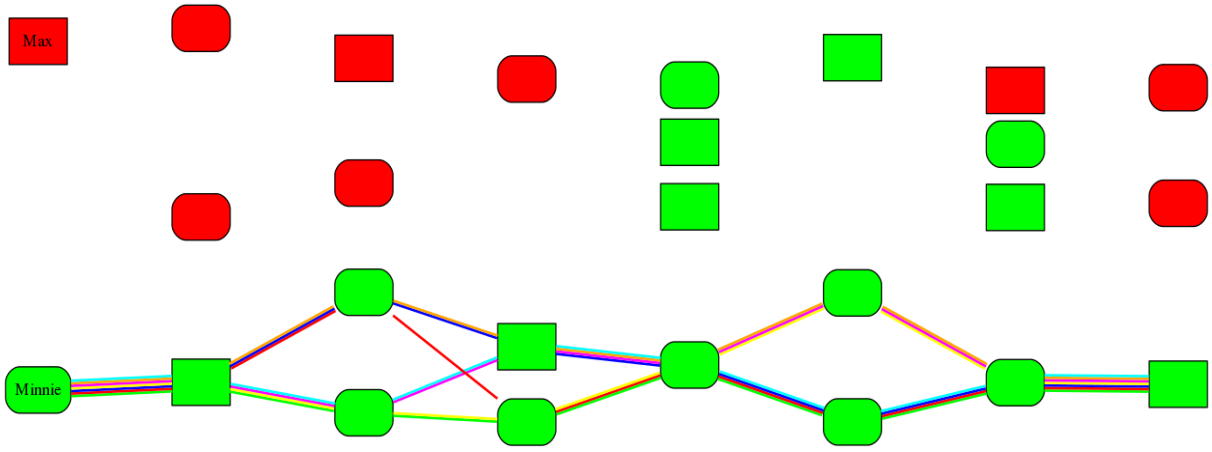
- Grafik



- Gesamtzahl Lösungswege: 2
- Sicherster Weg (grün): 0.5 s Vorsprung (Gesamtlaufzeit 89.7 s)
- Gefährlichster Weg (rot): 0.1 s Vorsprung (Gesamtlaufzeit 89.8 s)

Beispiel 5

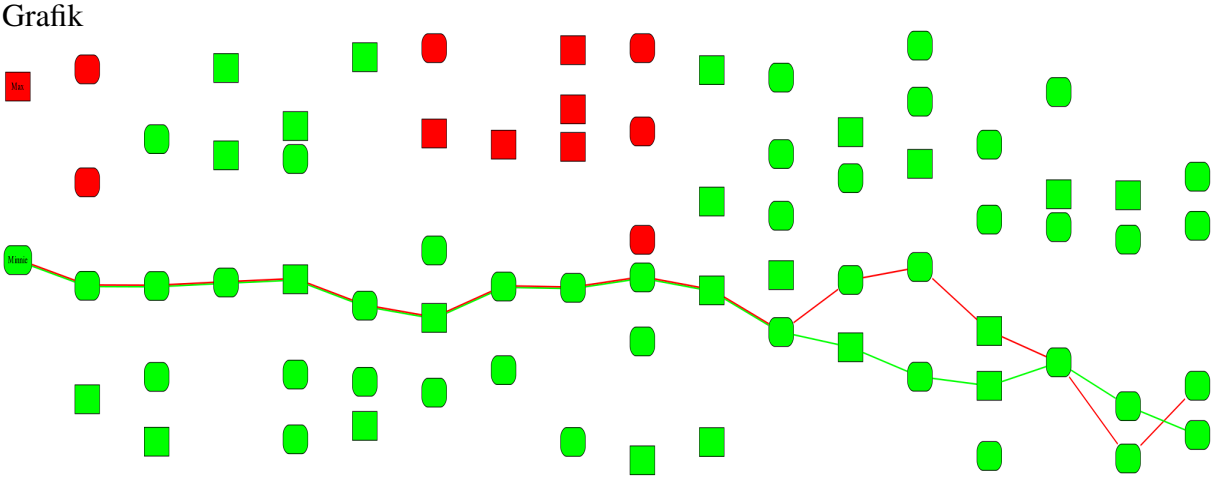
- Grafik



- Gesamtzahl Lösungswege: 7
- Sicherster Weg (grün): 5.4 s Vorsprung (Gesamtlaufzeit 89.9 s)
- Gefährlichster Weg (rot): 1.2 s Vorsprung (Gesamtlaufzeit 94.1 s)

