# Lösungshinweise und Bewertungskriterien

## Allgemeines

Das Wichtigste zuerst: Wir haben uns sehr darüber gefreut, dass einmal mehr so viele Leute sich die Mühe gemacht und die Zeit zur Bearbeitung der Aufgaben genommen haben.

Natürlich waren nicht alle Einsendungen perfekt, und einige eher äußerliche Anforderungen wurden häufiger missachtet. Im Einzelnen:

- Lösungsidee, Programm-Dokumentation und insbesondere auch Programm-Ablaufprotokolle und ausreichend viele Beispiele müssen ausgedruckt sein. Wir können aus Zeit- und Kostengründen keine Ausdrucke machen und auch nicht jedes eingesandte Programm ausführen.
- Noch schlechter als Einsendungen nur auf Datenträgern wären für uns übrigens Einsendungen via E-Mail oder anderen Internet-Wegen, auch wenn das für die Teilnehmer noch so praktisch wäre. Papiereinsendungen sind (zumindest zur Zeit und sicher auch noch in den nächsten Jahren) einfach unumgänglich.
- Beispiele werden als Teile des Programm-Ablaufprotokolls immer erwartet. Zu wenige Beispiele und erst recht die Nichtbearbeitung vorgegebener Beispiele führen zu Punktabzug. Es ist auch nicht ausreichend, Beispiele nur auf CD abzugeben, ins Programm einzubauen oder den Bewertern das Erfinden und Testen von Beispielen zu überlassen. Ohne abgedruckte Beispiele ist die Bewertung einer Lösung in der knappen vorhandenen Zeit nicht möglich. Leider fehlten in vielen Einsendungen die Beispiele, was oft das Erreichen der zweiten Runde verhindert hat.
- Zu einer Einsendung gehören auch lauffähige Programme. Kompilierung von Quellcode ist während der Bewertung nicht möglich. Für die gängigsten Skript-Sprachen stehen Interpreter zur Verfügung. Ohne die Abgabe eines eigenständig ausführbaren Programms

ist die Bewertung oft schwierig. Denken Sie insbesondere daran, wenn Sie eine Entwicklungsumgebung wie BlueJ benutzen, bei der die erstellten Programme ohne Weiteres nur innerhalb der Umgebung selbst laufen.

So, vielleicht denken Sie ja an diese Anmerkungen, wenn Sie (hoffentlich) im nächsten Jahr wieder mitmachen.

Auch die folgenden eher inhaltlichen Dinge sollten Sie beachten:

- Lösungsideen sollten Lösungsideen sein und keine Bedienungsanleitungen oder Wiederholungen der Aufgabenstellung. Es soll beschrieben werden, welches Problem hinter der Aufgabe steckt und wie dieses Problem grundsätzlich angegangen wird. Ein einfacher Grundsatz: Bezeichner von Programmelementen wie Variablen, Prozeduren etc. dürfen nicht verwendet werden – eine Lösungsidee ist nämlich unabhängig von solchen Realisierungsdetails.
- Auch ein Programmablauf-Protokoll soll keine Bedienungsanleitung sein. Es beschreibt nicht, wie das Programm ablaufen sollte, auch nicht die zum Ablauf nötigen Interaktionen mit dem Programm, sondern protokolliert den tatsächlichen, inneren Ablauf eines Programms. Am besten protokolliert ein Programm seinen Ablauf selbst, z. B. durch Herausschreiben von Eingaben, Zwischenschritten oder -resultaten und Ausgaben.
- Oben wurde schon gesagt, dass Beispiele immer dabei sein sollten, zumindest eines davon in einem Programm-Ablaufprotokoll. Das hat seinen Grund: An den Beispielen ist oft direkt zu sehen, ob bestimmte Punkte korrekt beachtet wurden. Viele meinen nun, wir könnten die Programme ja laufen lassen und selbst auf Beispieldaten ansetzen, und liefern keine Beispiele oder nur Beispieldaten in elektronischer Form. Das können wir aber aus Zeitmangel in der Regel nicht. Außerdem ist nicht immer sicher, dass Programme, die auf dem eigenen PC laufen, auch auf einem anderen Computer ausführbar sind. Generell muss man sich darauf einstellen, dass nur das Papiermaterial angesehen wird!
- Mit den verschiedenen Beispielen sollten Sie wichtige Varianten des Programmablaufs zeigen, also auch Sonderfälle, die Ihre Lösung behandeln kann. Die Konstruktion solcher Testfälle ist eine ganz wesentliche Tätigkeit des Programmentwurfs.

Einige Anmerkungen noch zur Bewertung:

- Pro Aufgabe werden maximal fünf Punkte vergeben, bei Mängeln gibt es entsprechend weniger Punkte. Für die Gesamtbewertung sind die drei am besten bewerteten Aufgabenlösungen maßgeblich, es lassen sich also maximal 15 Punkte erreichen. Einen 1. Preis erreichen Sie mit 14 oder 15 Punkten, einen 2. Preis mit 12 oder 13 Punkten und eine Anerkennung mit 9 bis 11 Punkten. Die Preisträger sind für die zweite Runde qualifiziert.
- Bei den Junioraufgaben gibt es auch je ein positiv formuliertes Bewertungskriterium. Bonuspunkte für diese Kriterien gibt es aber nur in der Juniorliga; dort können also maximal sechs Punkte pro Aufgabe und insgesamt maximal zwölf Punkte erreicht werden. In der Juniorliga wird ein 1. Preis für 10 bis 12 Punkte, ein 2. Preis für 8 oder 9 Punkte und eine Anerkennung für 6 oder 7 Punkte vergeben.

- Auf den Bewertungsbögen bedeutet ein Kreuz in einer Zeile, dass die (meist negative) Aussage in dieser Zeile auf Ihre Einsendung zutrifft. Damit verbunden ist dann in der Regel der Abzug eines oder mehrerer Punkte. Eine Wellenlinie bedeutet „na ja, hätte besser sein können“, führt aber alleine nicht zu Punktabzug. Mehrere Wellenlinien können sich aber zu einem Punktabzug addieren. Gelegentlich sind lobende Anmerkungen der Bewerter mit einem '+' versehen.
- Wellenlinien wurden übrigens häufig für die Dokumentation (Lösungsidee, Programm-Dokumentation, Programm-Ablaufprotokoll und kommentierter Programm-Text) verteilt, obwohl Punktabzug auch gerechtfertigt gewesen wäre.
- Aber auch so 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.

Zum Schluss:

- Sollte Ihr Name auf der Urkunde falsch geschrieben sein, können Sie gerne eine neue anfordern.
- Es ist verständlich, wenn jemand, der nicht weitergekommen ist, über eine Reklamation nachdenkt. Gehen Sie aber bitte davon aus, dass wir kritische Fälle, insbesondere die mit 11 Punkten, schon genau und mit Wohlwollen geprüft haben.

## Danksagung

An der Erstellung der Lösungsideen haben mitgewirkt: Robert Czechowski (Junioraufgabe 1), Jan Balzer (Junioraufgabe 2), Thomas Leineweber (Aufgabe 1), Martin Thoma (Aufgabe 2), Joachim Priesner (Aufgabe 3), Tim Kiefer und Julian Eberius (Aufgabe 4) sowie Toni Mattis und Torben Hagerup (Aufgabe 5).

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

# Junioraufgabe 1: Rechteckschoner

## J1.1 Lösungsidee

Die Aufgabe verlangt, dass der Bildschirmschoner Rechtecke anzeigt, deren Eigenschaften verändert werden können.

Daher bietet es sich an, das ganze objektorientiert anzugehen. Ein Rechteck ist eine Klasse, die als Eigenschaften zumindest die Position und die Größe des Rechtecks besitzt.

Zusätzlich verlangt die Aufgabe, dass die Rechtecke ineinander geschachtelt werden können. Dafür kombinieren wir die Klasse für unser Rechteck mit der Struktur eines binären Baums; das Rechteck besitzt bis zu zwei Kinderrechtecke.

Als Methoden verwenden wir eine zum Zeichnen des Rechtecks auf den Bildschirm und eine zum zufälligen Verändern des Rechtecks oder seiner Kinderrechtecke. Beide Funktionen werden rekursiv durch den ganzen Baum („zeichne“) oder durch Teile des Baums aufgerufen („aendere“).

Schließlich kann die Rechteckklasse noch nach Belieben um weitere Eigenschaften erweitert werden, die im Konstruktor zufällig gesetzt werden, um das Bild interessanter zu machen, wie in der Aufgabenstellung gefordert. Im Beispiel wurde die Farbe (zufälliger RGB-Wert) und die Liniendicke (zufällig, zwischen 1 und 3 Pixel) gewählt, es sind aber beliebige andere Modifikationen möglich.

Zur Konstruktion wird nun ein großes Rechteck angelegt, dass sich „von ganz alleine“ Kinderrechtecke mit zufälligen Eigenschaften erstellt und damit den ganzen Bildschirm füllt.

Die Aufgabenstellung sagt nun, dass die Rechtecke aktualisiert (also verändert) werden sollen. Dabei ist sehr viel Spielraum für eigene Interpretationen gelassen. Im hier konstruierten Fall kann man durch einen oder mehrere Äste des Baumes laufen, dann willkürlich Rechtecke löschen und durch neue ersetzen. Dadurch, dass die neuen Rechtecke sich selber wieder Kinderrechtecke erstellen, wird ein Teil des Bildes geändert.

Auch andere Herangehensweisen sind möglich. Es sollte jedoch darauf geachtet werden, dass sich nicht jedesmal alle Rechtecke ändern, sondern immer nur Teile des Bildes. Vor allem bei der Positionierung und Dimensionierung der Rechtecke ist darauf zu achten, dass Rechtecke sich nicht überlappen und keine Rechtecke negative Kantenlängen bekommen (außer natürlich, wenn das beabsichtigt ist und zu schönen Effekten führt.)

## J1.2 Programmdokumentation

Eine Rechteckklasse kann zum Beispiel so aussehen:

```
class Rechteck
{
public:
    Rechteck(int x,int y, int breite, int hoehe);
    ~Rechteck();

    void aendere();
    void zeichne(QPainter*);

    void erzeugeKinder();
    void vernichteKinder();

private:
    // Elementare Eigenschaften
    int breite, hoehe;
    int x, y;

    // Baumstruktur
    Rechteck *kind1, *kind2;

    // Weitere Eigenschaften
    QColor color;
    int liniendicke;
};
```

Bei Konstruktion eines neuen Rechtecks werden zunächst die Eigenschaften zufällig initialisiert. Danach werden die Kinderrechtecke nach folgenden Regeln erstellt:

1. Das Rechteck wird an der größeren Kantenlänge geteilt.
2. Das Rechteck wird so geteilt, dass jeder der Teile mindestens ein Viertel der Gesamtgröße erhält.
3. Die beiden Kinderrechtecke werden so in den Teilen platziert, dass zwischen allen Kanten ein bestimmter Abstand (hier 5 Pixel) eingehalten wird.

Sollte die kleinere Kantenlänge einen bestimmten Wert unterschreiten (20 Pixel), so werden keine Kinderrechtecke erstellt.

```
const int margin = 5; int minLaenge = 20;
...
void Rechteck::erzeugeKinder()
{
    if (hoehe > breite)
    {
        if (breite < minLaenge) return; // Wenn Rechteck zu klein,
           abbrechen

        // Zufällige Höhe des ersten Rechtecks
        int r = rand() % (hoehe-3*margin)/2 + (hoehe-3*margin)/4;

        // Rechtecke anlegen
        kind1 = new Rechteck(x+margin, y+margin, breite - 2*margin, r);
        kind2 = new Rechteck(x+margin, y+r+2*margin, breite-2*margin,
           hoehe-r-3*margin);
```

```
    }
    else
    {
        if (hoehe < minLaenge) return; // Wenn Rechteck zu klein,
            abbrechen

        // Zufällige Breite des ersten Rechtecks
        int r = rand() % (breite-3*margin)/2 + (breite-3*margin)/4;

        // Rechtecke anlegen
        kind1 = new Rechteck(x+margin, y+margin, r, hoehe-2*margin);
        kind2 = new Rechteck(x+r+2*margin, y+margin, breite-r-3*margin,
            hoehe-2*margin);
    }
}
```

Um den Quellcode zu verdeutlichen, zeigt Abbildung 1 ein Bild mit den Maßen eines Rechtecks:

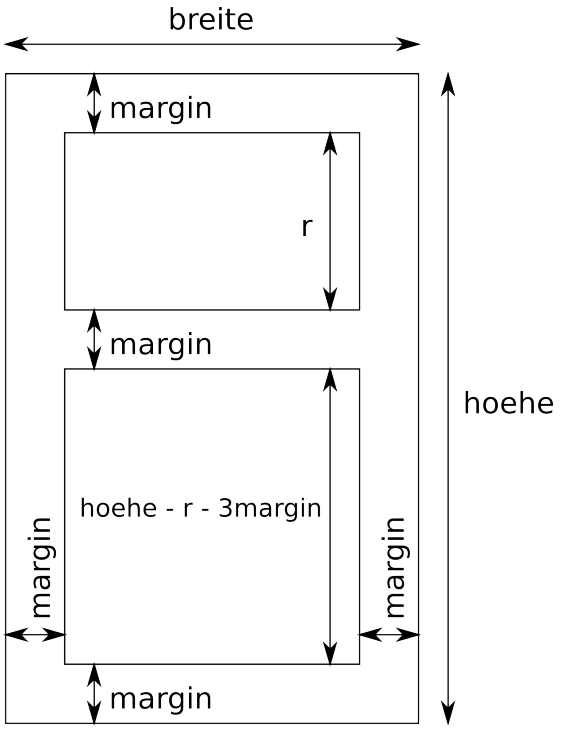


Abbildung 1: Rechteckmaße

## J1.3 Programmablaufprotokoll

Die Ausgabe kann in etwa so aussehen wie in den Abbildungen (aber auch völlig anders). Für einen Bildschirmschoner bietet sich vor allem ein schwarzer Hintergrund an, aber hier wurde ein weißer Hintergrund gewählt, um beim Ausdrucken Farbe zu sparen.

