BWINF – Tipps und Tricks

Felix Arends Tobias Polley

13.Dezember 2010

Vorwort

Herzlichen Glückwunsch. Da Du dieses Dokument gefunden hast, hast Du schon Engagement für Deine BWINF-Einsendung bewiesen. Das ist sehr gut. Denn eines können wir als ehemalige Teilnehmer garantieren: Engagement braucht es.

Mit diesem Dokument wollen wir Dir helfen, Deine künftigen Einsendungen zu verbessern. Das Dokument ist voll von konkreten Tipps, aber auch dezenteren Hinweisen oder auch Verweisen auf zahlreiche Bücher. Natürlich müssen nicht alle exakt befolgt werden, aber meistens stellen sie eine gute Richtlinie dar. Zwei Sachen liegen uns besonders am Herzen:

- **Wenn Du eine Stelle dieses Dokuments nicht verstehst, gib nicht auf!**
  Spätestens im Studium geht es einem bei Fachliteratur fast ständig so. Die üblichen Strategien in diesem Fall beinhalten nochmaliges Lesen, Beispiele nachrechnen, nochmaliges Lesen am nächsten Tag, zeitweiliges Überspringen[1], Freunde fragen, Lehrer zu Rate ziehen und zu allerletzt: bei den Autoren nachfragen.

- Wir haben an dieses Dokument einen extrem hohen Qualitätsanspruch gestellt, und es in einem Zeitraum von 10 Monaten erstellt. Wir sind beide ehemalige Teilnehmer, und haben ein abgeschlossenes Hochschulstudium bzw. stehen kurz davor. Wir haben uns jahrelang bei der deutschen Qualifikation zur Internationalen Informatikolympiade engagiert, zunächst als Teilnehmer, dann als Coaches und Teamleiter. **Gehe also nicht davon aus, daß unsere Einsendung repräsentativ für den Durchschnitt der Teilnehmer ist!**
  Wir haben einen großen Teil der Zeit investiert, um zu versuchen, eine möglichst perfekte Einsendung zu erstellen.

In Teil I stellen wir eine echte, vollständige Einsendung zum Bundeswettbewerb Informatik (BWINF) vor, die die Aufgabe „Drehzahl“ der ersten Runde des 29. BWINF bearbeitet.

In Teil II listen wir, als Kern dieses Dokuments, eine Reihe von Hinweisen auf, die Dinge beschreiben, die beim Erstellen von Einsendungen zu tun oder zu unterlassen sind.[2]

Wir haben uns bemüht, möglichst allgemeinverständliche Sprache zu verwenden. Auch wenn die Lösungsidee fast schon komplett allein durch die Formeln (1), (2) und (3) beschrieben wird, haben wir diese auch in (kursiv gedruckten) deutschen Sätzen formuliert, so dass ein Verständnis der mathematischen Notationen nicht notwendig ist.

Dieses Dokument sollte mindestens einmal komplett von vorne nach hinten durchgelesen werden, um sicherzustellen, dass alle Begriffe bekannt sind. Je nach Vorkenntnissen empfehlen wir, Anhang A (die Aufgabenstellung) und B (Mathematische Notationen) zuerst zu lesen.

Wir danken Thomas Leineweber und Ludwig Schmidt für zahlreiche Hinweise. Wir danken natürlich auch dem BWINF und seinem Geschäftsführer, Herrn Dr. Pohl: Ohne sie wären wir in unserem Leben nicht dort, wo wir sind. Ohne sie gäbe es auch nicht dieses Dokument. Vielen Dank!

Tobias Polley
Bonn
Dezember 2010

[1] An dieser Stelle sei erwähnt, daß man Teil I oder auch Kapitel 6 komplett überspringen kann.
[2] Dabei ist dies einzig und allein unsere Meinung als langjährige Bewerter.

Inhaltsverzeichnis

Augensumme $k = x + y$	Augenkombinationen geordnete Paare (x, y)	Wahrscheinlichkeit $\mathbb{P}(W = k)$
2	$(1,1)$	$\frac{1}{36} = 0{,}02\overline{7}$
3	$(1,2)$, $(2,1)$	$\frac{2}{36} = 0{,}05\overline{5}$
4	$(1,3)$, $(2,2)$, $(3,1)$	$\frac{3}{36} = 0{,}08\overline{3}$
5	$(1,4)$, $(2,3)$, $(3,2)$, $(4,1)$	$\frac{4}{36} = 0{,}11\overline{1}$
6	$(1,5)$, $(2,4)$, $(3,3)$, $(4,2)$, $(5,1)$	$\frac{5}{36} = 0{,}13\overline{8}$
7	$(1,6)$, $(2,5)$, $(3,4)$, $(4,3)$, $(5,2)$, $(6,1)$	$\frac{6}{36} = 0{,}16\overline{6}$
8	$(2,6)$, $(3,5)$, $(4,4)$, $(5,3)$, $(6,2)$	$\frac{5}{36} = 0{,}13\overline{8}$
9	$(3,6)$, $(4,5)$, $(5,4)$, $(6,3)$	$\frac{4}{36} = 0{,}11\overline{1}$
10	$(4,6)$, $(5,5)$, $(6,4)$	$\frac{3}{36} = 0{,}08\overline{3}$
11	$(5,6)$, $(6,5)$	$\frac{2}{36} = 0{,}05\overline{5}$
12	$(6,6)$	$\frac{1}{36} = 0{,}02\overline{7}$

Tabelle 1: Wahrscheinlichkeiten für die Augensumme zweier Würfel

Teil I
Die Einsendung

1 Vorwissen

Beim Lösen dieser Aufgabe kommt man um ein bisschen Wahrscheinlichkeitsrechnung nicht herum. Insbesondere verwenden wir häufig die Wahrscheinlichkeit, dass die Summe zweier Würfel eine bestimmte Zahl k ergibt. *Die Summe zweier Würfel ist die Summe zweier unabhängiger, gleichverteilter, ganzzahliger Zufallsvariablen x und y mit Werten von 1 bis 6: $k = x + y$.*

Die Wahrscheinlichkeit, dass ein Würfel eine bestimmte Zahl x zeigt, ist $\frac{1}{6} = 0{,}1\overline{6}$.