Beispiel 6

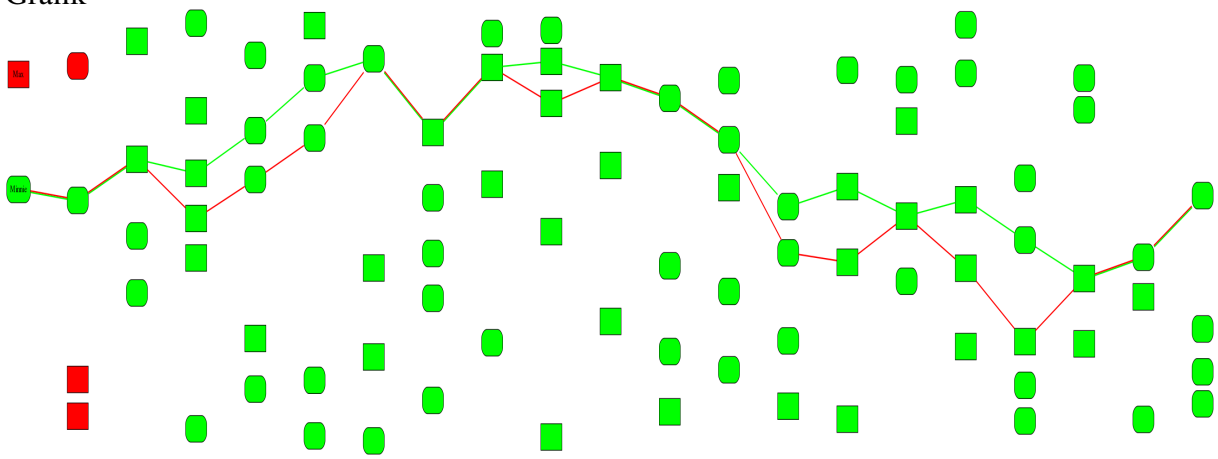
- Grafik



- Gesamtzahl Lösungswege: 60
- Sicherster Weg (grün): 11.4 s Vorsprung (Gesamtlaufzeit 218.6 s)
- Gefährlichster Weg (rot): 0.1 s Vorsprung (Gesamtlaufzeit 226.6 s)