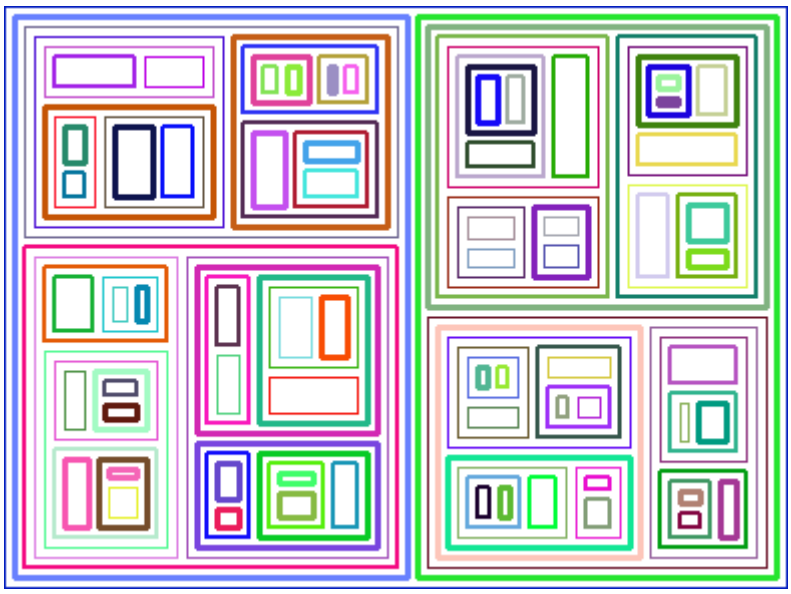

Abbildung 2: Beispiel

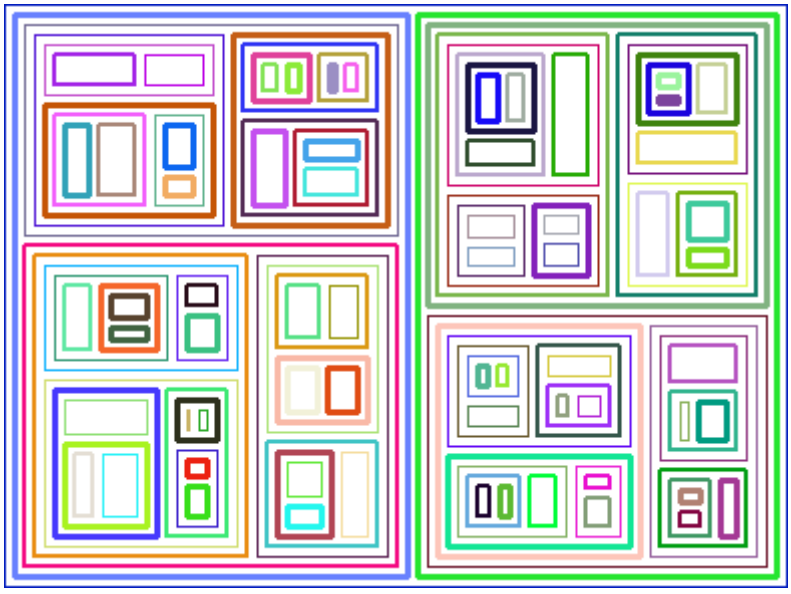

Abbildung 3: Nach der ersten Aktualisierung

Hier ist nur ein Beispiel (Abbildung 2) mit zwei Aktualisierungen: Bei der ersten Aktualisierung (Abbildung 3) sieht man Unterschiede sowohl im Kästchen links oben als auch links unten. Bei der zweiten Aktualisierung (Abbildung 4) sieht man links oben, rechts oben und rechts unten Unterschiede.

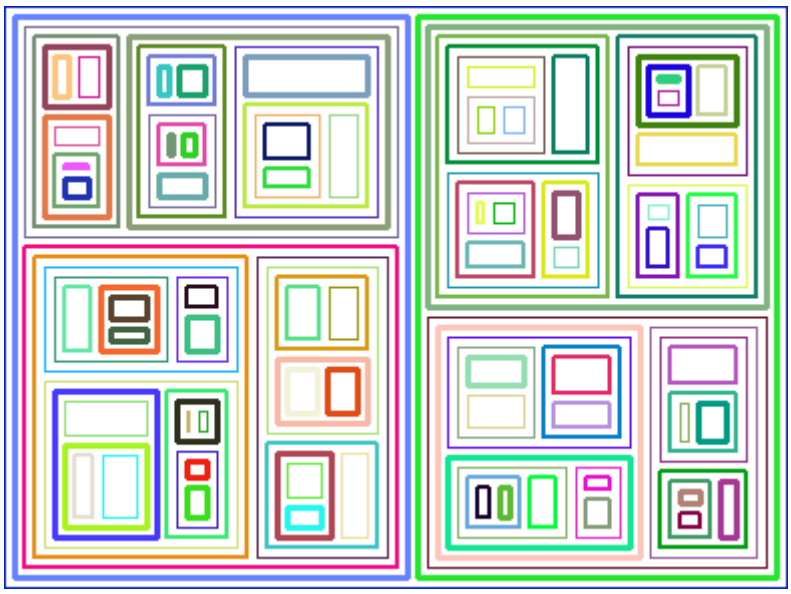

Abbildung 4: Nach der zweiten Aktualisierung

## J1.4 Bewertungskriterien

- Die Rechtecke sollen fehlerfrei ineinander geschachtelt werden, sich also nicht überlappen, „negative“ Kantenlänge haben oder anderweitig fehlerhaft platziert sein.
- Beim Zeichnen der Rechtecke soll nicht nur die Größe, sondern auch andere Eigenschaften der Rechtecke variiert werden (z.B. Farbe, Dicke, etc.). Dies kann in jedem Schritt zufällig, aber auch systematisch, z.B. in Abhängigkeit von der Rekursionstiefe passieren.
- Jedes erzeugte Rechteck muss auch aktualisierbar sein. Entscheidend ist also, dass die Rechtecke so verwaltet werden, dass auf jedes zugegriffen werden kann. Die Aktualisierung kann durch Neuzeichnen erledigt werden (nur des Rechtecks selbst oder auch aller enthaltenen Rechtecke) und soll nur einen Teil (oder evtl. mehrere) des Bildes betreffen – jedenfalls nicht das gesamte Bild.
- Es müssen genügend Beispiele vorhanden sein. Die Aufgabenstellung verlangt drei verschiedene Bilder mit jeweils zwei Veränderungen. Weniger ist dann akzeptabel, wenn trotzdem ersichtlich wird, dass die Lösung funktioniert.
- Bonuspunkt: Die Rechtecke werden mit Hilfe einer strukturell rekursiven Datenstruktur verwaltet, die einen Baum realisiert und besonders flexible Änderungen erlaubt.

# Junioraufgabe 2: Glücksrad

## J2.1 Einfache Lösung

Um den Millionengewinn zu bekommen, muss der Spieler jedes Feld des Rads genau einmal besuchen. Deswegen können wir ein erfolgreiches Spiel als eine beliebige Anordnung der Felder bzw. Buchstaben betrachten, z. B. *ACDFEB*. Die Wahrscheinlichkeit für eine Folge ist das Produkt der Einzelwahrscheinlichkeiten, von einem Feld zum nächsten zu kommen, wie es in der Aufgabenstellung beschrieben ist. Unser Programm könnte also alle möglichen Folgen durchrechnen und die Wahrscheinlichkeiten aufsummieren, um die Gesamtwahrscheinlichkeit für den Millionengewinn zu erhalten.

Aus der Kombinatorik weiß man, dass sich die Anzahl der möglichen Anordnungen von $n$ Elementen durch die Fakultät $n! = n \times (n-1) \times \cdots \times 2 \times 1$ berechnen lässt. Für ein Glücksrad mit sechs Feldern ergeben sich so $6! = 720$ gültige Folgen. Da es maximal 10 Felder gibt, muss unser Programm höchstens $10! = 3\,628\,800$ verschiedene Gewinnfolgen betrachten. Außerdem muss das Programm die Wahrscheinlichkeit jeder Gewinnfolge berechnen, was pro Folge $(n-1)$ Multiplikationen entspricht. Die Komplexität (Laufzeit) unseres Programms bewegt sich also im Bereich $n!$, was bei der kleinen Eingabegröße[1] noch in Ordnung ist.

Für das Durchrechnen aller Gewinnfolgen bietet sich ein rekursiver Algorithmus an. Wir beginnen mit der Anfangssituation, bei der alle Felder noch nicht besucht wurden. Von da aus ruft sich die rekursive Methode für jedes bisher noch freie Feld selbst auf. Bei diesem Ansatz werden einige Multiplikationen eingespart, deswegen benötigt das Programm nicht ganz $n!$ Schritte. Die tatsächliche Anzahl liegt zwischen $(n-1)!$ und $n!$ und beträgt $\Sigma_{k=0}^{n-1} \frac{(n-1)!}{k!}$. Die Analyse ersparen wir dem Leser – es ist sinnvoller, sich mit der effizienteren Lösung weiter unten zu beschäftigen ;).

Um bei einer Drehung um $k$ Felder das erreichte Feld richtig zu berechnen, muss man bedenken, dass nach dem letzten Feld des Glücksrads wieder das erste kommt, Feld A. Dies kann man durch Modulo-Rechnung bzgl. der Feldanzahl $n$ simulieren. Um nicht zu viel hin- und herzurechnen, bietet es sich an, die Felder ab 0 zu zählen, also $A = 0$, $B = 1$. Springt man z. B. bei $n = 6$ von Feld $D = 3$ aus fünf Felder weiter, so ist die neue Position $(3+5) \bmod 6 = 2$ (Feld $C$).

## J2.2 Effizientere Lösung

Auch wenn die oben beschriebene Lösung in akzeptabler Zeit das richtige Ergebnis liefert, sollte man sich doch fragen, ob man die Wahrscheinlichkeit für den Millionengewinn nicht *effizienter* berechnen kann. Um zu zeigen, wie dies funktioniert, führen wir zunächst den Begriff "Zustand" für das Glücksrad ein. Ein Zustand speichert, welche Felder noch besucht werden müssen, welche bereits besucht wurden und auf welchem Feld der Spieler aktuell steht. Eine kompakte Schreibweise dafür ist $AbC\hat{D}Ef$. In diesem Fall wurden die Felder $A, C, D$ und $E$

[1] Bei anderen Problemstellungen ist $n$ üblicherweise weitaus größer als 10, daher ist eine Laufzeit von $n!$ in der Regel sehr schlecht.

**Algorithmus 1** Einfacher Glücksrad-Algorithmus

```
n = [Anzahl der Felder]
felder[0..n-1] = (0, ..., 0)          ▷ gibt an, ob die Felder besucht sind oder nicht
wkeit[1..n] = [Wahrscheinlichkeiten, 1..n Felder weiterzudrehen]

Ergebnis = BERECHNE(n, 1, felder[n])

function BERECHNE(n, pos, felder[n])
    if felder[0..n-1] == [1..1] then
        return 1                                   ▷ wir haben schon gewonnen
    end if
    gesamt = 0
    for k = 1..n do                                ▷ iteriere über alle Sprungweiten
        pos' = (pos + k) mod n
                                                   ▷ besuche das neue Feld, falls es noch frei ist
        if felder[i] == 0 then
            felder[pos'] = 1                       ▷ markiere Feld als besucht
            gesamt = gesamt + wkeit[k] * BERECHNE(n, pos', felder[n])
            felder[pos'] = 0                       ▷ mache Änderung rückgängig
        end if
    end for
    return gesamt
end function
```

bereits besucht, der Spieler befindet sich auf Feld $D$ und die Felder $b$ und $f$ sind noch frei. Im nächsten Spielzug darf sich das Rad also nur um 2 (zu $f$) oder 4 (zu $b$) Felder weiterdrehen. Betrachten wir nun einen anderen Zustand, nämlich $AbCdE\hat{F}$. Hier sieht es ähnlich aus: nur ein Sprung von 2 oder 4 Felder führt zum Erfolg. Deshalb ist die Wahrscheinlichkeit, von diesen Zuständen aus den Millionengewinn zu erhalten, *genau gleich*. In einem effizienten Algorithmus betrachten wir diese Situation nur einmal, währen die oben beschriebene Lösung beide Zustände durchrechnet.

Als nächstes führen wir eine noch kompaktere Schreibweise für Zustände ein, um die vorherigen Überlegungen zu äquivalenten Zuständen mit einzubeziehen. Es genügt nämlich zu speichern, welche Felder vom aktuellen Feld aus gesehen frei sind. Die Beschriftung der Felder und die absolute Position kann weggelassen werden. So lassen sich die beiden Beispielzustände durch den Bitstring[2] 110101 darstellen: bereits besuchte Felder werden durch eine 1 gekennzeichnet, freie Felder durch eine 0. Der Spieler befindet sich dabei immer auf dem Feld ganz links – die Bitstrings beginnen also immer mit einer 1. Deswegen können wir auch die erste Stelle weglassen, unser Beispiel reduziert sich zu 10101. Ein Bitstring beschreibt daher genau, welche Art von Feldern – besetzt oder frei – auf die aktuelle Position folgen.

[2]Es ist kein Zufall, dass wir gerade Bitstrings wählen, da sich diese sehr effizient verarbeiten lassen. Daher lässt es sich nicht vermeiden, in den nächsten Absätzen etwas detailliert zu werden. Bedenkt, dass es in einer Lösungsidee nur dann sinnvoll ist, Details zu beschreiben, wenn diese ein wichtiger Teil der Lösung an sich sind.

Tabelle 1: Mögliche Beschriftungen für einen Bitstring

| (1) | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|
| $\hat{D}$ | E | f | A | b | C |
| $\hat{F}$ | A | b | C | d | E |

Da die Bitstrings nicht länger als 9 Bits sein können (die Glücksräder haben höchstens 10 Felder), können wir die Bitstrings bequem als *Integer* in unserem Programm codieren. Dreht sich das Glücksrad in einem Spielzug um $k$ Felder weiter, so ergibt sich der neue Bitstring wie folgt:

1. setze das $k$-te Bit (von links) auf 1
   z. B. 00101 zu 0**1**101 für $k = 2$
2. füge eine 1 vorne hinzu
   z. B. 01101 zu **1**01101
3. rotiere den Bitstring um $k$ Stellen nach links
   z. B. $\mathit{rotleft}(101101, 2) = 110110$
4. entferne das vorderste Bit
   z. B. 110110 zu 10110

In einigen Programmiersprachen gibt es vordefinierte Funktionen für die Rotation von Bitstrings. In anderen Sprachen, wie z. B. C oder C++, muss man dafür eine eigene Prozedur schreiben, welche die Rotation dann mit primitiven Bitoperationen simuliert.

Wir wissen nun, wie wir den aktuellen Zustand als Bitstring speichern können. In unserem modifizierten Algorithmus erledigt dies ein Array `P`: Wir übergeben dem Array einen als Integer codierten Index. So speichert beispielsweise `P[10101]`, also `P[21]`, die Wahrscheinlichkeit, von diesem Zustand aus den Millionengewinn zu erhalten. Wann immer ein Zustand generiert wird, schlägt das Programm zunächst im Array `P` nach, ob die Wahrscheinlichkeit bereits berechnet wurde. Ist dies der Fall, so wird die Rekursion an dieser Stelle beendet und das Ergebnis zurückgeliefert. Da es viele redundante Zustände gibt, werden viele Teilbäume des Rekursionsbaums eingespart. Man beachte, dass es für $n$ Felder $(n-1)!$ verschiedene Zustände gibt, die sich aber auf $2^{n-1}$ Bitstrings reduzieren lassen. Bei $n = 10$ ist dies immerhin ein Verhältnis von 362880 zu 512.