Die Wahrscheinlichkeit, dass der erste von zwei Würfeln eine bestimmte Zahl x, und der zweite y zeigt, ist $\frac{1}{6} \cdot \frac{1}{6} = \frac{1}{36} = 0{,}02\overline{7}$, da die beiden Würfel *unabhängig* voneinander sind. Man beachte hierbei, dass die Reihenfolge in (x, y) relevant ist. Tabelle 1 zeigt die Wahrscheinlichkeiten für die möglichen Werte der Summe. Wir schreiben $\mathbb{P}(W = k)$ für die Wahrscheinlichkeit, dass eine Augensumme von k gewürfelt wird.

$\mathbb{P}(W = k)$

2 Lösungsidee

Eine *Karte* wird durch eine natürliche Zahl von 1 bis 9 dargestellt. Eine *Kartenmenge* ist eine Menge von Karten. Ein Spielzustand wird eindeutig durch die Kartenmenge der noch nicht umgedrehten Karten beschrieben. Diese Karten bezeichnen wir als die *offenen Karten*. Wir schreiben $P(H)$ für die erwartete Punktzahl einer Kartenmenge H, wenn genau die Karten in H offen sind und der Spieler „optimal“ spielt. Zum Beispiel ist $P(\{2\}) = 43 + 2 \cdot \frac{1}{36}$, da dem Spieler bereits $43 = 1 + 3 + 4 + 5 + 6 + 7 + 8 + 9$ Punkte sicher sind und er zwei weitere Punkte genau dann erhält, wenn als nächstes eine Zwei gewürfelt wird ($= 2 \times \boxdot$), was mit einer Wahrscheinlichkeit von $\frac{1}{36}$ geschieht, er also die verbleibende Karte 2 umdrehen kann.

Karte
Kartenmenge
offenen Karten
$P(H)$

Optimal zu spielen bedeutet, immer genau die Karten herumzudrehen, die die danach zu erwartetende Punktzahl maximieren (unter der Annahme, dass man weiterhin optimal spielt).

Optimal spielen

Zunächst definieren wir die Menge $Z(H)$ der *erlaubten Züge*, wenn der Spieler Kar-

$Z(H)$

tenmenge H besitzt. $Z(H)$ ist also eine Menge von Kartenmengen. Zum Beispiel ist

$$Z(\{1,2,3\}) = \{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\}\} \quad .$$

Jede Kartenmenge in $Z(H)$ enthält demnach entweder eine oder zwei Karten. Alle Karten, die in einer Kartenmenge aus $Z(H)$ vorkommen, müssen auch in H vorhanden sein.

$Z(H)$ ist also die Menge aller Teilmengen von H, die entweder 1 oder 2 Elemente haben:

$$Z(H) := \{A \subseteq H \,|1 \leq |A| \leq 2\}$$

Ist nun die Summe W der beiden Würfelwürfe bekannt, lässt sich die Menge der zulässigen Kartenmengen weiter einschränken. Wir schreiben $Z(H,W)$ für die Menge der Kartenmengen aus $Z(H)$, deren Summe W ergibt. W $Z(H,W)$

$Z(H,W)$ ist also die Menge aller Teilmengen von H, die entweder 1 oder 2 Elemente haben und summiert W ergeben:

$$Z(H,W) := \left\{ A \subseteq H \,\middle|\, 1 \leq |A| \leq 2 \text{ und } \sum_{j \in A} j = W \right\} \tag{1}$$

Zum Beispiel ist

$$Z(\{1,2,3\},3) = \{\{3\},\{1,2\}\} \quad .$$

Damit können wir nun relativ einfach eine Formel für die erwartete Gesamtpunktzahl $P(H,W)$ aufstellen, wenn die Kartenmenge H offen ist, die Augensumme der Würfel W beträgt und der Spieler optimal spielt: *Der optimale Zug bei den offenen Karten H und Würfelsumme W ist derjenige, der die erwartete Punktzahl in der durch ihn erreichten Situation maximiert. Ist kein Zug mehr möglich, entspricht $P(H,W)$ der aktuell erreichten Punktzahl.* $P(H,W)$

$$P(H,W) = \begin{cases} 45 - \sum_{k \in H} k & \text{wenn } Z(H,W) = \{\} \\ \max_{K \in Z(H,W)} P(H \setminus K) & \text{andernfalls.} \end{cases} \tag{2}$$

Der Ausdruck $H \setminus K$ stellt die Kartenmenge H ohne die Karten aus K dar. $H \setminus K$

Mit diesen Definitionen kann die **Aufgabenstellung** wie folgt umformuliert werden:

1. Gegeben H und W, für welches $K \in Z(H,W)$ ist $P(H,W) = P(H \setminus K)$?
2. Was ist der Wert von $P(\{1,\ldots,9\})$?