Beispiel 7

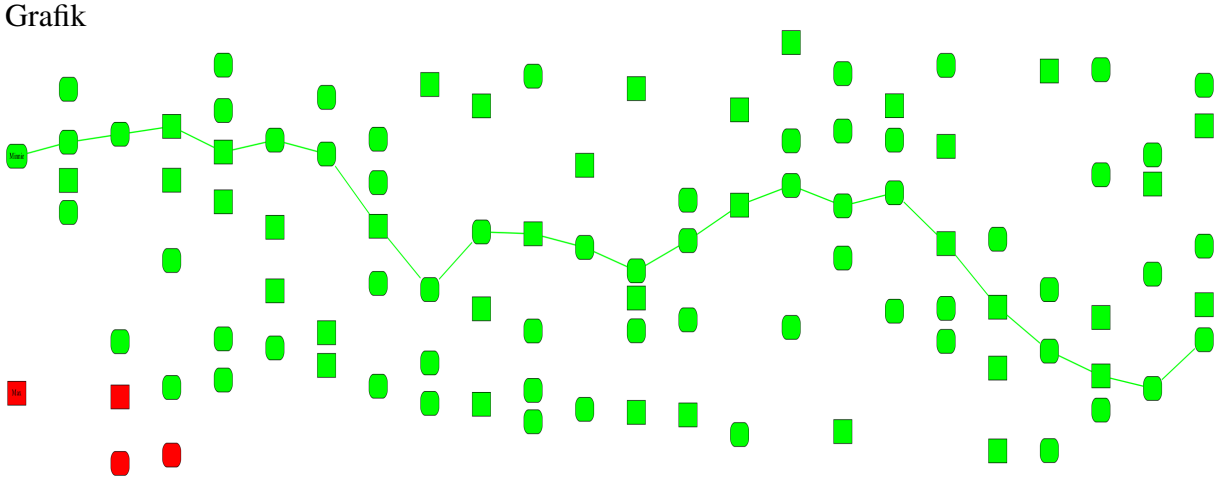
- Grafik



- Gesamtzahl Lösungswege: 156254
- Sicherster Weg (grün): 13.9 s Vorsprung (Gesamtlaufzeit 269.9 s)
- Gefährlichster Weg (rot): 0.0 s Vorsprung (Gesamtlaufzeit 283.9 s)

Beispiel 8

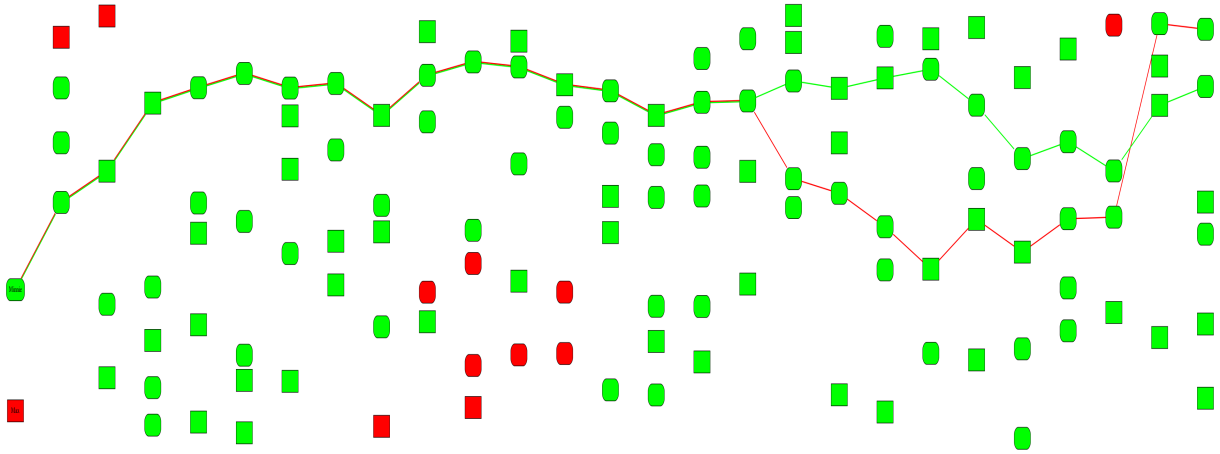
- Grafik



- Gesamtzahl Lösungswege: 24484257
- Sicherster Weg (grün): 80.7 s Vorsprung (Gesamtlaufzeit 309.7 s)
- Gefährlichster Weg: 0.0 s Vorsprung (Gesamtlaufzeit 384.0 s)