## J2.3 Ergebnisse

Erwartungsgemäß wird die Wahrscheinlichkeit, den Millionengewinn zu erhalten, kleiner, je mehr Felder es gibt. Interessante Ergebnisse erhält man auch, wenn man alle Sprungwahrscheinlichkeiten bis auf eine auf 0 setzt. Die Gewinnwahrscheinlichkeit ist dann entweder 1 oder 0, je nachdem, ob sich mit einer einzigen Sprungweite alle Felder nacheinander abgehen lassen oder nicht.

**Algorithmus 2** Besserer Glücksrad-Algorithmus

```
n = [Anzahl der Felder]
p[0..2^(n−1)]                          ▷ Wahrscheinlichkeiten, den Millionengewinn zu erhalten
wkeit[1..n] = [Wahrscheinlichkeiten, 1..n Felder weiterzudrehen]
start = 0                                          ▷ anfangs ist kein Feld außer A besucht

Ergebnis = BERECHNE(n, start)

function BERECHNE(n, b)                  ▷ der aktuelle Zustand ist im Bitstring b codiert
    gesamt = 0
    if b = 11...1 then
        return 1                         ▷ alle Felder besetzt, wir haben schon gewonnen
    end if
    for k = 1..n do                                  ▷ iteriere über alle Sprungweiten
                                        ▷ besuche das neue Feld, falls es noch frei ist
        if k-tes Bit von B ist 0 then
            setze k-tes Bit in b auf 1
            b' = 1b
            rotiere b' um k Felder nach links
            entferne vorderstes Bit bei b'
            if p[b'] existiert nicht then
                p[b'] = BERECHNE(n, b')
            end if
            gesamt = gesamt + wkeit[k] * p[b']
        end if
    end for
    return gesamt
end function
```

Tabelle 2: Testläufe und ihre Ergebnisse

| | *Wahrscheinlichkeiten* | | | |
|---|---|---|---|---|
| *Felder* | $(\frac{1}{n},\ldots,\frac{1}{n})$ | $(1,0,\ldots)$ | $(0,0,1,\ldots)$ | $(\frac{1}{2},\frac{1}{2},0,\ldots)$ |
| 1 | 1 | 1 | n.a. | n.a. |
| 2 | 0.5 | 1 | n.a. | 0.5 |
| 3 | 0.25 | 1 | 0 | 0.5 |
| 4 | 0.09375 | 1 | 1 | 0.25 |
| 5 | 0.0384 | 1 | 1 | 0.1875 |
| 6 | 0.015432 | 1 | 0 | 0.09375 |
| 7 | 0.006120 | 1 | 1 | 0.0625 |
| 8 | 0.002403 | 1 | 1 | 0.03125 |
| 9 | 0.00937 | 1 | 0 | 0.019531 |
| 10 | 0.000363 | 1 | 1 | 0.009766 |

Für das Beispiel aus der Aufgabenstellung mit sechs Feldern und den Sprungwahrscheinlichkeiten $(\frac{5}{15},\frac{4}{15},\frac{3}{15},\frac{2}{15},\frac{1}{15},0)$ ergibt sich eine Gewinnwahrscheinlichkeit von 0.04717. Tabelle 2 listet weitere Ergebnisse für verschiedene Wahrscheinlichkeitsverteilungen und unterschiedliche Anzahlen von Feldern auf.

## J2.4 Bewertungskriterien

Da es sich um eine Junioraufgabe handelt und die Eingabegröße sehr klein gewählt ist, reicht es aus, die einfache Lösung einzureichen. Bewertungskriterien für die einfache Lösung sind:

- Das Verfahren zur Berechnung der Gewinnwahrscheinlickeit muss korrekt sein; dazu sollte das Konzept der Multiplikation auf den Pfaden und der Addition der Pfadwahrscheinlichkeiten erkannt worden sein.
- Der Algorithmus sollte nachvollziehbar beschrieben werden, das Schlagwort „Rekursion“ alleine reicht nicht aus.
- Genügend Beispiele, auch eines für zehn Felder, sollten vorhanden sein.
- Die berechneten Werte sollten korrekt sein; Rundungsfehler sind aber akzeptabel.
- Bonuspunkt: Das realisierte Verfahren ist effizienter als die einfache Lösung (durch Ausnutzung der Äquivalenz verschiedener absoluter Positionen, durch die Verwendung von Bitoperationen etc.) und erlaubt, Ergebnisse für deutlich mehr als zehn Felder zu berechnen.

# Aufgabe 1: Wer spielt fair?

## 1.1 Anforderungen an ein Verfahren

Die Anforderungen aus der Aufgabenstellung werden zuerst genauer gefasst und um ein paar weitere Fragestellungen ergänzt.

Auf der einen Seite soll das Verfahren sicherstellen, dass das Programm fair spielt. Das bedeutet, dass sich das Programm vor jedem einzelnen Spielzug festlegt, wie es spielen wird. Diese Festlegung muss so geschehen, dass ein Spieler danach den gespielten Zug mit dem vorher festgelegten Zug vergleichen kann. Dabei muss der Spieler davon überzeugt sein, dass die hinterher offengelegte Festlegung nicht während des Spieles geändert worden ist.

Andererseits soll das Verfahren die Züge nicht einfach so vorab an den Spieler geben, dass er direkt sehen kann, welcher Zug als Nächstes kommen wird.

## 1.2 Vertrauenswürdiger Dritter (Trusted Third Party)

Wenn der Spieler einer dritten Person vertraut, kann das Programm die gewählten Züge an diese vertrauenswürdige dritte Person übermitteln, bevor die Spielereingabe gemacht wird. Dies kann z.B. per Email passieren. Wenn die Email bei der dritten Person angekommen ist, bekommt der Spieler eine Nachricht, dass er einen Zug machen kann. Nach dem Zug des Computers schickt die dritte Person die Email mit dem vorher gewählten Zug weiter an den Spieler, der dann vergleichen kann, ob der Computer geschummelt hat.

Dieses Verfahren ist sehr zeitaufwändig. Hier muss zusätzlich zu den Ein- und Ausgaben des Programms noch mit einer dritten Person kommuniziert werden. Diese dritte Person kann außerdem mit dem Programm oder mit dem Spieler unter einer Decke stecken und einerseits die Mails abändern, ganz andere Mails an den Spieler weiterleiten oder auch die Mails zu früh an den Spieler weiterschicken. Solange diese Person nicht vertrauenswürdig genug ist, ist dieses Verfahren nicht besonders geeignet; insbesondere kostet die Kontrolle des Programms unnötig zusätzliche Kommunikation.

## 1.3 Versteckende Verfahren

Um diese zusätzliche Kommunikation zu vermeiden, kann versucht werden, die Festlegung des Programms zu verstecken. Hierbei werden die Daten in einer Art direkt übertragen, in der der Spieler nicht auf die Schnelle erkennen kann, was als Nächstes kommt. Gleichzeitig muss der Spieler seine Eingabe so schnell machen, dass er in der Zwischenzeit nicht genügend Zeit hat, die Daten herauszubekommen. Damit kann der Spieler nach seiner Eingabe prüfen, ob die Ausgabe des Programms der vorherigen Festlegung entspricht, und es wird dem Spieler schwer gemacht zu schummeln.

### Große Textdatei

Eine Variante hiervon kann sein, dass vor Spielbeginn eine sehr große Textdatei erzeugt wird, in der pro Zeile jeweils zufällig Stein, Schere oder Papier steht. Das Programm kann nun drei Sekunden herunterzählen, an deren Ende der Spieler seine Wahl dem Programm übergeben muss und das Programm direkt seine eigene Wahl ausgibt. Kurz bevor die Zeit auf Null kommt, gibt das Programm eine Zahl aus. In der entsprechenden Zeile der Textdatei steht dann die Wahl des Programms. Wenn der Spieler zu lange für seine Eingabe braucht, ist die Eingabe des Spielers nicht mehr gültig. Nach dem Zug kann relativ einfach nachgeschaut werden, ob die Ausgabe des Programms korrekt war. Bei diesem Verfahren ist gesichert, dass das Programm nicht schummelt. Dies lässt sich leider für den Spieler nicht garantieren. Je kleiner die Zeiten aber sind, in denen der Spieler seine Eingabe machen muss, desto sicherer kann man sich sein, dass auch der Spieler nicht geschummelt hat.

### QR-Codes

Die Daten können auch als Barcode kodiert werden. Ein Barcode ist im Normalfall nicht sofort durch einen Betrachter lesbar. Es könnte z. B. ein 2-dimensionaler Barcode wie der QR-Code[3] angezeigt werden. So kann das Programm wie bei der großen Textdatei direkt vor der Eingabe des Spielers die eigene Festlegung als QR-Code anzeigen. Dabei kann in den QR-Code auch noch die aktuelle Zeit mit kodiert werden, damit nicht bei gleichen Festlegungen die gleichen Bilder auftauchen. Die Kombination *2011-11-14-21:56:23-Stein* ist z.B. in dem QR-Code aus Abbildung 5 codiert. So ein generierter QR-Code kann durch das Programm zusammen mit dem genutzten Zeitpunkt auf dem Bildschirm angezeigt werden; der Spieler hat dann 1-2 Sekunden Zeit, seine eigene Eingabe zu machen, damit der Code in der Zwischenzeit nicht auf die Schnelle ausgelesen wird. Nach dem Zug kann der Spieler z.B. mit der eigenen Handykamera den Code fotografieren und scannen, um die angezeigten Daten zu bestätigen.

Abbildung 5: QR-Code für *2011-11-14-21:56:23-Stein*

Bei beiden versteckenden Verfahren kann auf eine dritte Person verzichtet werden. Dafür ist es wichtig, dass dem Spieler nur minimal Zeit für seine Eingabe eingeräumt wird, damit der Spieler selber nicht schummeln kann. Da im wirklichen Leben bei dem Spiel versucht wird, gleichzeitig seine Wahl bekannt zu geben, sollte dies in den meisten Fällen keine besondere

[3] `http://de.wikipedia.org/wiki/QR-Code`

Zumutung an den Spieler sein. Dafür kann dann hinterher mit relativ einfachen Verfahren die Ausgabe des Programms verifiziert werden.

## 1.4 Kryptographische Verfahren

Mit Hilfe von kryptographischen Verfahren kann noch besser sichergestellt werden, dass weder das Programm noch der Benutzer schummeln können. In einer Variante kann die Information über den gewählten Zug durch eine kryptographische Hash-Funktion abgesichert werden. In einer anderen Variante kann ein asymmetrisches Kryptosystem mit Verschlüsselung und Signatur genutzt werden. Bei einer solchen Lösung benötigt der Spieler eine Implementierung des jeweiligen Verschlüsselungsverfahrens. Damit dies praktikabel ist, muss das Verfahren den leichten Zugang zu einer solchen Implementierung gewährleisten.

### Hashing

Das Programm kann den Zug wie bei dem Verfahren mit den QR-Codes um einen Zeitstempel und ein paar zufällige Zeichen erweitern und aus der dadurch entstehenden Zeichenkette mit Hilfe einer kryptographischen Hash-Funktion[4] einen Hashwert berechnen. Dieser Hash-Wert wird dem Spieler vor seiner Eingabe angezeigt. Wenn nun der Spieler seinen Zug gemacht hat, gibt das Programm seinen Zug und den Eingabetext für die Hash-Funktion aus. Der Spieler kann dann selber prüfen, ob die Hash-Funktion den gleichen Hash ergibt.

Z. B. können sich Spieler und Programm auf die Hash-Funktion SHA256[5] einigen. Das Programm möchte *Stein* ziehen und berechnet für die Zeichenkette

„jsielshfpe Stein jskae 2011-11-11-11:11:11“

den Hash

306ad85fa307e226a00930a0432820a98dc57938a2fd91f4fc49c62288909e4f.

Es gibt also vor der Eingabe des Spielers den Hash aus und nach der Eingabe des Spielers gibt das Programm zusätzlich die Ausgangs-Zeichenkette aus. Bei diesem Verfahren kann das Programm nur schummeln, wenn die entsprechende Hash-Funktion geknackt wurde, es also zu zwei unterschiedlichen Zügen mit Hilfe der zufälligen Zeichen Zeichenketten erzeugen kann, die den gleichen Hash ergeben. Bei dem Algorithmus SHA-256 sind aber bisher keine solchen Kollisionen gefunden worden. Deshalb sollte dieses Verfahren momentan sicher genug sein.

Damit der Spieler nicht schummeln kann, muss das Programm ein paar zufällige Zeichen mit in den Hash aufnehmen. Sonst könnte der Spieler einfach ein paar Zeitstempel ausprobieren und schauen, ob der entsprechende Hash-Wert herauskommt.

[4] `http://de.wikipedia.org/wiki/Kryptographische_Hashfunktion`
[5] `http://de.wikipedia.org/wiki/Secure_Hash_Algorithm`

### Asymmetrische Kryptographie

Genauso kann das Programm mit einem asymmetrischen Kryptosystem[6] jeden Zug (wie beim Hashing mit einem Zeitstempel und ein paar zufälligen Zeichen) mit neuen Schlüsseln verschlüsseln und signieren. Bevor der Nutzer seine Eingabe macht, gibt das Programm den verschlüsselten und signierten Zug zusammen mit dem öffentlichen Schlüssel (zur Signaturprüfung) an den Spieler. Nach der Eingabe gibt das Programm den privaten Schlüssel an den Spieler. Dieser kann dann den Zug entschlüsseln und nach der Prüfung der Signatur die Wahl des Programms kontrollieren. Wenn bei jedem Zug neue Schlüssel durch das Programm generiert und genutzt werden, kann der Benutzer auch nicht vor seiner Eingabe die übertragenen Daten entschlüsseln.

So kann man das Kryptographiesystem von *Gnu Privacy Guard* (GnuPG) nutzen. Die Kontrolle durch den Spieler kann hier mit einer graphischen Benutzeroberfläche durchgeführt werden. Bei jedem Zug müssen nun aber nicht nur die verschlüsselten Daten sondern auch die Schlüssel zur Verifizierung ausgelesen werden. Dies macht die Verifizierung deutlich komplizierter als bei den versteckenden Verfahren. Dadurch schneidet es bei der Praktikabilität schlechter ab als diese.