Um $P(H,W)$ tatsächlich zu berechnen, müssen wir noch eine Formel für $P(H)$ angeben. Dieser Wert muss alle möglichen Würfelwürfe nach ihrer Wahrscheinlichkeit gewichten. Da nach dem Würfelwurf optimal weitergespielt wird (also eine Rekursion entsteht), ist die erwartete Gesamtpunktzahl genau $P(H,W)$, sobald W bekannt ist. Dabei nimmt W Werte zwischen 2 und 12 an, siehe Tabelle.

$$P(H) = \mathbb{P}(W=2) \cdot P(H,2) + \ldots + \mathbb{P}(W=12) \cdot P(H,12)$$

kurz

$$= \sum_{k=2}^{12} \mathbb{P}(W=k) \cdot P(H,k) = \mathbb{E}_W\left(P(H,W)\right) \tag{3}$$

Hier ist der Pseudocode zur Berechnung beider Funktionen:

```
1  // Wahrscheinlichkeiten für die Summe zweier Würfel, siehe oben
2  probability[13] = { 0, 0, 1 / 36, ..., 2 / 36, 1 / 36 }
3
4  FUNCTION P(H, k):
5    result = Summe der Karten nicht in H // Game over
6
7    // eine Karte
8    IF (k ∈ H)
9      result = P(H \ {k})
10
11   // zwei Karten
12   FOR a ∈ H
13     b = k − a // zweite Karte
14     IF b ∈ H and b > a
15       IF P(H \ {a, b}) > result
16         result = P(H \ {a, b})
17
18   RETURN result
19
20
21 FUNCTION P(H):
22   result = 0
23   FOR roll = 2 .. 12
24     result = result + probability[roll] * P(H,roll)
25   RETURN result
```

Wie zu erwarten rufen sich die beiden Funktionen gegenseitig auf. Die Ausführung terminiert, weil:

- $P(H, k)$ die Funktion $P(H)$ nur mit weniger Karten in H aufruft
- $P(H)$ die Funktion $P(H, k)$ nie mit mehr Karten in H aufruft

Schließlich lässt sich noch beobachten, dass es für die Funktion P nur $512 = 2^9$ verschiedene Eingabewerte gibt (alle möglichen Teilmengen von $\{1, \ldots, 9\}$). Der Rückgabewert kann nach der ersten Berechnung also gespeichert werden, was die Laufzeit der Berechnungen erheblich verbessert. Auf diese Art und Weise wird $P(H)$ nur höchstens 512 Mal berechnet. Dementsprechend wird $P(H, k)$ höchstens $11 \cdot 512 = 5632$ Mal berechnet.

2.1 Laufzeitanalyse

Da die Anzahl der Karten und die der Würfel in der Aufgabenstellung fest vorgegeben ist, ist die Laufzeit zunächst einmal konstant[3]. Wir können allerdings zwei künstliche Parameter einführen, nämlich die Anzahl der Karten n und die Anzahl der Würfel m. n
Wir argumentieren ähnlich wie zuvor. Um die zu erwartende Punktzahl bei einem Spiel m
mit n Karten auszurechnen, muss $P(H)$ höchstens für alle 2^n Teilmengen der Karten ausgewertet werden. Deswegen muss $P(H, W)$ höchstens $2^n \cdot 6 \cdot m$ Mal ausgewertet werden[4].

Wir setzen voraus, dass die Grundrechenarten in konstanter Zeit durchführbar sind und dass Zahlen konstant viel Speicher belegen, wie das zum Beispiel bei `int` und `double` immer der Fall ist. Die Laufzeit von $P(H)$ ist nun ohne Berücksichtigung des rekursiven Aufrufs $\mathcal{O}(m)$, wenn die Wahrscheinlichkeiten vorberechnet sind. Ohne Berücksichtigung des rekursiven Aufrufs liegt die Laufzeit von $P(H, W)$ in $\mathcal{O}(n \cdot m)$. Der Faktor $\mathcal{O}(m)$ resultiert von den Mengenoperationen, die auf H ausgeführt werden müssen, bzw. von der Zeit, die es kostet H für den rekursiven Aufruf zu kopieren.

[3] Auch wenn H die Laufzeit natürlich beeinflusst, interessiert das nicht, denn mit $H = \{1, ..., 9\}$ wird der *worst case* erreicht, für den unsere im Folgenden angegebene Schranke gilt.

[4] Genaugenommen sind die gültigen Werte für k nur $m, \ldots, 6m$, also $5m + 1$ verschiedene Werte.

Insgesamt hat die Berechnung von $P(H)$ damit eine Laufzeitkomplexität von $\mathcal{O}(2^n \cdot m^2 \cdot n)$ bei einer Speicherkomplexität von $\mathcal{O}(2^n)$ (wir setzen voraus, dass das Zwischenspeichern und Auslesen der Ergebnisse in konstanter Zeit möglich ist).

3 Programmdokumentation

Zum Speichern von Kartenmengen haben wir eine Datenstruktur entwickelt, die nur mit bis zu 32 Karten funktioniert, dafür aber sehr effizient ist. Dazu wird eine Kartenmenge als Bitvektor dargestellt. Die Datenstruktur ist mit der Klasse Cards abstrahiert. Intern wird ein **unsigned int** verwendet, in dem das i-te Bit genau dann gesetzt ist, wenn die Karte mit dem Wert i in der Kartenmenge vorhanden ist.

Wenn a und b Objekte vom Typ Cards sind, dann ist a.With(b) die Vereinigungsmenge beider Kartenmengen und a.Without(b) ist die Kartenmenge a ohne die Karten in b. Der Ausdruck a.ContainsCard(c) gibt **true** zurück, wenn die Karte mit dem Wert c in a enthalten ist. Alle drei Operationen können mit Bitarithmetik implementiert werden:

```
1  // Returns a new set without the given cards.
2  Cards Without(const Cards& other) const {
3    return Cards(cards_ ^ (cards_ & other.cards_));
4  }
5
6  // Returns a new set including the given cards.
7  Cards With(const Cards& other) const {
8    return Cards(cards_ | other.cards_);
9  }
10
11 bool ContainsCard(int card) const { return (cards_ >> card) & 1; }
```

Um zu überprüfen, ob die Kartenmenge leer ist, muss der **unsigned int** lediglich mit 0 verglichen werden.