Beispiel 9

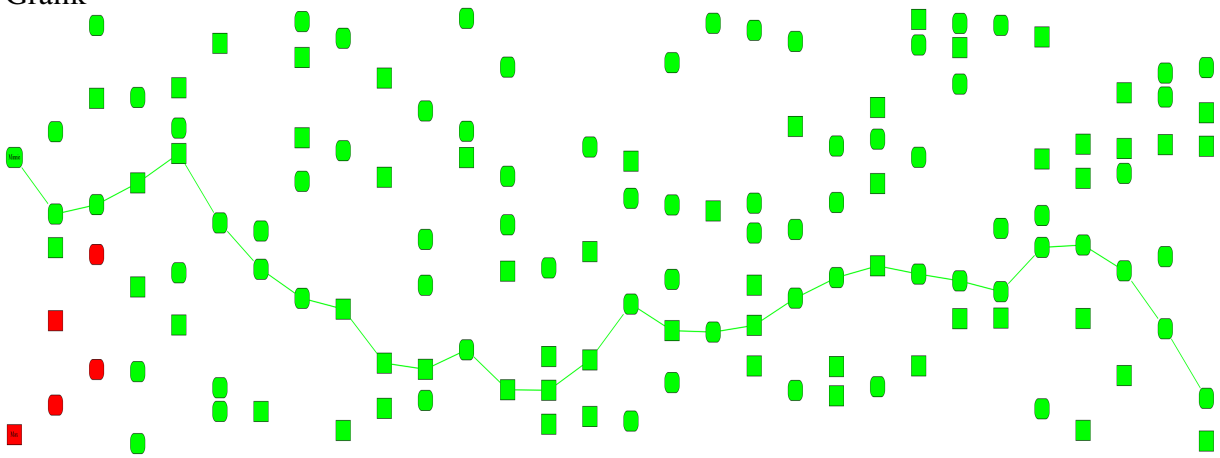
- Grafik



- Gesamtzahl Lösungswege: 1998
- Sicherster Weg (grün): 18.3 s Vorsprung (Gesamtlaufzeit 349.6 s)
- Gefährlichster Weg (rot): 0.0 s Vorsprung (Gesamtlaufzeit 370.5 s)

Beispiel 10

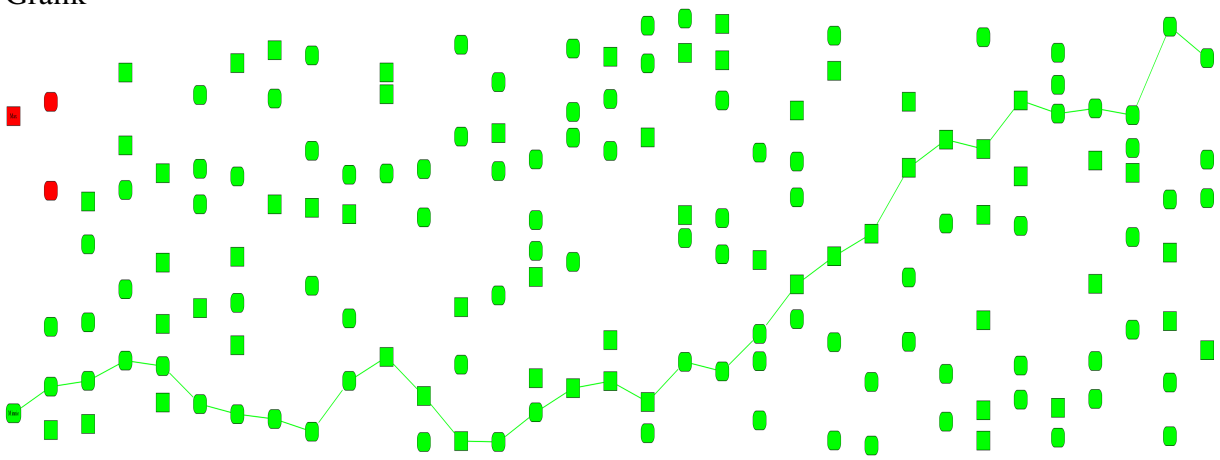
- Grafik



- Gesamtzahl Lösungswege: sehr, sehr viele
- Sicherster Weg (grün): 75.8 s Vorsprung (Gesamtlaufzeit 397.4 s)