## 1.5 Auswahl eines Verfahrens

Am einfachsten zu bedienen ist vermutlich einmal das Verfahren mit den QR-Codes und andererseits das Verfahren mit der Hash-Funktion. Bei beiden ist es relativ einfach, die Daten des Zuges zu verschleiern, aber hinterher auch zu verifizieren. Bei der Variante mit den QR-Codes müsste das Programm aber auf eine zeitnahe Eingabe des Spielers achten, damit in der Zwischenzeit der QR-Code nicht dekodiert wurde. Diese zeitnahe Eingabe ist bei der Variante mit der Hash-Funktion nicht notwendig. Aber auch ein Verfahren wie das aus Abschnitt 1.3 ist als Lösung dieser Aufgabe akzeptabel.

## 1.6 Bewertungskriterien

Es ist nicht gefordert, ein gewähltes Verfahren zu implementieren. Punktabzug gibt es deshalb insbesondere bei offensichtlich falschen bzw. unzureichenden Aussagen zum eigenen Verfahren.

- Das Verfahren zum Nachweis der Programmfairness muss klar und nachvollziehbar beschrieben sein. Eine algorithmische Notation ist vorteilhaft.
- Es muss ausreichend und schlüssig begründet sein, warum das vorgeschlagene Verfahren die in der Aufgabenstellung genannten Anforderungen (Nachweis der Fairness des Programms, Praktikabilität).

[6] `http://de.wikipedia.org/wiki/Asymmetrisches_Kryptosystem`

- In der Lösung werden die Anforderungen bzgl. der Fairness von beiden Seiten (Programm und Spieler) betrachtet: Das Verfahren muss dem Benutzer zeigen, dass das Programm nicht schummelt. Außerdem soll das Verfahren so weit möglich verhindern bzw. es möglichst schwierig machen, dass der Spieler schummeln kann (mindestens wie bei den versteckenden Verfahren mit Zeitbeschränkung für die Eingabe). Das Verfahren sollte auch grundsätzlich sinnvoll sein, also nicht etwa seine eigenen Züge dem Spieler (fast) unverschleiert verraten.
- Das Verfahren sollte praktikabel sein im Sinne der Aufgabenstellung. Für die wenigsten Menschen ist es üblich, die Computer-Implementierung eines Verschlüsselungsverfahrens zu nutzen. Für eine Lösung, die genau das beim Spieler voraussetzt, sollte die Praktikabilität dieses Aspekts besonders begründet sein.
- Das Verfahren sollte an mindestens einem Ablaufbeispiel demonstriert werden.

# Aufgabe 2: Aladins Lampen

Das Spiel, das in dieser Aufgabenstellung zunächst beschrieben wird, ähnelt „Lights Out“[7]. Dass hier alle Lampen an- statt ausgeschaltet werden sollen, macht keinen Unterschied, nur sind die Lampen anders angeordnet. Wichtig ist, dass im Laufe der Teilaufgaben Lösungen für beliebige Varianten des Spiels mit beliebigen Schaltungen gefunden werden müssen. Gegenüber bekannten Lösungen für „Lights Out“ benötigen wir also einen allgemeineren Ansatz.

Die Nummer eines Tasters bzw. Schalters sei im Folgenden die der Lampe, der er am nächsten ist.

## 2.1 Teilaufgabe 1

*Gib eine Tastenfolge an, durch die bei der abgebildeten Ausgangssituation erreicht werden kann, dass alle Lampen angeschaltet sind.*

Die erste Teilaufgabe kann relativ schnell durch Ausprobieren gelöst werden: Es müssen die Schalter 1, 2, 3 und 7 gedrückt werden.

Dabei können folgende Beobachtungen gemacht werden:

- Die Reihenfolge, in der die Schalter gedrückt werden, spielt keine Rolle.
- Es ist nur von Bedeutung, ob ein Schalter gedrückt wird oder nicht. Wird ein Schalter zweimal gedrückt, hätte er genauso gut gar nicht gedrückt werden können. Und jede ungerade Anzahl an Schalterdrücken hat die gleiche Wirkung wie ein einzelner Schalterdruck.

Es muss also für jeden Schalter entschieden werden, ob er gedrückt wird oder nicht. Jede sinnvolle Kombination von Schalterdrücken kann man – bei $n$ Schaltern – als Folge von $n$ binären Zahlen (oder auch als $n$-stellige Binärzahl) darstellen. Damit wird offensichtlich, dass es $2^n$ solcher Schalterkombinationen gibt.

## 2.2 Teilaufgabe 2

*Entwickle ein Verfahren, das für eine beliebige Ausgangssituation eine Tastenfolge liefert, durch die alle Lampen angeschaltet werden.*

Eine einfache Lösung ist es, alle $2^n$ Schalterkombinationen systematisch auszuprobieren und diejenige auszugeben, die als erste alle Lampen anschaltet. Es gibt aber deutlich effizientere Lösungen.

Man kann das Spiel auch als eine Gleichung auffassen. Die einzelnen Schalter sind $n$-dimensionale Vektoren, wobei $n$ die Anzahl der Lampen ist. Kann ein Schalter den Zustand einer Lampe ändern, ist der Eintrag des Vektors an dieser Stelle 1, sonst 0. Es leuchten bereits

[7] `http://en.wikipedia.org/wiki/Lights_Out_(game)`

einige Lampen, also muss ein Vektor, der die Ausgangssituation repräsentiert, zu den Schalter-Vektoren addiert werden. Am Ende müssen alle Lampen leuchten. Das bedeutet, dass die Summe der Vektoren einem Vektor entsprechen muss, dessen Einträge überall gleich 1 sind.

Da einr zweimalige Zustandsänderung einer Lampe den Ausgangszustand nicht verändert, wird im Folgenden im Restring $\mathbb{Z}/2\mathbb{Z}$ gerechnet. Das bedeutet, dass $1+1 \equiv 0 \pmod 2$ gilt.

Es ergibt sich also folgende Gleichung:

$$x_1 \cdot \underbrace{\begin{pmatrix} 1 \\ 1 \\ 0 \\ \vdots \\ 1 \end{pmatrix}}_{\text{Schalter 1}} + x_2 \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 0 \end{pmatrix} + \cdots + x_7 \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} + \underbrace{\begin{pmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 1 \end{pmatrix}}_{\text{Anfangssituation}} \equiv \underbrace{\begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}}_{\text{gewünschte Endsituation}}$$

Diese kann mit dem Gaußschen Eliminationsverfahren gelöst werden. Dabei subtrahiert man zuerst den Ausgangssituations-Vektor von dem Endsituations-Vektor und schreibt alles in eine erweiterte Koeffizientenmatrix:

$$\left(\begin{array}{ccccccc|c} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right)$$

Nun eliminiert man in jedem Schritt eine Variable, bis man eine Dreiecksform erhält:

$$\left(\begin{array}{ccccccc|c} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right)$$

Aus dieser Gleichung kann Schritt für Schritt die Lösung gefolgert werden: $x_7 = 1$, $x_6 = 0$, $x_5 = 0$, $x_4 = 0$, $x_3 = 1$, $x_2 = 1$, $x_1 = 1$

Dieser Lösungsweg ist auf eine beliebige Ausgangssituation anwendbar. Damit ist Teilaufgabe 2 gelöst.

### Teilaufgabe 2 ohne Vektorrechnung

Den obigen Ansatz kann man auch ohne Kenntnisse von Vektorrechnung bzw. Linearer Algebra erarbeiten. Die oben aufgestellte Gleichung mit Vektoren kann auch als Gleichungssystem

geschrieben werden, in dem die Unbekannten $x_i$ in jeder Zeile auftreten. Lösungsverfahren für Gleichungssysteme werden in der Schule relativ früh vermittelt. Das Besondere ist, wie oben auch, dass modulo 2 gerechnet werden muss.

Diese Besonderheit kann man sich ersparen, wenn man die binären Schalter-Vektoren als Arrays logischer Werte auffasst und mit der Operation XOR arbeitet. Schreibt man die Boole'schen Werte *true* und *false* als 0 bzw. 1, entspricht XOR genau der Addition modulo 2. Verknüpft man zwei oder mehrere Schalter-Vektoren mit XOR, drückt das genau den Effekt aus, den man durch Drücken aller entsprechenden Schalter erzielt. Nun kann man für alle $2^n$ Schalterkombinationen deren Gesamteffekt berechnen. Interessant sind nun Kombinationen, die als Gesamteffekt einen Vektor haben, der nur genau eine 1 enthält: mit solchen Kombinationen lassen sich einzelne Lampen gezielt schalten, also auch aus jeder Ausgangssituation die noch nicht leuchtenden Lampen einschalten.

Für jede Lampe ist nun eine solche Schalterkombination zu ermitteln und in einer Tabelle zu speichern. Das ist aufwändig, muss aber für eine gegebene Schaltung nur einmal erfolgen. Die Schalterkombinationen kann man wiederum als logische bzw. binäre Arrays speichern: *true*/1 bedeutet „Schalter wird gedrückt“. Für eine neue Ausgangssituation müssen dann die zu den noch nicht leuchtenden Lampen gehörenden Schalterkombinationen miteinander verrechnet werden. Das geschieht wieder mit XOR, wodurch genau die Schalter mit ungeraden Druckanzahlen im resultierenden Array wieder mit *true*/1 markiert sind. Insgesamt müssen also diese Schalter gedrückt werden, damit alle Lampen leuchten.

## 2.3 Teilaufgabe 3

*Prüfe, ob die oben skizzierte Schaltung brauchbar ist.*

In dieser Teilaufgabe soll für die zweite gegebene Schaltung überprüft werden, ob jede mögliche Ausgangssituation lösbar ist. Falls das der Fall ist, gilt die Schaltung als brauchbar, sonst nicht. Da es 9 Lampen gibt, sind $2^9 = 512$ Ausgangssituationen möglich. Zwar ist dies eine Größenordnung, bei der noch alle Ausgangssituationen überprüft werden könnten, allerdings ist das nicht nötig. Wenn es möglich ist, bei einer komplett dunklen Ausgangssituation durch eine bestimmte Schalterkombination jede beliebige Lampe – und nur diese – zum Leuchten zu bringen, dann ist es auch möglich, eine beliebige Kombination zum Leuchten zu bringen. Daraus folgt, dass nur 9 Endsituationen überprüft werden müssen.

Durch Anwendung eines der bereits beschriebenen Verfahren können folgende Schalterkombinationen gefunden werden, die nur den Zustand der jeweils genannten Lampe verändern:

- Lampe 1: 2, 5, 6, 7, 8
- Lampe 2: 2, 4, 6, 7, 9
- Lampe 3: 2, 4, 5, 8, 9
- Lampe 4: 2, 3, 4, 8, 9
- Lampe 5: 1, 2, 3, 5
- Lampe 6: 1, 2, 6, 7, 8
- Lampe 7: 3, 5, 6, 7, 9
- Lampe 8: 1, 3, 4, 6, 8
- Lampe 9: 1, 4, 5, 7, 9

Will man beispielsweise nur Lampe 1 und 2 zum Leuchten bringen, müssen die Schalter 4, 5, 8 und 9 gedrückt werden.

Da jede Lampe einzeln angesprochen werden kann, ist die Antwort auf diese Aufgabe: Ja, die Schaltung ist brauchbar.

Ein alternativer Lösungsweg ist zu zeigen, dass die $n$ Schalter-Vektoren in dem $n$-dimensionalen Vektorraum, der alle Kombinationen von leuchtenden Lampen repräsentiert, linear unabhängig sind. Dies kann beispielsweise bewiesen werden, indem gezeigt wird, dass die Determinante der Vektorenmatrix ungleich 0 ist.

## 2.4 Teilaufgabe 4

*Schreibe ein Programm, das solange zufällige Schaltungen generiert und überprüft, bis eine brauchbare gefunden wurde.*

Bevor die zufällige Schaltung generiert wird, sollte man sich Gedanken machen, welche Bedingungen diese erfüllen muss:

- Wie viele Lampen sollte es mindestens / höchstens geben?
- Wie viele Lampen sollte ein Schalter mindestens / höchstens betätigen?

Bei den Maximalgrenzen ist darauf zu achten, dass eine Schaltung dieser Größe auch in annehmbarer Laufzeit überprüft werden kann. Die Anzahl der Schalter soll gleich der Anzahl der Lampen sein.

Das Verfahren zur Überprüfung einer Schaltung auf ihre Brauchbarkeit wurde bereits in Teilaufgabe 3 beschrieben. Es können also $n$ zufällige Schalter (also Schalter mit zufällig gewählten Schaltverbindungen) erzeugt werden, wobei $n$ die Anzahl der Lampen ist, und mit dem beschriebenem Verfahren überprüft werden. Dabei muss man darauf achten, dass jede Lampe von mindestens einem Schalter betätigt werden kann und die Schalter linear unabhängig sind. Dies kann beim Hinzufügen eines neuen Schalters geschehen.

Allerdings kann das auch deutlich effizienter gelöst werden. In der Hauptdiagonale der erweiterten Koeffizientenmatrix in Dreiecksform (vgl. Teilaufgabe 2) muss jeder Eintrag gleich Eins sein, wenn die Vektoren unabhängig sein sollen. Damit ist gewährleistet, dass die erzeugte Schaltung brauchbar ist. Wie die Einträge oberhalb lauten ist egal, sie können also zufällig auf Eins oder Null gesetzt werden. Da es jedoch nicht erwünscht ist, dass ein Schalter nur eine Lampe betätigen kann und ein anderer eventuell alle Lampen, muss noch eine Möglichkeit gefunden werden, wie der untere Teil der Matrix bearbeitet werden kann.

Die Eins, die man in eine Zeile $i$ und Spalte $j$ unter der Hauptdiagonalen schreibt, kann entfernt werden, indem man Zeile $i$ − Zeile $j$ rechnet. Die Matrix muss jedoch immer noch nur Einsen in der Diagonalen haben. Also muss der Null-Eintrag in Zeile $j$, Spalte $i$ blockiert werden. Mit der selben Logik darf der Eintrag in Zeile $j$, Spalte $i$ nicht mehr bearbeitet werden, wenn in Zeile $i$, Spalte $j$ oberhalb der Hauptdiagonalen gleich 1 gesetzt wurde.