Die Funktion $P(H)$ aus der Lösungsidee ist als ExpectedScore implementiert. Für das Zwischenspeichern der Rückgabewerte wird eine std::map verwendet (das Abspeichern und Auslesen kostet hier also $\mathcal{O}(2^n) = \mathcal{O}(n)$ statt $\mathcal{O}(1)$ Zeit).

Für die Funktion $P(H, W)$ gibt es keine direkte Entsprechung im Quelltext. Stattdessen berechnet BestNextMove(cards, roll) die Kartenmenge K, die umgedreht werden sollte (und darf), sodass $P(H \setminus K)$ maximal ist, wenn die Kartenmenge cards sichtbar auf dem Tisch liegt und die Augensumme der Würfel roll beträgt. Ansonsten folgt die Implementation dem Pseudocode.

4 Programm-Ablaufprotokoll

Unser Programm terminiert für alle getesteten Eingaben im Millisekundenbereich (ungefähr 10 ms auf einem gewöhnlichen PC).

4.1 Beispiel

Im ersten Paragraphen der Lösungsidee gaben wir das Beispiel $P(\{2\}) = 43 + \frac{2}{36} = 43{,}0\overline{5}$. Wir lassen unser Programm zweimal laufen und geben zuerst an, dass eine 4 gewürfelt wurde, und anschließend, dass eine 2 gewürfelt wurde. Wie erhofft sind die zu erwartenden Punktzahlen 43,0556 bzw. 45.

```
1 Welche Karten halten Sie in der Hand?
2 Bitte geben Sie eine durch Leerzeichen getrennte Liste ein:
3 2
4 Der Erwartungswert der Punktzahl lautet: 43.0556
5 Welche Summe wurde gewuerfelt?
6 4
7 Das Spiel ist zuende.
```

k	$\mathbb{P}(W = k)$	K	$P(\{1,2,3\} \setminus K)$
2	$\frac{1}{36}$	$\{2\}$	41,5
3	$\frac{2}{36}$	$\{3\}$	42,2222
		$\{1,2\}$	42,1667
4	$\frac{3}{36}$	$\{1,3\}$	43,0556
5	$\frac{4}{36}$	$\{2,3\}$	44
$6 \dots 12$	$\frac{26}{36}$	*keine möglichen Züge*	

Tabelle 2: Wahrscheinlickeiten und zu erwartende Punktzahlen für die Folgezüge von $H = \{1,2,3\}$

```
1 Welche Karten halten Sie in der Hand?
2 Bitte geben Sie eine durch Leerzeichen getrennte Liste ein:
3 2
4 Der Erwartungswert der Punktzahl lautet: 43.0556
5 Welche Summe wurde gewuerfelt?
6 2
7 Spielen Sie die Karte(n) 2.
8 Der Erwartungswert lautet nun 45.
```

4.2 Beispiel

Als nächstes demonstrieren wir, dass die Berechnung von $P(H)$ unserer Formel in Gleichung 3 entspricht. Dazu berechnen wir $P(\{1,2,3\})$ indem wir für jede mögliche Würfelsumme k jeden gültigen Zug K betrachten und den jeweils optimalen auswählen. Die Ergebnisse sind in Tabelle 2 festgehalten.

In diesem Beispiel ist der nächste Zug eindeutig, es sei denn, die Würfelsumme beträgt 3. In diesem Fall ist es am besten die Karte mit der 3 herumzudrehen. Aus den berechneten Werten ergibt sich insgesamt

$$P(\{1,2,3\}) = \frac{1}{36} \cdot 41{,}5 + \frac{2}{36} \cdot 42{,}2222 + \frac{3}{36} \cdot 43{,}0556 + \frac{4}{36} \cdot 44 + \frac{26}{36} \cdot 39 = 40{,}142$$

In der Tat berechnet unser Programm:

```
1 Welche Karten halten Sie in der Hand?
2 Bitte geben Sie eine durch Leerzeichen getrennte Liste ein:
3 1 2 3
4 Der Erwartungswert der Punktzahl lautet: 40.142
```

4.3 Beispiel

Die Aufgabenstellung fragt außerdem nach dem Erwartungswert für ein Spiel, das optimal gespielt wird. Dieser Erwartungswert ist $P(\{1,2,3,4,5,6,7,8,9\})$ und wird von unserem Programm wie folgt berechnet:

```
1 Welche Karten halten Sie in der Hand?
2 Bitte geben Sie eine durch Leerzeichen getrennte Liste ein:
3 1 2 3 4 5 6 7 8 9
4 Der Erwartungswert der Punktzahl lautet: 33.0361
```

4.4 Beispiel

Zuletzt geben wir noch an, welche Karten am Anfang eines Spiels umzudrehen sind, wenn eine bestimmte Augensumme gewürfelt wurde (siehe Tabelle 4.4).

Augensumme	Optimaler Zug	Erwartete Punktzahl
2	2	28,8138
3	3	30,1165
4	4	30,5934
5	5	31,7285
6	6	32,5017
7	7	33,3038
8	8	34,6778
9	9	36,1272
10	1 9	33,546
11	2 9	33,2141
12	3 9	34,2641

Tabelle 3: Optimale Züge am Anfang eines Spiels

Die optimale Strategie zu Beginn eines Spiels ist recht einfach zu beschreiben: Beträgt die Würfelsumme 9 oder weniger, so dreht man die Karte mit dem Wert, der der Würfelsumme entspricht, herum. Andernfalls wählt man 9 und die entsprechende zweite Karte.