Beispiel 11

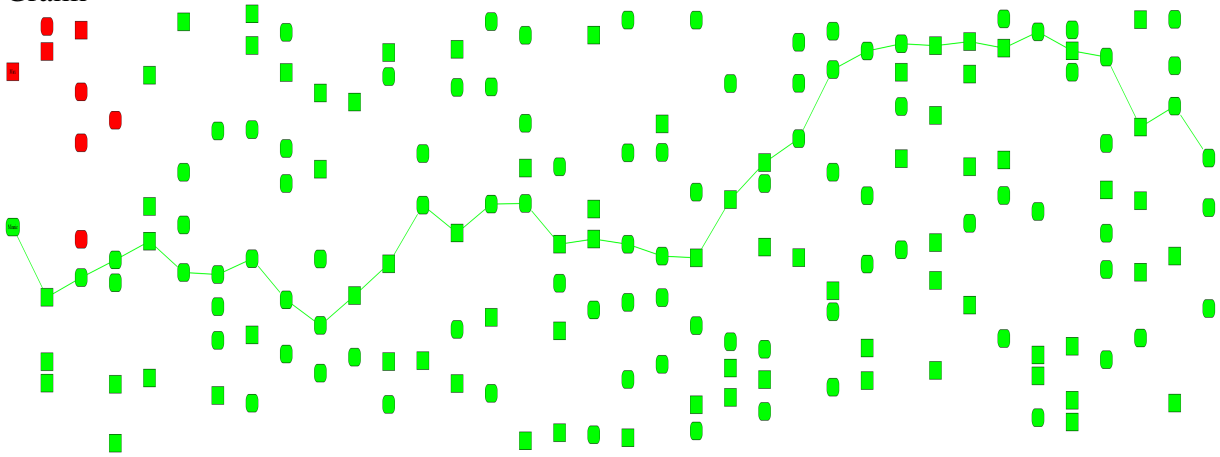
- Grafik



- Gesamtzahl Lösungswege: sehr, sehr viele
- Sicherster Weg (grün): 35.2 s Vorsprung (Gesamtlaufzeit 439.8 s)

Beispiel 12

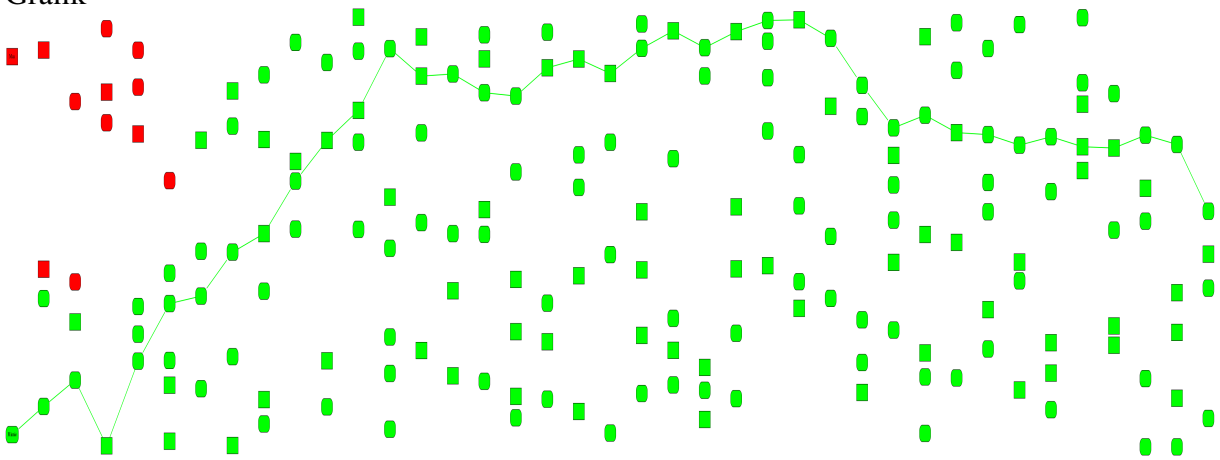
- Grafik



- Gesamtzahl Lösungswege: sehr, sehr viele
- Sicherster Weg (grün): 102.0 s Vorsprung (Gesamtlaufzeit 484.9 s)