Der Algorithmus könnte also folgendermaßen aussehen:

```
LAMPEN_MIN = 4
LAMPEN_MAX = 15
LAMPEN_PRO_SCHALTER_MIN = 2
LAMPEN_PRO_SCHALTER_MAX = 7

anzahlLampen = randomInteger(LAMPEN_MIN, LAMPEN_MAX)

// Initialisiere Matrix mit 0en und fülle die Hauptdiagonale mit 1en.
matrix = erstelle Dreiecksmatrix()

// Hier bedeutet 1, dass kein Wert an diese Stelle eingefügt werden darf.
blocked = copy(matrix)

for (schalter = 0; schalter < anzahlLampen; schalter++) {
  // liste := alle Lampen, die der aktuelle Schalter verändern darf.
  liste = getUnblocked(blocked, schalter)

  // 1 wegen der Hauptdiagonale
  if (len(liste) + 1 < LAMPEN_PRO_SCHALTER_MIN) {
    restartAlgorithm()
  }

  // Anzahl Lampen, die dieser Schalter zusätzlich noch verändern soll.
  tmpMax = min(len(liste), LAMPEN_MAX)
  lampen = randomInteger(LAMPEN_PRO_SCHALTER_MIN, tmpMax) - 1

  while (lampen > 0) {
    neueLampe = random(liste)
    matrix[schalter][neueLampe] = 1
    blocked[schalter][neueLampe] = 1
    blocked[neueLampe][schalter] = 1
    liste = getUnblocked(blocked, schalter)
    lampen--
  }
}

ausgabe(matrix)
```

## 2.5 Bewertungskriterien

- Die Lösungen zu den Teilaufgaben 1 und 3 müssen korrekt sein. Insbesondere muss in Teilaufgabe 1 eine korrekte Tastenfolge bzw. Schalterkombination angegeben worden sein, und in Teilaufgabe 3 muss erkannt worden sein, dass die Lösung brauchbar ist.
- Wichtige Eigenschaften des Spiels, wie die Tatsache, dass die Reihenfolge der Tastendrücke keine Rolle spielt oder dass bei der Häufigkeit der Tastendrücke nur zwischen gerader (einschließlich 0) und ungerader Anzahl unterschieden werden muss, sollen erkannt worden sein.
- Das Verfahren für Teilaufgabe 2 (Lösung für sieben Lampen) muss korrekt, verlässlich und systematisch funktionieren. Es genügt also nicht, durch zufälliges Betätigen der

Schalter eine Lösung zu finden.

- Das Verfahren für Teilaufgabe 4 (Generierung brauchbarer Schaltungen) muss korrekt und systematisch sein. Es ist akzeptabel, wenn in Abweichung von der Aufgabenstellung ein Verfahren angegeben wird, das gezielt brauchbare Schaltungen generiert.
- Fehlende Lösungen für die Teile 2 und 4 werden wie fehlerhafte Lösungen bewertet.
- Die Lösungsansätze zu den Teilaufgaben 2 und 4 bzw. ihre Realisierung sollten die jeweiligen Probleme einigermaßen effizient lösen. Insbesondere führt bei Teilaufgabe 2 die zuerst genannte Lösung durch (systematisches) Ausprobieren zu einem Abzug.
- Für die Teilaufgaben 2 und 4 sollten Beispiele angegeben sein: für Teil 2 eine weitere Ausgangssituation mit sieben Lampen, für Teil 4 eine per Programm generierte brauchbare Schaltung.

# Aufgabe 3: Fehlerfrei puzzeln

## 3.1 Lösungsidee

Mit Problemen wie diesem beschäftigt man sich in der kombinatorischen Optimierung. Hierbei geht es darum, aus einer großen Menge von Elementen (hier: alle möglichen Anordnungen der Puzzlesteine) eine Teilmenge auszuwählen, die bezüglich einer gegebenen Kostenfunktion (hier: geringe Anzahl der Strafpunkte) optimal ist. Manchmal müssen noch zusätzliche Nebenbedingungen erfüllt sein.

Es gibt verschiedene Ansätze, solch ein Problem zu lösen:

**Ausprobieren aller Lösungen** Wenn wir, etwa mit Backtracking, alle möglichen Steinpositionen durchprobieren wollten, müssten wir extremen Rechenaufwand betreiben: Wenn wir davon ausgehen, dass alle Steine verschiedene Muster besitzen (es gibt $4^4 = 256$ mögliche Muster, aber nur 96 Steine), dann ist die Anzahl der möglichen Positionierungen dieser Steine $96! = 96 \cdot 95 \cdot \ldots \cdot 2 \cdot 1$ – eine Zahl mit 150 Stellen!

**Beschränkung des Suchraums** Bestimmte Verfahren der kombinatorischen Optimierung reduzieren die Zahl der Lösungskandidaten, indem bestimmte Lösungen von vornherein ausgeschlossen werden. Diese Verfahren erzeugen aber nur garantiert optimale Lösungen – an denen sind wir hier aber nicht interessiert; wir wissen ja nicht, ob es eine Belegung ohne Strafpunkte gibt.

**Heuristiken** Heuristiken nutzen spezielles Wissen über die Problemstruktur, um mit begrenzten Ressourcen zu guten Ergebnissen zu kommen. In unserem Fall ist das beispielsweise die Tatsache, dass man zwei Steine vertauschen und dadurch die Lösung verbessern oder verschlechtern kann. Diese Algorithmen gehen einen Kompromiss zwischen Rechenzeit und Lösungsgüte ein – garantieren also keine optimale Lösung und auch keine kurze Rechenzeit. Mit den richtigen Parametern kann man aber trotzdem ansehnliche Ergebnisse erhalten.

Bei diesem Problem wollen wir eine als **Simulated Annealing**[8] bekannte Heuristik anwenden. Beim Annealing handelt es sich um ein Verfahren aus der Werkstoffkunde: Dabei wird ein Metall zuerst erhitzt und dann langsam abgekühlt. Während des Abkühlungsprozesses haben die Atome genug Zeit, sich in eine energetisch optimale Position zu begeben, wodurch die Struktur des gesamten Materials optimiert wird.

Übertragen auf unser Problem bedeutet das:

- In der Erhitzungsphase bewegen sich die Moleküle unkontrolliert durcheinander. Analog müssen wir beliebige Änderungen an der Lösungsstruktur zulassen.

[8] vgl. P. Rossmanith: Simulated Annealing. In B. Vöcking et al.: Taschenbuch der Algorithmen. Springer-Verlag, 2008.

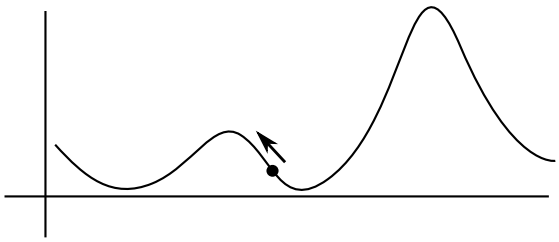

Abbildung 6: Methode des steilsten Anstiegs

- Je niedriger die Temperatur beim Abkühlen, desto geringer ist die Wahrscheinlichkeit, dass eine Atombindung gelöst wird und die Materialstruktur sich verändert. Somit lassen wir also mit sinkender Temperatur immer weniger Änderungen zu, die die Struktur beeinträchtigen.

Es bleibt noch zu klären, wie die „zufälligen“ Strukturänderungen umgesetzt werden sollen. Hier bietet es sich an, die „kleinstmögliche“ Änderung zu betrachten, nämlich das Vertauschen zweier Steine. Mit dieser Operation können wir nämlich jede beliebige Anordnung in jede andere überführen.

Ein Vertauschen zweier Steine kann prinzipiell zu einer Verbesserung oder einer Verschlechterung der Lösung führen. Da stellt sich natürlich die Frage, warum wir nicht einfach zufällig Steine vertauschen, wobei wir *nur* Verbesserungen zulassen. Dieses Verfahren gibt es auch, es heißt *Methode des steilsten Anstiegs.*

In Abbildung 6 ist diese Methode für das Problem gezeigt, das Maximum einer Funktion zu finden. Mit der Methode des steilsten Anstiegs würden wir vom Startpunkt aus so lange nach oben gehen, bis es nicht mehr höher geht. Hier ist es sofort klar, dass diese Methode nicht zum globalen Maximum der Funktion führt. Stattdessen erreichen wir ein lokales Optimum und stecken dort fest.

Simulated Annealing verhindert genau dieses Feststecken, indem temporär auch Verschlechterungen der Lösung zugelassen werden. Die Wahrscheinlichkeit dafür nimmt mit der Temperatur ab. Algorithmus 3 zeigt die Struktur des Simulated-Annealing-Algorithmus.

Nun kommt es darauf an, die Funktionen $\mathbb{P}$ und $T$ sowie die zufällige Veränderung der Lösung auszuwählen. Für Letzteres ist, wie vorhin erwähnt, das zufällige Vertauschen zweier Steine eine gute Wahl.

Wenn die Lösung durch die zufällige Veränderung verbessert wird oder gleich gut bleibt, wollen wir diese Veränderung auf jeden Fall annehmen. Somit definieren wir $\mathbb{P}(L', L_n, t) := 1$ für $SP(L') \leq SP(L_n)$, wobei $SP(L)$ die Anzahl Strafpunkte der Lösung $L$ bezeichnet.

Die Wahrscheinlichkeit dafür, dass eine Verschlechterung der Lösungsgüte angenommen wird, soll mit der Temperatur sinken. Ebenso sollen kleine Verschlechterungen eher angenommen werden als große. Dies führt (für $SP(L') > SP(L_n)$) zur Wahl von:

$$\mathbb{P}(L', L_n, t) := e^{-(SP(L')-SP(L_n)/t)}$$

**Algorithmus 3** Simulated Annealing

**Eingabe:** Anfangslösung $L_0$, max. Schrittanzahl $N$
**Ausgabe:** Möglichst gute Lösung

```
n ← 0
while Lösung ist nicht optimal und n < N do
    Berechne Lösung L' aus L_n durch zufällige Veränderung
    if P(L', L_n, T(n)) ≥ rand(0,1) then      ▷ mit einer gewissen Wahrscheinlichkeit ...
        L_{n+1} ← L'                           ▷ ... behalte Veränderung bei
    else
        L_{n+1} ← L_n                          ▷ ... sonst mache Änderung rückgängig
    end if
end while
return L_N
```

Bleibt noch die Wahl der Temperaturfunktion. Die Temperatur soll zu Anfang so hoch sein, dass beliebige Veränderungen zugelassen werden, und danach immer weiter abnehmen. Eine Möglichkeit ist es, die Temperatur zu Anfang auf $T_0 := 8$ zu setzen (dies ist die größtmögliche Verschlechterung beim Vertauschen zweier Steine) und alle $k$ Schritte mit einer Konstanten $o < 1$ zu multiplizieren. Die Temperaturformel lautet dann:

$$T(n) = T_0 \cdot o^{n/k}$$

Die optimalen Werte für $o$ und $k$ muss man durch Probieren herausfinden; möglich sind z.B. $o = 0,96$ und $k = 5000$.

## 3.2 Implementierung

Dieser Algorithmus muss natürlich implementiert werden, damit das Verfahren auf beliebige Eingabedateien angewandt werden kann. Eine grafische Ausgabe des Ergebnisses oder der Zwischenergebnisse ist sinnvoll, muss aber nicht sein. Ebenso könnte man ein Schaubild der Temperaturfunktion und der Lösungsgüte zeichnen.

Ein kleines Problem bereitet das nicht dokumentierte Eingabeformat. So etwas kann einem im Alltag als Programmierer leider auch begegnen – man muss dann durch etwas Nachdenken auf die korrekte Lösung kommen. Offensichtlich beschreibt jede Zeile einen Stein, denn es sind 96 Zeilen. Die ersten beiden Zahlen geben die Koordinaten an, denn sie sind aus dem Bereich von 1 bis 8 beziehungsweise 1 bis 12. Interessant wird es bei den Farben – es gibt keinen Hinweis darauf, in welcher Reihenfolge die Farben angegeben sind. Man sucht also nach Steinen, die nur ein einziges Mal vorkommen, etwa den Stein mit drei grünen und einer orangenen Seite. In der Anfangsanordnung steht er an Position $(2|3)$ mit der Bezeichnung `Green Green Green Orange` – die letzte Farbe muss die auf der linken Seite sein. Mit weiteren solchen Überlegungen findet man heraus, dass die Steinfarben in der Reihenfolge `Oben Rechts Unten Links` angegeben sind.

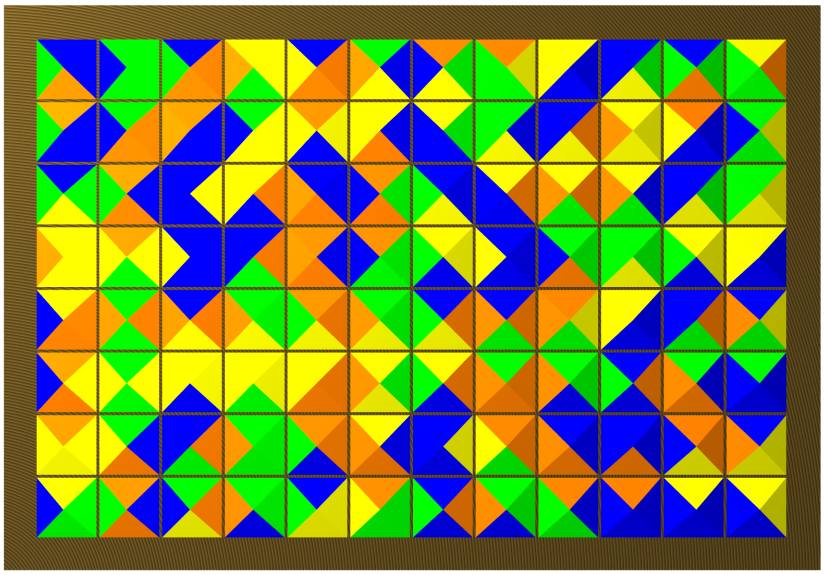

Abbildung 7: Eine fehlerfreie, also optimale Lösung des Puzzlespiels.

## 3.3 Ergebnisse

Mit Simulated Annealing kann die im Aufgabenblatt angegebene Lösung gefunden werden – aber auch die Lösung ohne Strafpunkte, die in Abbildung 7 gezeigt ist. Mit den Augen lässt sich die Korrektheit einer Lösung übrigens einigermaßen gut durch „schräge Blicke“ entlang der Diagonalen im Puzzle-Bild ermitteln.

## 3.4 Weitere Lösungsideen

Auch ohne Kenntnis von Verfahren wie Simulated Annealing lässt sich diese Aufgabe bearbeiten und eine fehlerfreie Lösung für das gegebene Puzzle finden. Erfolgreich können insbesondere Verfahren sein, die zufällige Operationen nutzen – wie beim Simulated Annealing. Zum Beispiel können nicht nur einzelne Steine zufällig für eine Vertauschung ausgewählt werden, sondern auch ganze fehlerfreie Bereiche des Spielfelds. Auch eine solche Umplatzierung einer Teillösung kann den Weg aus einem lokalen Optimum eröffnen.