5 Programm-Text

5.1 cards.h: Datenstruktur für Kartenmengen

```
1  #ifndef __CARDS_H__
2  #define __CARDS_H__
3
4  // The data structure used throughout the program to represent a set of cards.
5  // It supports constant time adding and removing of cards, as well as checking
6  // whether the set is empty or contains a given card.  The < operator is
7  // overloaded so that card sets can be used in combination with std::map.
8  class Cards {
9   public:
10    Cards() : cards_(0) {}
11    explicit Cards(int c) : cards_(1 << c) { }
12    explicit Cards(int c1, int c2) : cards_((1 << c1) | (1 << c2)) { }
13
14    // Returns a new set without the given cards.
15    Cards Without(const Cards& other) const {
16      return Cards(cards_ ^ (cards_ & other.cards_));
17    }
18
19    // Returns a new set including the given cards.
20    Cards With(const Cards& other) const {
21      return Cards(cards_ | other.cards_);
22    }
23
24    bool ContainsCard(int card) const { return (cards_ >> card) & 1; }
25    bool IsEmpty() const { return cards_ == 0; }
26    bool operator < (const Cards& other) const { return cards_ < other.cards_; }
27
28   private:
29    Cards(unsigned int cards) : cards_(cards) {}
30
31    unsigned int cards_;
32 };
33
34 #endif  // __CARDS_H__
```

5.2 drehzahl_strategy.cpp: Implementation der optimalen Strategie

```
1  // This file contains the actual algorithms for computing the next moves and
2  // the expected final score for a given hand.
3
4  #include "drehzahl_strategy.h"
5
6  #include <algorithm>
7  #include <map>
8
9  #include "cards.h"
10
11 // Computes the probability that a sum of "roll" is rolled when throwing two
12 // dice.
13 static double Probability(int roll) {
14   const int min_dice_1 = std::max(1, roll - 6);
15   const int max_dice_1 = std::min(roll - 1, 6);
16   return 1.0 * (max_dice_1 - min_dice_1 + 1) / 36;
17 }
18
19 // Computes the sum of the played cards in the range [1 .. 9],
20 // given the set of remaining cards.
21 static double SumOfPlayedCards(const Cards& hand) {
22   double result = 0.0;
23   for (int card = 1; card <= 9; card++) {
24     if (!hand.ContainsCard(card)) {
25       result += card;
26     }
27   }
28   return result;
29 }
30
31 // Returns the cards that are best played next, assuming that the cards in
32 // "hand" are available and a sum of "roll" was thrown with the dice.
33 Cards BestNextMove(const Cards& hand, int roll) {
```

```
34    Cards best_cards;
35    double max_score = 0.0;
36
37    if (hand.ContainsCard(roll)) {
38      best_cards = Cards(roll);
39      max_score = ExpectedScore(hand.Without(best_cards));
40    }
41
42    for (int card_1 = 1; roll - card_1 > card_1; card_1++) {
43      const int card_2 = roll - card_1;
44      if (hand.ContainsCard(card_1) && hand.ContainsCard(card_2)) {
45        const Cards cards(card_1, card_2);
46        const double score = ExpectedScore(hand.Without(cards));
47        if (score > max_score) {
48          max_score = score;
49          best_cards = cards;
50        }
51      }
52    }
53
54    return best_cards;
55  }
56
57  // Computes the expected score conditional on the event that the player has
58  // the cards in "hand".
59  double ExpectedScore(const Cards& hand) {
60    static std::map<Cards, double> cache;
61    if (cache.find(hand) != cache.end()) {
62      return cache[hand];
63    }
64
65    double result = 0.0;
66    const double current_score = SumOfPlayedCards(hand);
67
68    for (int roll = 2; roll <= 12; roll++) {
69      const Cards next_move = BestNextMove(hand, roll);
70      if (next_move.IsEmpty()) {
71        result += Probability(roll) * current_score;  // Game over.
72      } else {
73        result += Probability(roll) * ExpectedScore(hand.Without(next_move));
74      }
75    }
76
77    return cache[hand] = result;
78  }
```

Teil II
Kommentare

6 Wie kommt man auf diese Lösung?

Die Antwort auf diese Frage gehört nicht in die Einsendung!

Nicht den Weg dokumentieren, sondern die Lösung.

Dies ist sowohl für Einsendungen zum BWINF so, als auch in allen mathematischen Beweisen. Man muss zwischen den anfänglichen Überlegungen, die man nach dem Lesen der Aufgabe angestellt hat bis man zur Lösungsidee gekommen ist, und der Argumentation unterscheiden, wieso diese Idee korrekt ist, also funktioniert.

Ersteres ergibt zwar meistens eine nette Geschichte, ist aber nicht relevant. (Gehört also weggelassen.) Letzteres ist natürlich unabdingbar.

Aber: Zum Begründen von Designentscheidung kann das Erwähnen von nicht gewählten Alternativen wichtig sein!

Heuristik?