Beispiel 13

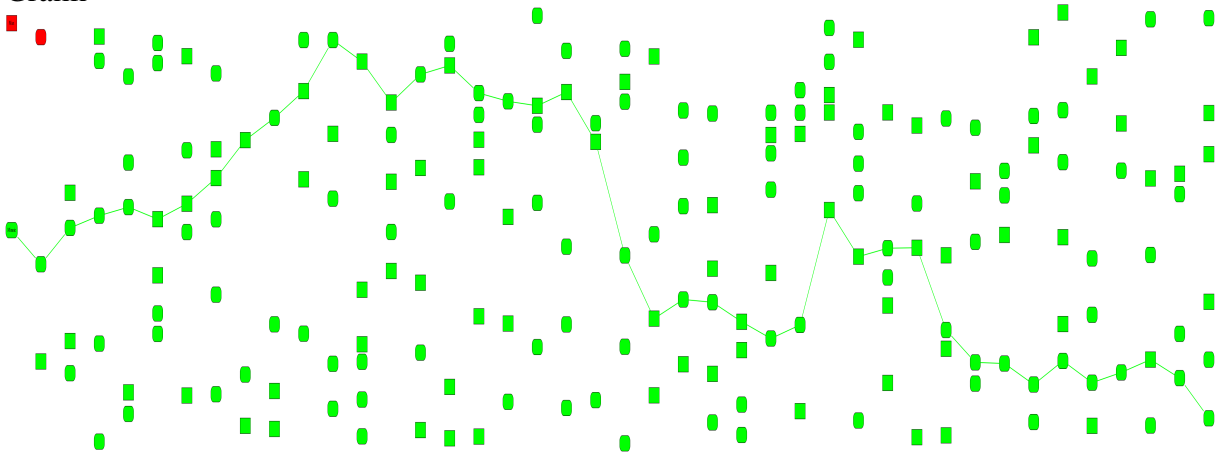
- Grafik



- Gesamtzahl Lösungswege: sehr, sehr viele
- Sicherster Weg (grün): 74.0 s Vorsprung (Gesamtlaufzeit 532.5 s)

Beispiel 14

- Grafik



- Gesamtzahl Lösungswege: sehr, sehr viele
- Sicherster Weg (grün): 91.2 s Vorsprung (Gesamtlaufzeit 597.1 s)

5.7 Bewertungskriterien

- Die Funktionsweise des Verfahrens soll nachvollziehbar erläutert werden.
- Wie zu Beginn beschrieben, genügt es zu bestimmen, ob Max Minnie an den Buhnenlöchern abfangen kann. Das sollte erkannt worden sein. Ein mögliches Abfangen zwischen den Buhnen zu berechnen ist unnötig kompliziert.
- Das benutzte Verfahren soll korrekt sein. Dazu sollte insbesondere erkannt worden sein, was es heißt, dass ein Minnie-Weg sicher ist, und nachvollziehbar begründet sein, wieso das gewählte Verfahren korrekt ist.
- Das Verfahren darf nicht zu ineffizient sein: Es muss in der Lage sein, alle vorgegebenen Beispiele in vernünftiger Zeit zu lösen. Ein stumpfes Durchprobieren aller Wege führt also in der Regel zu Punktabzug.
- Das Verfahren soll auch in der Lage sein zu erkennen, wenn es keinen Lösungsweg gibt und dies geeignet ausgeben (mit einer Exception abzustürzen ist keine geeignete Art!). Da unter den BWINF-Beispielen keine gänzlich unsicheren sind, sollte dieser Fall idealerweise mit einem eigenen Beispiel getestet werden.
- Die Lösung von etwa fünf der vierzehn vorgegebenen Beispiele soll dokumentiert sein, d.h. es soll jeweils ein Minnie-Weg angegeben sein.
- Die Ausgabe des Lösungswegs soll gut zu lesen sein. Eine einfache Liste der jeweiligen Nummer der Buhne mit den Positionen der genutzten Löcher ist ausreichend. Eine grafische Ausgabe ist nicht erforderlich.

Aus den Einsendungen: Perlen der Informatik

Allgemeines

Worte des Wettbewerbs: Alkorythmus, Vorschleife, Bruth-force

Codeschnipsel: `plus++ ;`

Unsere Idee war es, erst einmal das Programm manuell zu schreiben.

Die Lösungsidee wird in ein Java implementiert.

Die Aufgabe besteht aus zwei Aufgabenteilen, die ich in ein Programm zusammengefasst und durch eine deutliche Linie voneinander getrennt habe.