Auch Verfahren, die prinzipiell ein Backtracking darstellen, können erfolgreich sein, wenn Ansätze zur Einschränkung des Suchraums geeignet kombiniert werden: Perfekte Teillösungen für kleine Bereiche des Feldes können durch vollständige Suche ermittelt werden und als Ausgangspunkt für heuristisch gesteuertes Puzzeln dienen. Bei der Belegung eines Feldes kann man einen Stein so wählen, dass zur Belegung der Nachbarfelder möglichst viele andere Steine zur Verfügung stehen. Außerdem kann man berechnen, welche Steine bevorzugt am Rand liegen sollten, weil die Anzahl der Partnersteine abweicht. Wenn z.B. 17 Steine an der

unteren Seite ein gelbes Dreieck haben, aber 20 Steine an der oberen Seite ein gelbes Dreieck, müssen mindestens drei Steine mit gelbem Dreieck am oberen Rand liegen.

## 3.5 Bewertungskriterien

- Das Lösungsverfahren sollte gut nachvollziehbar beschrieben sein.
- Ein mehr oder weniger naiver Ansatz kann diese Aufgabe nicht lösen und ist deshalb nicht akzeptabel. Das Lösungsverfahren bzw. seine Implementierung sollten korrekte Ergebnisse liefern und nicht allzu sehr auf das Eingabebeispiel ausgerichtet sein.
- Für das gewählte Verfahren sollen auch mögliche Nachteile angesprochen werden. Insbesondere sollte bei der Anwendung von Heuristiken nicht behauptet werden, dass die Lösung (garantiert) optimale Ergebnisse liefert.
- Wir wollen nicht verlangen, dass die optimale Lösung gefunden wird. Mehr als ein oder zwei Strafpunkte sollten die gefundenen Lösungen aber nicht haben.
- Zumindest die beste gefundene Lösung sollte abgebildet oder auf andere Weise anschaulich und überprüfbar dargestellt sein. Auf evtl. noch vorhandene Fehler sollte in Text und oder Bild hingewiesen werden.

# Aufgabe 4: Zaras dritter Fehler

Die Aufgabe ist algorithmisch nicht sehr schwer. Die nötigen Manipulationen an Zeichenketten werden von den gängigen Programmiersprachen gut unterstützt. Für die Suche nach deutschen Wörtern können Wörterbücher aus dem Internet verwendet werden. Dazu kommt, dass die Aufgabe nicht verlangt, die Texte fehlerfrei und eindeutig vom Programm identifizieren zu lassen – es ist lediglich verlangt, die Lösungen zu ermitteln. Dafür genügt es auch, eine halbwegs kleine Zahl möglicher Kandidaten zu generieren, unter denen man dann den richtigen wählen kann.

Die Bearbeitung der Aufgabe kann gut in zwei Teile untergliedert werden. Im ersten Teil wird der Text verarbeitet, und für jede Zeile des Textes wird für alle Zahlenfolgen je ein Lösungskandidat generiert. Im zweiten Teil müssen dann alle Kandidaten geprüft werden, um diejenigen zu ermitteln, die tatsächlich deutsche Sätze sind.

## 4.1 Erster Teil: Vorverarbeitung und Kandidatensuche

Als erstes muss natürlich der Text eingelesen werden – am besten zeilenweise, denn die Zeilenanfänge sind die Startpunkte für die Lösungskandidaten. Dabei muss man die UTF-8 Kodierung berücksichtigen. Beim Einlesen (oder unmittelbar danach) kann man den Text verarbeiten, wobei zu den Verarbeitungsschritten folgendes zählt:

**Sonderzeichen entfernen:** Die gängigen Sonderzeichen zum entfernen sind neben dem Leerzeichen: `-.,:;?!"{}[]()'#*`. Dazu kommen die Ziffern 0–9 sowie die etwas ungewöhnlichen Anführungszeichen: » und «

**In Kleinbuchstaben konvertieren:** Für den späteren Abgleich mit einem Wörterbuch ist es vorteilhaft, den Text unmittelbar in Kleinbuchstaben umzuwandeln.

**Leerzeilen entfernen:** Um keine doppelten Lösungen zu erhalten, sollte man Leerzeilen aus dem Text streichen.

Für die ersten beiden Schritte gibt es Unterstützung in den meisten Programmiersprachen (in java etwa `String.replaceAll` und `String.toLowerCase`). Wenn man diese Funktionen nicht kennt, kann man den Text immer noch zeichenweise durchgehen und entsprechende Zeichen ersetzen oder verwerfen.

Hat man den Text eingelesen, dann kann man Kandidaten generieren: Dabei wird die zu bearbeitende Zahlenfolge zuerst eingelesen und dann auf den Text angewendet, um den geheimen Satz, der in dieser Zeile beginnen würde, zu extrahieren. Da ein geheimer Satz aus Buchstaben mehrerer Zeilen zusammengesetzt ist, muss man dabei die Zeilengrenzen überspringen können. Dafür gibt es mehrere Möglichkeiten.

1. Man kann den gesamten Text zu einem langen String konkatenieren (dabei muss man sich die Stellen merken, an denen eine Zeile beginnt). Dann muss man an den jeweiligen Zeilenanfängen nur beginnen und jeweils die Buchstaben abzählen.

2. Man kann die Zeilen des Textes in einem Array belassen und muss dann jeweils am Zeilenende aufpassen, dass man in der nächsten Zeile mit der richtigen Differenz weiter zählt.
3. Als dritte Möglichkeit kann man die Zeilen einzeln belassen, aber jede Zeile um die folgenden (z.B. zehn) Zeilen erweitern. Bei dieser (etwas verschwenderischen) Lösung kann man sich die Mühe am Zeilenende sparen und stattdessen einfach abzählen.

Beim Abzählen kann es von Vorteil sein, wenn man beim Einlesen die Zahlenfolge in die Folge der *Präfixsummen* umwandelt: z.B. wird dann aus $13, 43, 7\ldots$ die Folge $13, 56, 63\ldots$ Dann kann man vorhandene Funktionen (wie zum Beispiel `String.charAt`) nutzen, um die Buchstaben an den einzelnen Positionen leicht zu ermitteln.

## 4.2 Zweiter Teil: Wörterbuchabgleich

Um den richtigen geheimen Satz unter allen Kandidaten zu finden, kann man versuchen, jeden Kandidaten in kleinere Abschnitte zu zerlegen und diese als deutsche Wörter zu identifizieren. Im richtigen Satz sollten mehrere (bestenfalls alle) Abschnitte auch deutsche Wörter sein.

Für die Identifizierung der Wörter benötigt man ein Wörterbuch mit allen oder zumindest häufigen deutschen Wörtern. Eine kurze Onlinesuche liefert verschiedene Möglichkeiten. Sehr gut funktioniert zum Beispiel ein kostenloser Korpus mit ca. 1,2 Millionen Einträgen[9]. Zum Speichern des Korpus sollte man eine geeignete Datenstruktur verwenden (z.B. Set und nicht gerade List), da sehr oft geprüft werden muss, ob eine bestimmte Buchstabenfolge ein Wort ist.

Bei der Suche nach Wörtern gibt es verschiedene Ansätze. Größe des Textes und Anzahl der Kandidaten lassen Brute-Force-Ansätze zu. Im Folgenden werden zwei Varianten vorgestellt, die jeweils mit Beispielen, Pseudocode und der Abbildung 8 verdeutlicht werden. Alternativ ist auch denkbar, die Kandidaten nicht komplett in Wörter zu zerlegen, sondern Vorkommen von Wörtern darin zu suchen – je länger die vorkommenden Wörter sind, desto besser. Bei einem solchen Ansatz kommt man mit unvollständigen Wörterbüchern aus; es kann sogar genügen, die Menge aller Wörter aus *Effi Briest* als „Wörterbuch“ zu verwenden.

### Variante 1:

Man beginnt vorn an der Zeichenfolge und sucht nach dem kürzesten möglichen Wort. Dafür untersucht man die ersten beiden Buchstaben (`Start=1, i=2`). Handelt es sich um ein Wort aus dem Wörterbuch, trennt man das Wort ab und setzt die Suche mit dem verbleibenden String (rekursiv) fort (`Start=3, i=4`). Handelt es sich nicht um ein Wort, dann nimmt man einen weiteren Buchstaben dazu (`i=5`), dann den nächsten usw. Wenn man am Ende der Zeichenkette angekommen ist und unterwegs immer Wörter gefunden hat, dann ist man mit hoher Wahrscheinlichkeit auf die Lösung gestoßen. Findet man unterwegs an einer Stelle kein Wort, muss man in der Rekursion zurückgehen (Backtracking), die bisher gefundenen Wörter

[9] `http://sourceforge.net/projects/germandict/`

verlängern und damit erneut suchen (das liegt daran, dass es Wörter gibt, die andere Wörter als Präfix haben).

**Algorithmus 4** Prüfen eines Kandidaten, Variante 1

```
function CHECKFORCODE(String codeWord)
    keep = false
    if codeWord.length()==0 then                          ▷ wir sind fertig
        return true
    end if
    for index = 1 → codeWord.length() do
        word = codeWord.substring(0, index)
        if words.contains(word) then                      ▷ suche Wörter im Rest des Strings
            keep = keep || CHECKFORCODE(codeWord.substring(index))
        end if
    end for
    return keep
end function
```

Ein Beispiel mit dem Kandidaten aus der Aufgabe (siehe auch Abbildung 8, links sowie den Pseudocode in Algorithmus 4): `ohneliebekeinewahrheit`

**`ohneliebekeinewahrheit:`** Als erstes wird „oh“ als Wort im Wörterbuch gefunden und abgetrennt. Danach wird mit dem verbleibenden String `neliebekeinewahrheit` weiter gesucht.

**`neliebekeinewahrheit:`** Hier findet man keinen Präfix mehr, der ein Wort darstellt. Entsprechend geht man einen Schritt zurück und verlängert das zuvor gefundene Wort.

**`ohneliebekeinewahrheit:`** Als nächstes findet man „ohne“ als Wort im Wörterbuch und setzt die Suche fort mit `liebekeinewahrheit`.

**`liebekeinewahrheit:`** Zuerst findet man (fälschlicherweise) „lieb“ als Wort und trennt es ab.

**`ekeinewahrheit:`** Hier gibt es erneut keinen Präfix, der als Wort im Wörterbuch steht. Entsprechend muss einen Schritt zurück gegangen werden.

**`liebekeinewahrheit:`** Jetzt nimmt man das „e“ noch dazu und findet „liebe“ im Wörterbuch. Die Suche setzt sich mit dem Rest der Zeichenkette wie oben fort.

Da viele Wörter im Deutschen Zusammensetzungen kürzerer Wörter sind, bekommt man oft mehrere Lösungen angeboten. Für die Zahlenfolge 1 erhält man z.B. die folgenden möglichen Lösungen (und noch mehr), aus denen sich der Klartext für den menschlichen Betrachter aber unmittelbar ergibt.

- `das den ken soll man den pferden über lassen die ha ben den größe ren kopf`
- `das den ken soll man den pferden über lassen die ha ben den größeren kopf`
- `das den ken soll man den pferden über lassen die hab enden größe ren kopf`

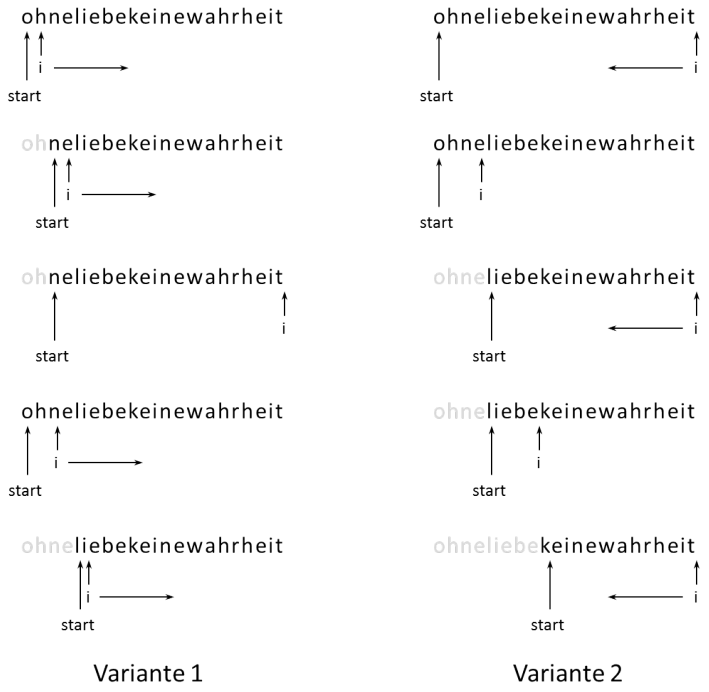


Abbildung 8: Vergleich beider Varianten

- `das den ken soll man den pferden über lassen die hab enden größeren kopf`
- `das den ken soll man den pferden über lassen die haben den größe ren kopf`
- `das den ken soll man den pferden über lassen die haben den größeren kopf`
- `...`

### Variante 2:

Die zweite Variante ist, jeweils das längstmögliche Wort zu suchen, indem man den Satz-Kandidaten vom Ende zum Anfang durchsucht (siehe Abbildung 8 rechts und Algorithmus 5). Dabei lässt man einen Index `i` vom Ende zum Anfang laufen und prüft für jede Position, ob `Satz[Start, i]` ein Wort ergibt. Findet man ein Wort, dann setzt man `Start` nach dessen Ende (`i+1`) und `i` wieder auf die letzte Stelle des Satz-Kandidaten. Findet man solange gültige Wörter bis `i==Ende des Kandidaten`, so hat man die Zerlegung mit den jeweils längstmöglichen deutschen Wörtern und höchstwahrscheinlich die richtige Lösung gefunden.

Hierbei ist Folgendes zu beachten: Wenn zwei aufeinanderfolgende Wörter ein korrektes zusammengesetztes Wort bilden (z.b. „will“ „kommen“ und „willkommen“) so kann es passieren, dass die Zerlegung mit dem längeren Wort nicht zum gewünschten Ergebnis führt. Beispiel: der Satz lautet „ich will kommentator werden“. Der oben beschriebene Algorithmus liefert aber „ich willkommen tatorwerden“ und bricht ab. Deshalb muss vor einem Abbruch immer versucht werden, das letzte gefundene Wort noch einmal zu zerlegen, indem man

**Algorithmus 5** Prüfen eines Kandidaten, Variante 2

```
function RSEARCH(String candidate, int start, int end)
                    ▷ Suche das längste Wort, das bei start beginnt und vor oder mit end endet.
    for i = end → start do
        word = candidate.substring(start, i)
        if words.contains(word) then
            return i
        end if
    end for
    return start
end function

function CHECKFORCODE(String candidate)
    foundWords = list()
    start, end = 0, candidate.length()
    while start < candidate.length() do
        end = RSEARCH(candidate, start, end)
        if end == 0 then                          ▷ Kandidat ist überprüft, kein Wort gefunden.
            return null
        else if end == start then
                ▷ Nach dem letzten Wort keine weiteren gefunden, teile das letzte Wort auf.
            start, end = foundWords.pop()
            end -= 1
        else                                      ▷ Wort gefunden, danach weitersuchen.
            foundWords.push((start, end))
            start, end = end, candidate.length()
        end if
    end while
    return foundWords
end function
```

`Start` auf den Anfang des letzten Wortes setzt und `i` direkt vor das Ende des letzten Wortes. Dann lässt man den Algorithmus wie beschrieben fortfahren. Im Beispiel wird dann „ich will“ mit dem Rest „kommentatorwerden“ gefunden, was zu der korrekten Zerlegung führt.