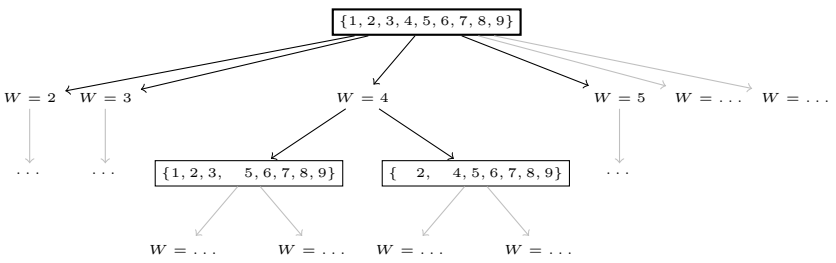
Zum Beispiel wenn es darum geht, sich zwischen einem exakten Lösungsalgorithmus und einer Heuristik[5] zu entscheiden: Zur Begründing, wieso man sich für die Heuristik entschieden hat, gehört, dass man erwähnt, wie die exakte Lösung funktioniert hätte und wieso man sie nicht gewählt hat, etwa „Das Suchen der exakten Lösung würde die Traversierung des kompletten Baumes erfordern, was für ein Baumgröße von $2^N - 1 \approx 10^{12}$ auf heutigen Rechnern unpraktikabel ist.“ Danach folgt dann natürlich die Begründung, wieso die gewählte Heuristik gut sein soll.

Wie kommt man jetzt also auf diese Lösung?

Bei der Aufgabe handelt es sich, wie sogar im Text erwähnt ist, um ein Spiel. Mit ein bisschen Erfahrung würde man damit sofort an „Minimax“[6] (Der Würfel entspricht dem Gegner.) oder „Alpha-Beta-Pruning“[7] denken, doch auch ohne das kommt man nach ein bisschen Überlegen auf die im Folgenden Schritt für Schritt entwickelte Methode, die genau der Minimax-Methode in unserem Anwendungsfall entspricht.

Zunächst kann man sich überlegen, wie die Struktur des Spielverlaufes aussieht. Es ist klar, dass nur die Augensummen (z.B. $W = 4$) und nicht die einzelnen Würfelergebnisse ($\boxdot + \boxdot$, oder $2 \times \boxdot$) für die Struktur des Verlaufs relevant sind.

Das Spiel beginnt mit den Karten $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ in der Hand. Abwechselnd wird gewürfelt und Karten werden umgedreht (d.h. aus der Menge/Hand entfernt). Man könnte an einen Baum denken, der abwechselnd aus Karten- und Augensummenebenen besteht:



Dieser Baum ist unhandlich groß: Er hat etwa $1{,}2 \cdot 10^8$ Knoten. (Wie man darauf kommt, ist leider nicht einfach. Wir haben ein Programm eingesetzt.) Alle Knoten mit heutigen Computern

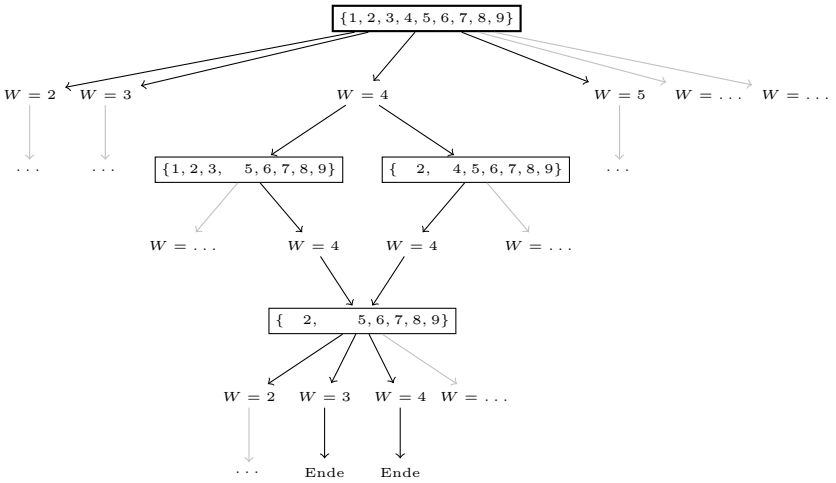
[5] Ein Programm, das versucht, gut zu raten.

[6] `http://de.wikipedia.org/wiki/Minimax-Algorithmus`

[7] `http://de.wikipedia.org/wiki/Alpha-Beta-Suche`

zu betrachten ginge also gerade noch in hinnehmbarer Zeit, da die Taktfrequenzen im Bereich von $3\,\mathrm{GHz} = 3 \cdot 10^9\,\mathrm{Hz}$ liegen.

Zum Glück fällt aber trotzdem auf, dass viele Kartenmengen doppelt vorkommen, zum Beispiel schon in der nächsten Ebene. Durch das Zusammenlegen gleicher Knoten wird so aus dem Baum ein Graph:



(Das „Ende“ ist erreicht, wenn kein Zug mehr möglich ist.)

Die Frage „Wieviele *Kartenknoten* gibt es im Graphen?“ ist nun einfacher zu beantworten: Sie hat dieselbe Antwort wie „Wieviele verschiedene Teilmengen hat die Menge $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$?“. Da jede der Zahlen $1 \ldots 9$ entweder in einer Teilmenge vorkommt oder nicht (2 Möglichkeiten), ist die Antwort $2^9 = 512$.

Die Antwort auf die Frage „Wieviele *Augensummenknoten* gibt es im Graphen?“ ist dann $512 \cdot 11 = 5632$. Diese Anzahl von Knoten kann ein heutiger Computer ohne Probleme verarbeiten.

Um diesen Satz wirklich zu begründen müsste man technisch noch wesentlich mehr ins Detail gehen, etwa mit der Abbildung der Kartenmengen auf **unsigned int**, wie wir das in der Programmdokumentation getan haben. — Auf jeden Fall überzeugt die 5632 erstmal, dass die Verlaufsstruktur geeignet klein ist, und man sich der eigentlichen Fragestellung wieder zuwenden kann.