Hinweis vom betreuenden Lehrer: Die jar-Datei muss in der Konsole gestartet werden, da Patrick jede Ausgabe über die Konsole macht.

Ich habe keine Zeit mehr! Ich hoffe der Code ist Kommentar genug.

Das Programm ist ein Selbstläufer und muss über die Datei "Hauptmethode"gestartet werden.

Da mein Programm ohne eine Eingabe rechnet, ist die Komplexität stets die gleiche.

Luftballons

VERBACKEN()

LAMA

Leider keine Perlen. Das Programmieren von LAMAs ist offensichtlich eine ernste Angelegenheit.

Sprichwort

Schaltage, Schalljahr

```
#define TOTALAPOCALYPSE 900000000
```

Grundlage dieses Programms ist die Benutzeroberfläche. *Hm, wie kann eine Oberfläche gleichzeitig Grundlage sein?*

Osten kann nur im März oder April liegen.

Bis zum Jahr $9 \cdot 10^8$ wird sich die Sonne kontinuierlich zu einem roten Riesen entwickeln und jegliches höhere Leben auf der Erde zerstören. Daher ist der Suchraum (glücklicherweise ??) endlich.

Bei meinem Programm ist keine besondere Speicherkomplexität vorhanden.

Wie zu sehen ist, wird Christian es vermutlich doch nicht mehr selber erleben, außer es gibt noch ein paar besondere neue Erkenntnisse in der Medizin.

Mein Computer war dennoch zu langsam, um das zu berechnen, vielleicht liegt es aber auch am Code.

Rhinozelfant

Rhizonefant, Rhinalzelfant, Rhinzozelfant

Durch Magie werden die Pixel eingefärbt.

... kann es ... sehr schwierig werden, Rhinozelfanten zu erkennen, wenn der Anteil der weißen Pixel im Bild schon von der Verarbeitung ziemlich hoch ist. Im Winter sollte man sich also mit der Beobachtung pinker, flauschiger Einhörner zufrieden geben."

Ein Rhinozelfant in voller Pracht, bitte nicht füttern!

Damit die Pixel weiß gefärbt werden, welche nebeneinander liegen, wird das Bild spaltenweise von links nach rechts durchlaufen.

Die Bilder, die ausgegeben wurden, waren ein Feuerwerk der Farbenvielfalt.

Rotation

Breath-First Search

Da der Computer nicht denken kann, lassen wir ihn einfach alle Möglichkeiten, wie man das Feld nacheinander drehen könnte, ausprobieren.

Dies sollte allerdings nahezu niemals einen negativen Einfluss auf das Ergebnis haben. *Dann ist „dies“ wohl nicht der Grund für die falschen Ergebnisse.*

Die größte Schwierigkeit ist natürlich das Einlesen des Textdokumentes, was heißen soll, dass ich daran gescheitert bin.

```
int unendlichkeit = 20;
```

Und trotz der großen Zeitdauerschwankungen kann festgestellt werden, dass die Laufzeitkomplexität überschaubar ist.

Radfahrspaß

Das Ergebnis für Parcours 9 ist nur eine Vermutung, da die Schulstunde für das Testen leider nicht ausreichte.

Diese drei Zahlen sollte sich jeder Fahrer, welchem der Gewinn ernst ist, problemlos merken können.

Buhnenrennen

Buchenreihen

Max hat kein Problem damit, mit 30km/h in eine Wand zu rennen.

Dies ist offensichtlich sehr praktisch für den Hund, da er dann später gefressen wird.

Dazu stellen wir uns einen Grafen vor. *Aber welchen: Graf Dracula, Graf Zahl, ... ?*

Disclaimer: Wir haben bereits 4 Aufgaben gelöst, nehmen sie diese bitte nicht allzu ernst.

```
if (!alive) {  // Falls Minnie gefangen wurde:
  cout << ...; // Es ihrer Familie mitteilen
  return;      // und weinend nach Hause laufen.
}
```

Ich muss auch ehrlich zugeben, dass viele Teile des Programms nicht wirklich elegant gelöst sind. [...] Ich muss das Programm ja nicht nochmal durchlesen.