Diese Variante wird im Gegensatz zu der ersten immer nur eine Lösung zurückliefern, und zwar die, die aus den längstmöglichen Wörtern besteht. Das muss nicht immer die richtige Lösung sein. So liefert der Algorithmus für die Zahlenfolge 7 etwa `werden aal hält bei dem schwanz dem bleibt er weder halb noch ganz`. In diesem Fall ergeben „wer“ und „den“ auch ein zusammengesetztes, legales Wort. Die Zerlegung des restlichen Satzes verläuft danach trotzdem erfolgreich. Nur wenn die Zerlegung keinen Erfolg gehabt hätte, hätte der Algorithmus „werden“ aufgetrennt und einen neuen Versuch gestartet.

Alle anderen Zahlenfolgen werden korrekt und eindeutig gelöst.

## 4.3 Ergebnisse

| **Zahlenfolge** | **Ergebnis** |
|---|---|
| 0 | Ohne Liebe keine Wahrheit |
| 1 | Das Denken soll man den Pferden überlassen. Die haben den größeren Kopf. |
| 2 | Wenn du irgendetwas verloren hast, nimm an, du hättest es einem Armen gegeben. |
| 3 | Der Ellenbogen ist dem Munde nahe, dennoch kann man nicht selbst hinein beißen. |
| 4 | Wovon das Herz voll ist, davon geht der Mund über. |
| 5 | Je verdorbener der Staat, desto mehr Gesetze hat er. |
| 6 | Erinnere dich Mensch, dass du aus Staub bist und zu Staub zurückkehren wirst. |
| 7 | Wer den Aal hält bei dem Schwanz, dem bleibt er weder halb noch ganz. |
| 8 | Schlechte Handwerker klagen über ihr Werkzeug. |
| 9 | Wenn man durch Zweifel läuft, ist der Weg zum Himmel lang. |
| A | Jede Dummheit leidet am Ekel vor sich selbst. |
| B | Ein guter Spruch ist die Wahrheit eines ganzen Buches in einem einzigen Satz. |

Tabelle 3: Ermittelte Klartexte (Satzzeichen zur besseren Lesbarkeit ergänzt)

Die Klartexte zu den gegebenen Zahlenfolgen sind in Tabelle 3 zusammengefasst.

## 4.4 Bewertungskriterien

- Natürlich ist das vorrangige Ziel, die Klartexte zu ermitteln. Auch wenn ein Programm nicht explizit gefordert ist, ist es wegen des Textumfangs mehr als sinnvoll, dieses Ziel zumindest teilweise per Software zu erreichen. Eine rein manuelle Lösung wird nicht akzeptiert, auch nicht die Auflistung aller möglichen Strings für jede der über 9500 Textzeilen. Es ist aber in Ordnung, wenn ein Programm eine gute Vorauswahl liefert.
- Das Entschlüsselungsverfahren kann nur dann funktionieren, wenn es keine Fehler hat. Mögliche Ursachen für Probleme:
  - Nicht alle Sonderzeichen entfernt. Speziell die Ziffern 0–9 können dafür sorgen, dass eine Lösung nicht gefunden wird.
  - Probleme mit der UTF-8 Kodierung, speziell mit den deutschen Umlauten (führt unter Umständen dazu, dass nur die Lösungen ohne Sonderzeichen gefunden werden: 0, 5, A, B).
  - Beginn der Suche nicht nur am Start einer Zeile, sondern bei jedem Buchstaben.
- Zumindest an einem Beispiel sollte das Verfahren ausführlicher erläutert werden.
- Das Beispiel aus der Aufgabenstellung legt die Annahme nahe, dass Zara bei der Verschlüsselung immer die minimale Distanz zum nächsten benötigten Buchstaben wählt. Diese Annahme ist zwar im Nachhinein richtig, vereinfacht die Lösung aber erheblich, und die Verallgemeinerung eines einzelnen Beispiels ist allzu mutig.
- Alle Texte sollten entschlüsselt sein. Keinen Abzug gibt es, wenn nur der Text nicht entschlüsselt wurde, bei dem auch Ziffern ignoriert werden müssen; dies ging aus der Aufgabenstellung nicht klar hervor.
- Die Lösungstexte sollen vollständig und übersichtlich präsentiert werden.

# Aufgabe 5: Städtepartner

Der wichtigste Schritt bei der Lösung eines Informatik-Problems ist sehr häufig, die zu verarbeitenden Informationen so zu organisieren, dass sich das Problem klar und knapp formulieren lässt.

Schauen wir uns die Problemlage noch einmal informell an: Gegeben sind eine Menge von Städten sowie eine Menge von Städtepartner-Paaren. Für jede Stadt ist bekannt, wie viele Städtepartner-Feste sie höchstens ausrichten kann. Nun soll für jedes Städtepaar entschieden werden, welche Stadt das Fest für diese Partnerschaft ausrichten soll; für jedes Paar muss also eine der beteiligten Städte so ausgewählt werden, dass keine Stadt öfter ausgewählt wird als sie Feste ausrichten kann.

Mit den Beispiel-Eingabedaten wird eine mathematische Formalisierung des Problems zumindest bzgl. der Organisation der Informationen nahe gelegt:

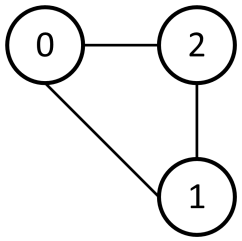


Abbildung 9: Einfaches Beispiel mit 3 Städten

Städte und Städtepartnerschaften können nämlich als ungerichteter Graph $G = (V, E)$ aufgefasst werden: Jeder Knoten $v \in V$ des Graphen entspricht dabei einer Stadt, und jede Kante $e = \{u, v\} \in E$ entspricht einer Städtepartnerschaft. Außerdem ist durch die Obergrenzen eine Funktion $c : V \to \{0, 1, 2, \ldots\}$ definiert: Für jeden Knoten $v \in V$ bedeutet $c(v)$ die vereinbarte Obergrenze für die Anzahl der Feste, die von der entsprechenden Stadt auszurichten sind. Abbildung 9 zeigt den Graphen für ein einfaches Beispiel mit drei Städten und drei Partnerschaften; für jeden Stadt-Knoten $v$ ist die Fest-Obergrenze $c(v)$ angegeben.

Ein Algorithmus zur Lösung des Problems muss nun für jede Kante einen der beteiligten Knoten dem anderen vorziehen. Diese Wahl kann erledigt werden, in dem der Kante eine Richtung gegeben wird: aus jeder ungerichteten Kante $\{u, v\}$ wird eine gerichtete Kante, und zwar entweder $(u, v)$ – die Kante mündet im Knoten $v$ – oder $(v, u)$ – die Kante mündet in $u$. Die Kanten sind so zu richten, dass im sich daraus ergebenden gerichteten Graphen kein Knoten $v$ mehr als $c(v)$ einmündende Kanten hat – sofern dies möglich ist. Eine Stadt richtet dann für jede einmündende Kante ein Fest aus.

## 5.1 Lösungsansatz 1: Flussberechnung

Ein Lösungsansatz führt das Problem auf die Berechnung eines maximalen Flusses zurück, ein klassisches und wohluntersuchtes Problem, für das viele effiziente Algorithmen bekannt sind. Das geht so:

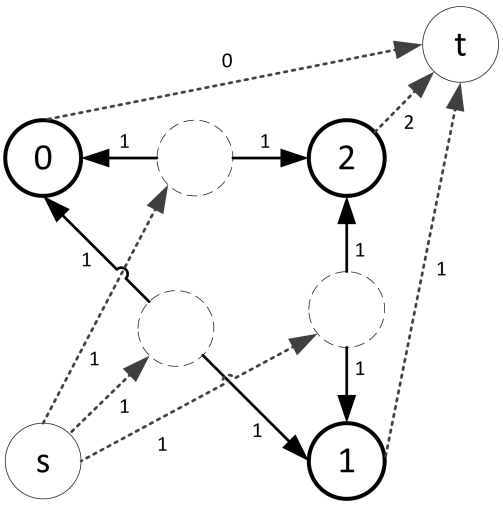


Abbildung 10: Transformierte Eingabe mit Mittelknoten (gestrichelt), Quelle s und Senke t

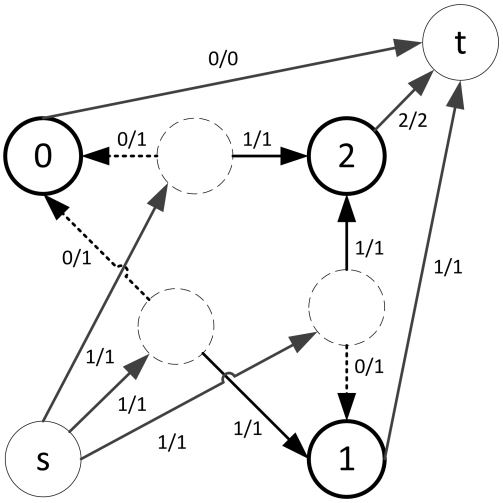


Abbildung 11: Ein maximaler Fluss im Beispiel-Netzwerk. Die Beschriftung zeigt an, wie viel „Flusseinheiten“ von der Kapazität genutzt wurden.

Erstelle aus dem ursprünglichen Graphen $G$ folgendermaßen ein gerichtetes Netzwerk: Ersetze jede ungerichtete Kante $e = \{u, v\} \in E$ durch einen neuen Knoten $w_e$ und zwei gerichtete Kanten mit Kapazität 1 von $w_e$ nach $u$ und von $w_e$ nach $v$. Die neuen Knoten der Form $w_e$ nennen wir „Mittelknoten“, da man sie sich mitten auf den ursprünglichen Kanten vorstellen kann. Führe zwei weitere neue Knoten ein, die Quelle (engl.: source), $s$, und die Senke (engl.: target), $t$. Von der Quelle führt zu jedem Mittelknoten eine gerichtete Kante mit Kapazität 1, und von jedem ursprünglichen Knoten $v \in V$ führt eine gerichtete Kante mit Kapazität $c(v)$ zur Senke. Fertig ist das Netzwerk. Abbildung 10 zeigt das Fluss-Netzwerk für das einfache Beispiel aus Abbildung 9.

Wir suchen jetzt einen möglichst großen Fluss im Netzwerk von der Quelle zur Senke, wobei die Kapazitäten der Kanten nicht überschritten werden dürfen – formal reden wir von einem maximalen ganzzahligen Fluss von $s$ nach $t$. Wird eine „Flusseinheit“ von $s$ zu einem Mittel-

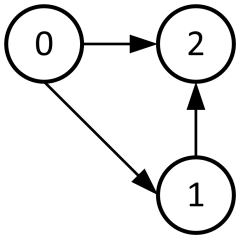


Abbildung 12: Eine Lösung des Beispiel-Problems

knoten geschickt – mehr geht nicht –, muss diese Flusseinheit entweder in die eine oder in die andere Richtung weiterlaufen, was wir als die Entscheidung interpretieren, die entsprechende Kante im ursprünglichen Netzwerk in genau dieser Weise zu richten. Die Kapazitäten der Kanten von ursprünglichen Knoten nach $t$ stellen sicher, dass kein Knoten zu viele einmündende Kanten erhält, und da der Fluss maximal ist, werden so viele Kanten wie überhaupt möglich gerichtet. Genau wenn ausnahmslos alle ursprünglichen Kanten vom maximalen Fluss gerichtet werden, hat das Ausgangsproblem eine Lösung, und die Richtung aller Kanten bzw. die Ausrichter aller Feste können an den Flussrichtungen direkt abgelesen werden. Abbildung 11 zeigt einen maximalen Fluss im Beispiel-Netzwerk, Abbildung 12 den entsprechenden gerichteten Graphen und die dadurch gegebene Lösung des Beispiel-Problems.

## 5.2 Lösungsansatz 2: Schrittweise Verbesserung

Für den zweiten Lösungsansatz muss man nichts über Flüsse in Netzwerken wissen. Wir fangen damit an, jede ungerichtete Kante beliebig zu richten; wir wählen also in irgendeiner Weise eine der beiden möglichen Richtungen. Dann berechnen wir für jeden Knoten $v \in V$ die Differenz $d(v) = a(v) - c(v)$, wobei $a(v)$ die Anzahl der gerichteten Kanten ist, die im Augenblick in $v$ münden. Die Größe $d(v)$ gibt an, wieviele Kanten mehr als „erlaubt“ in $v$ münden. Gilt $d(v) \leq 0$ für jeden Knoten $v$, hat unser zufälliges Richten also bereits zu einer Lösung geführt. Sonst müssen wir unsere Kantenrichtungen schrittweise verbessern.

Wenn wir noch keine Lösung haben, dann sei $v$ ein beliebiger Knoten mit $d(v) > 0$ und sei $U$ die Menge derjenigen Knoten, von denen aus $v$ erreicht werden kann (bei den momentanen Kantenrichtungen). $U$ kann z.B. mit einer in $v$ startenden Tiefensuche gefunden werden, die die Kanten in Rückwärtsrichtung durchläuft; dabei können wir auch einen Tiefensuchbaum mit Wurzel $v$ und Knotenmenge $U$ erstellen, in dem jeder Knoten seinen Vater kennt, was im Folgenden nützlich sein wird. Betrachten wir die Größe $D_U = \sum_{u \in U} d(u)$. Das Umdrehen einer Kante zwischen zwei Knoten in $U$ oder zwischen zwei Knoten außerhalb von $U$ verändert $D_U$ nicht, und das Umdrehen einer Kante zwischen einem Knoten in $U$ und einem Knoten außerhalb von $U$ erhöht $D_U$ um 1, da alle solche Kanten $U$ verlassen. Wenn $D_U$ positiv ist, gilt das also nicht nur für die jetzigen Kantenrichtungen, sondern für alle Kantenrichtungen, die überhaupt möglich sind. Aber dann muss es in $U$ immer, egal, was wir tun, einen Knoten $u$ mit $d(u) > 0$ geben, woraus wir sehen, dass unser Problem keine Lösung hat und wir aufgeben können.