Die letzte Frage der Aufgabenstellung ist sehr wegweisend: „Welche durchschnittliche Punktzahl pro Spiel kann ein Spieler erzielen, wenn er sehr viele Spiele mit stets optimalen Entscheidungen spielt?“

Die Bedingung „wenn er sehr viele Spiele [...] spielt“ ist entscheidend. Sie rechtfertigt, dass man bei der Berechnung der erreichbaren Punktzahl die Punktzahl nach jedem Würfelergebnis ($x = 1 \ldots 6$) mit einem Gewicht (also Faktor) von $\frac{1}{6}$ versieht und über alle Würfelergebnisse aufaddiert. Für $W = 2 \ldots 12$ gehen wir entsprechend mit den Wahrscheinlichkeiten aus Tabelle 1 vor.

Würde man sich fragen „Wieviele Augen zeigt ein Würfel über sehr viele Würfe durchschnittlich?“, so würde man auf diese Art und Weise

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 4 + \frac{1}{6} \cdot 5 + \frac{1}{6} \cdot 6 = 3,5$$

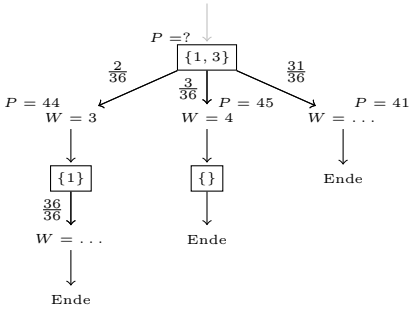
erhalten. Für 2 Würfel würde man mit den Werten aus Tabelle 1 folgendes erhalten:

$$\frac{1}{36} \cdot 2 + \frac{2}{36} \cdot 3 + \frac{3}{36} \cdot 4 + \frac{4}{36} \cdot 5 + \frac{5}{36} \cdot 6 + \frac{6}{36} \cdot 7 + \frac{5}{36} \cdot 8 + \frac{4}{36} \cdot 9 + \frac{3}{36} \cdot 10 + \frac{2}{36} \cdot 11 + \frac{1}{36} \cdot 12 = 7$$

Tatsächlich nennt sich diese Rechtfertigung in der Wahrscheinlichkeitstheorie das „Gesetz der großen Zahlen“. Der Beweis ist bei weitem nicht trivial. Dieses Gesetz aber auch nur zu erwähnen geht über jede normale Einsendung hinaus.

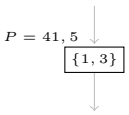
Die Frage nach der durchschnittlich erreichbaren Punktzahl kann man sich nun nicht nur „pro Spiel“ stellen, sondern auch aus jeder Spielsituation heraus. Betrachten wir zunächst eine mögliche Situation gegen Ende des Spiels: Der Spieler habe $\{1, 3\}$ auf der Hand. Welche Punktzahl P ist zu erwarten?

Schauen wir uns den Graphen dort an.

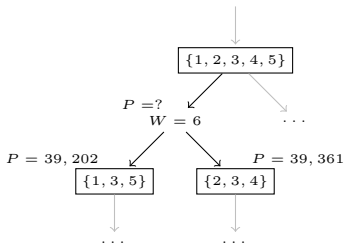


Die Karten $2, 4, 5, 6, 7, 8, 9$ hat der Spieler bereits umgedreht, also hat er 41 Punkte sicher. Würfelt er eine 3, so wird er als nächstes die 3 umdrehen, hat dann $41 + 3 = 44$ Punkte erreicht und das Spiel ist zuende, auch wenn die Karte 1 noch offen ist. Würfelt er 4, so wird er 1 und 3 umdrehen und hat $41+1+3 = 45$ Punkte erreicht. In allen anderen Fällen kann er nichts mehr tun. In der Situation $\{1, 3\}$ ist also $P = \frac{2}{36}\cdot 44+\frac{3}{36}\cdot 45+\frac{31}{36}\cdot 41 = 41,5$ *zu erwarten.* In der Mathematik nennt man dies auch den *Erwartungswert* und bezeichnet ihn mit $\mathbb{E}$.

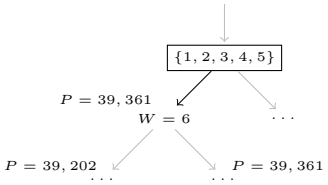
Unser berechnetes P können wir eintragen:



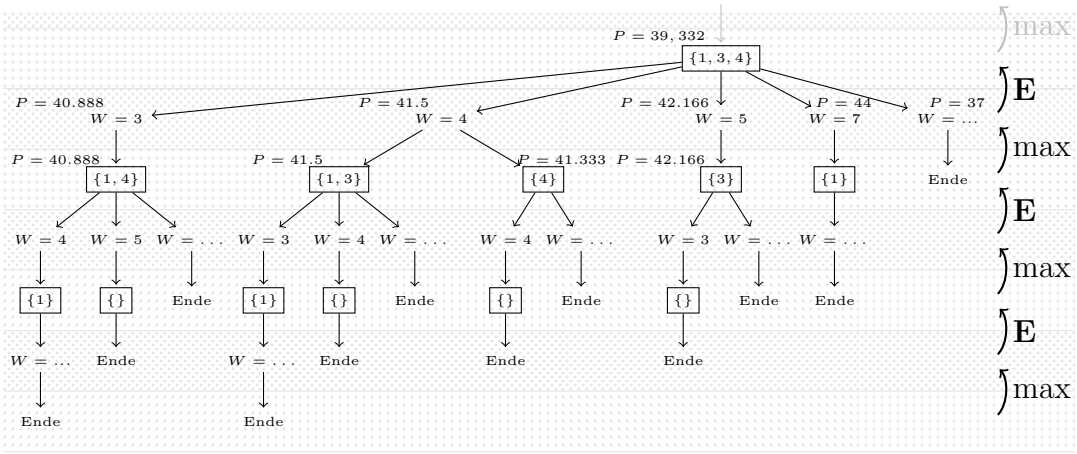
Nach der Situation $\{1, 3\}$ kam allerdings keinen Zug mehr, wo sich der Spieler zwischen mehreren Möglichkeiten entscheiden musste. Doch solche Situationen sind jetzt ganz einfach: Angenommen der Spieler hätte $\{1, 2, 3, 4, 5\}$ auf der Hand und $W = 6$ gewürfelt: Was soll er tun?