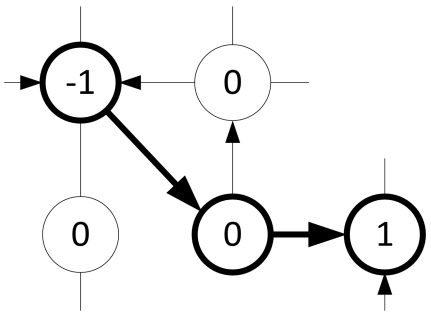


Abbildung 13: Zwei Knoten mit negativem und positivem $d$-Wert in einem ausschnittsweise dargestellten Beispielgraphen. Der Knoten mit positivem $d$-Wert ist von jenem mit negativem $d$-Wert aus erreichbar.

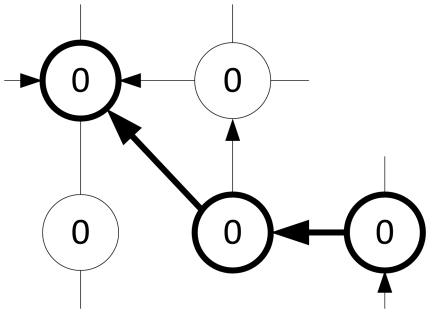


Abbildung 14: Die Umkehrung der Kantenrichtungen löst den Konflikt und lässt Knoten im Inneren des Pfades unbeeinflusst.

Nehmen wir also an, dass $D_U \leq 0$ gilt. Da $d(v) > 0$ ist, muss es in $U$ dann auch einen Knoten $u$ mit $d(u) < 0$ geben. Wir suchen jetzt einen beliebigen Pfad von $u$ nach $v$ (z. B. mit Hilfe des oben erwähnten Tiefensuchbaums) und drehen alle Kanten auf diesem Pfad um. Das verringert $d(v)$ um 1 (was Fortschritt bedeutet), erhöht $d(u)$ um 1 (was nicht stört) und lässt alle anderen $d$-Werte unverändert. Wenn dann noch positive $d$-Werte vorhanden sind, machen wir in derselben Weise weiter.

Sei $m = |E|$ die Anzahl der Kanten in $G$, und nehmen wir der Einfachheit halber an, dass $G$ zusammenhängend ist – wenn nicht, kann das Problem unabhängig auf jeder Zusammenhangskomponente von $G$ gelöst werden. Die Summe $P$ der positiven $d$-Werte kann nie, bei keinen Kantenrichtungen, größer als $m$ sein, da jede Kante höchstens 1 zu $P$ beiträgt. Jedes Umdrehen eines Pfades, wie oben beschrieben, verringert $P$ um 1, weswegen diese Prozedur höchstens $m$-mal ausgeführt werden muss. Die Bestimmung eines Pfades von einem Knoten mit negativem $d$-Wert zu einem Knoten mit positivem $d$-Wert kann in $O(m)$ Zeit erfolgen, so dass sich der Algorithmus insgesamt in $O(m^2)$ Zeit ausführen lässt.

## 5.3 Ergebnisse

Abbildung 15 zeigt eine gültige Verteilung der Feste für das in der Aufgabenstellung gegebene Beispiel.

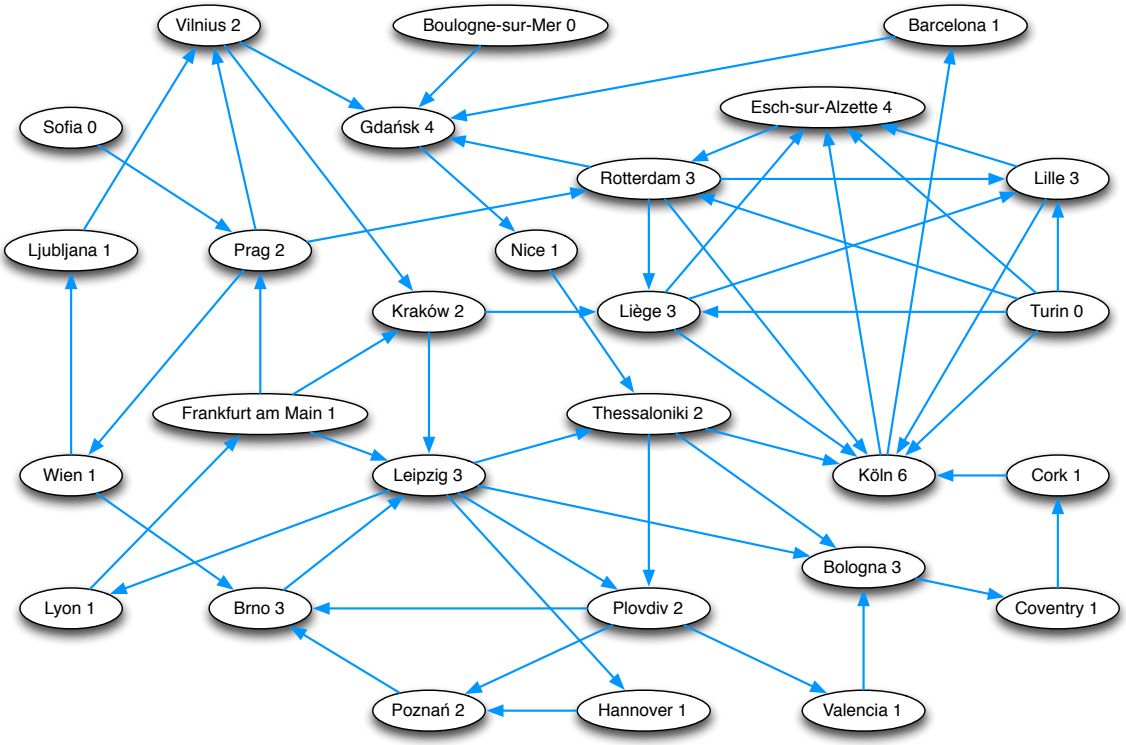


Abbildung 15: Festverteilung im Beispiel der Aufgabenstellung

## 5.4 Bewertungskriterien

- Für Greedy-Ansätze oder heuristische Verfahren sollten deren Schwächen erkannt und beschrieben sein. Insbesondere sollte nicht fälschlich behauptet werden, dass das Verfahren (garantiert) optimale Ergebnisse liefert.
- Weder das gewählte Verfahren noch seine Implementierung sollten Fehler aufweisen.
- Auch für das große Beispiel gibt es eine gültige Festverteilung. Das gewählte Verfahren muss effizient genug sein, ein Beispiel dieser Größe zu bearbeiten.
- Die Lösung für das in der Aufgabenstellung gegebene Städtepartnernetz muss nachvollziehbar dargestellt sein (etwa als gerichteter Graph, tabellarisch, ...). Außerdem muss erwähnt werden, ob eine Lösung für das große vorgegebene Beispiel berechnet wurde; wenn ja, sollte diese elektronisch eingereicht worden sein.
- Die Beispiele sollten korrekt gelöst sein. Es ist aber in der Regel nur das kleine Beispiel genau überprüft worden.

# Aus den Einsendungen: Perlen der Informatik

## Allgemeines

*Worte des Wettbewerbs:* Mittelpunk, Wahrscheinlichtstabelle, gespreichert, Mutterrechteck, algyrithmus, copy & past, GUI (Gasfische Benutzer Oberfläche)

Wie man im Schülerlexikon Wikipedia nachlesen kann …

unendlich $= 2^{64}$

Die direkte Lösung der Aufgabe erfolgt im gelösten Zustand des Programms.

Wenn es sich um eine Programmieraufgabe handelt, eignet sich für diesen Aufgabentyp besonders gut die Brute-Force-Methode.

Genau das habe ich versucht dem Programm beizubringen.

Dies ergibt sich aus dem unten erläuterten Eingabeformat und ließe sich recht leicht erhöhen, stünde der Autor nicht unter Zeitdruck.

Die Vorschleife dreht das Rad.

Sie können mich wie auf Seite *Fehler! Textmarke nicht definiert.* beschrieben kontaktieren.

Da mein Mathelehrer auch keine bessere Lösung wusste …

Da ich so viel nebenbei machen muss, also z.B. Schule =), war die Zeit eigentlich mein größtes Problem bzw. auch größter Gegner.

Also war wieder denken nötig …

## Rechteckschoner

Ich habe mich für den Datentyp Word entschieden, da die Auflösung der gängigsten heutigen Bildschirme sowohl in der Höhe als auch in der Breite deutlich unter 65536 Pixeln liegt, es sei denn, Moni will mit ihrem Programm übermäßig angeben und es an ihre Schule beamen.

Zufällig muss bestimmt werden, ob die Tochtergeneration (TG) die Höhe und Breite der Muttergeneration (MG) übernimmt.

`Render-All` hat die primitive Aufgabe, die Objekte linear durchzukauen.

Ein Struktur beschreibendes Objekt besitzt die Eigenschaften zur geographischen Allokation auf den weiten wilden Welten (www) des Bildschirms.

Leider kann in diesem Fall auf die sozialen Gesellschaftsbedürfnisse der Struktur nicht geachtet werden. Deshalb wird ausdrücklich vor psychologischen Problemen gewarnt, die eventuell zu Persönlichkeitsstörungen führen könnten. Arme Rechteckstruktur.

## Glücksrad

```
probabilities xs =
  sum.map.(product.map((\i->xs!!(i-1).flip mod (n+1)).delta.(0:))
  $ permutations[1..n] where n=length.xs - 1}
```
*Lösung in Haskell*

Sinnlos für den Veranstalter wäre ein Glücksrad mit nur einem Feld. Man kann sich kaum vorstellen, dass der Veranstalter bei jedem Dreh einen Millionengewinn ausschütten möchte.

Die in der Aufgabenstellung beispielhaft angegebene Wahrscheinlichtstabelle für 6 Felder hat einen merkwürdigen Nenner 15.

### Wer spielt fair?

„Ab heute nicht mehr von Computern betrügen lassen!“ *Überschrift der Bearbeitung*

Der Computer könnte zusammen mit dem Benutzer „Schere, Stein, Papier“ rufen.

Ein weiteres Kriterium, das zum Vergleich der verschiedenen Strategien genutzt werden kann, ist die Ästhetik.

Wenn der Spieler einige Male kontrolliert hat, wird er wahrscheinlich genug Vertrauen haben und nicht mehr kontrollieren, was den Spielfluss in Gang hält.

Man verwendet also die „Public Key Cryptography“ an. Möchte man 100% sicher gehen, dass der Computer nicht schummelt, verwendet man zusätzlich das RSA-Verfahren.

Es erscheint die Meldung „Computer hat seine Eingabe gemacht“ (als Beweis, dass der Computer seine Wahl vorher festgelegt hat).

Da AES ein offener Standard mit verfügbaren Implementierungen ist, ist die Überprüfung des Verfahrens auch für den Nutzer praktikabel.

Lieber Bob, für dein Problem ist eine Einwegverschlüsselung gut geeignet.

Zählen ist eine Fähigkeit, die ein Anwender besitzen sollte, da er sonst sowieso beim Schere-Stein-Papier spielen betrogen werden kann.

Zahlen sind nicht schön.

### Aladins Lampen

Den meisten Menschen ist modulare Arithmetik suspekt.

Diese Folge wurde empirisch ermittelt.

Er hat damit eine Laufzeit von $n!$ (z.B. bei 10 Feldern sind das 4 Minuten und 32 Sekunden).

Dann wird intern beliebig oft zufällig ein Schalter betätigt. Damit ist für den User gewährleistet, dass die Schaltung lösbar ist. Ob er sie lösen kann, ist eine andere Frage.

Schaltung 2 konnte von meinem Algorithmus nicht gelöst werden und ist somit nicht lösbar.

```
// Schalter + Lampe = Schlampe
```

Mit größeren Datentypen wäre jedoch allein die Geduld des Anwenders obere Grenze für die Größe der Schaltung.

### Fehlerfrei puzzeln

... für das Zeichnen des Puzzleteils auf einer GUI (wenn man die pain()-Methode richtig anpasst).

Die Farbanordnung liegt dann jedoch des weiteren mathematisch völlig inkorrekt in der Datei vor. Ich muss, glaube ich, nicht extra die Schmerzen erwähnen, die jedem Mathematiker/Informatiker beim Erkennen dieser Tatsache heimsuchen.

Start → Kacheln mit je 4 Farben → Endlosschleife → Passendes Stück finden ... → Ende
*Transkript eines Flussdiagramms*

Das Programm kann nicht mit Bestimmtheit als korrekt bezeichnet werden, da es nie zu einer fertigen Lösung kam.

Es ist hilfreich, einige Teile manuell zu vertauschen und dann das Puzzle neu aufzukochen.

## Zaras dritter Fehler

Da wir Informatiker sind, schreiben wir für die Suche nach Zaras Sätzen natürlich ein Programm.

Die Methode gibt den gereinigten Text zurück.

Die Idee für das Programm ist, wie so oft, recht einfach.

Der Quellcode ist weitaus komplizierter als die vereinfachte Beschreibung.

Punkt 1 des Lösungsverfahrens wird nicht näher beschrieben, da dieser kein bewertenswertes Maß an zielführenden Überlegungen beinhaltet.

Wovon das Herz voll ist, da ücainaeecoeienpmda.

## Städtepartner

Partnerschaf

Klar ist jedoch, dass ein auf Ausprobieren basierendes Verfahren bei einer Komplexität von $O(n^n)$ bei $n$ Städten wohl kaum in absehbarer Zeit zum Erfolg führen würde.

Aus Übersichtlichkeitsgründen und aus Mitleid mit dem vor meinem Fenster befindlichen Baum wird die komplette Lösung des Graphen nicht abgedruckt, sondern auf der beiliegenden CD abgelegt.

Irgendwann ist man aber an dem Punkt angekommen, an dem alle Trivialfälle gelöst sind und man vor der großen Frage steht, wie es nun weiter geht.

Es wurde die bestmögliche, wenn auch nicht optimale Lösung gefunden.

Nach einigen Tests zur Fehlerbehebung entdeckte ich: die Fehleranfälligkeit des Algorithmus steigt proportional zur Komplexität des Graphen.

Im Endeffekt handelt es sich bei dem Problem um dein Graphenproblem.

Das war's dann mit logischer Zuordnung, den Rest machen wir zufällig.

Die Lösung besteht also aus zwei Algorithmen, einem korrekten und einem heuristischen.

Sie ignoriert Städte, die null Party organisieren können und null offene Partnerschaften haben, die also „wunschlos glücklich“ sind.