Natürlich wird er sich für den Zug entscheiden, der ihn in eine Situation bringt, aus der eine höhere Punktzahl zu erwarten ist: Er wählt das Maximum. Dies ist hier $P = 39,361$, also dreht er die Karten 1 und 5 um. Kopieren wir also $P = 39,361$ eine Ebene nach oben:



So können wir im Graphen von unten her (d.h. von den Kartenknoten aus mit weniger Karten) Schritt für Schritt die P-Werte berechnen: Um P in einer Kartenebene zu berechnen, bilden wir den Erwartungswert **E** der Kinder (Augensummenknoten): Das entspricht einer Auswertung von $P(H)$. Um P in einer Augensummenebene zu berechnen, nehmen wir das Maximum der Kinder (Kartenknoten): Das entspricht einer Auswertung von $P(H, W)$.



Dieses Verfahren ist genau der Minimax-Algorithmus, wenn man **E** durch min ersetzt: Statt des Würfels, der zufällig einen Zug auswählt, liegt bei ihm ein Gegner vor, der versucht, die Punktzahl des Spielers zu *min*imieren.
Somit ist klar:

- Nach und nach ist P für alle Knoten im Graphen so berechenbar. Der Algorithmus **terminiert** also.
- In jeder Situation, die der Spieler entscheiden kann, wählt er den optimalen Zug. (Der Teilbaum muss dazu natürlich bereits komplett berechnet sein.) Der Algorithmus ist also **korrekt**.
- Die **Laufzeit** ist pro Knoten $\mathcal{O}(1)$, die Gesamtlaufzeit also linear in der Anzahl der Knoten.

  Zur Berechnung des Maximums bzw. Erwartungswerts werden jeweils maximal 5 Vergleiche und 5 Zuweisungen bzw. 11 Additionen, 11 Multiplikationen und 1 Zuweisung durchgeführt. Wie man technisch die Kinder eines Knoten bestimmt, ist ein Implementationsdetail, das man sich jetzt zu überlegen hat. Wenn man das wie in der Programmdokumentation beschrieben getan hat, kann man die Laufzeitanalyse abschließen: Zur Bestimmung der Kinder eines Augensummenknotens braucht man, grob geschätzt, nochmals maximal 20 Vergleiche, 31 Additionen, 13 Shift-, 14 UND- und 12 ODER-Operationen. Somit sind wir bei unter 200 Operationen pro Knoten. Da alle Speicherzugriffe in lineare

Arrays gehen (P und die Wahrscheinlichkeiten aus Tabelle 1), sind auch diese schnell. Insgesamt sind wir also bei unter 1126400 Operationen, dürfen also auf eine sehr schnelle Laufzeit hoffen zur Berechnung des kompletten Baumes hoffen.

Am obersten Kartenknoten erhalten wir mit $P = 33,036$ die Antwort auf die Frage nach der durschnittlich erreichbaren Punktzahl bei optimalem Spielerverhalten.

Was ist mit „Strategien“?

Zum Beispiel könnte man versuchen, immer wenn der Spieler eine Wahl hat, zwei Karten möglichst aus der Mitte zu nehmen, etwa bei $\{1, 2, 3, 4, 5\}$ und $W = 6$ die 2 und 3 umzudrehen. (Sich so die 1 für später aufzuheben, da sie oft hilfreich ist.)

Hierbei handelt es sich zunächst mal lediglich um irgendeine Spielstrategie, von der zwar vermutet wird, dass sie gut spielt, aber deren Optimalität nicht bewiesen ist — also eine Heuristik.

Im Falle dieser Aufgabe ist eine Heuristik jedoch fehl am Platz: Ihre Optimalität könnte zwar schnell durch ein Programm bewiesen oder widerlegt werden (da der Graph eben recht klein ist), aber dann hat man das Problem ja bereits optimal gelöst, und braucht keine Heuristik mehr.

Natürlich wäre es nett, wenn obige Strategie optimal wäre: Dann könnte man auch als Mensch schnell den jeweils optimalen Zug bestimmen. Das ist aber nicht in der Aufgabe gefragt. Es handelt sich um den Bundeswettbewerb *Informatik*. Das obige Programm, das das Problem auf einem Rechner optimal löst, reicht völlig als Lösung der Aufgabe aus.

Auch der Versuch, alle 512 Kartenknoten als Mensch auf einem Papier oder mit Hilfe eines Kartenspiels auszuprobieren, ist unnötig — auch wenn das Betrachten einzelner Fälle natürlich hilft, die oben beschriebene Methode zu entwickeln.

Wäre der Strukturgraph des Spiels allerdings zu groß gewesen, so dass man ihn nicht im Rechner schnell komplett absuchen können würde, so wäre eine Heuristik (oder der Vergleich von mehreren) natürlich genau der richtige Lösungsansatz gewesen. Mit ein bisschen Erfahrung kann man die Frage „Heuristik oder nicht?“ meistens extrem schnell beantworten.

Heuristiken

Obwohl Heuristiken, wie gerade erwähnt wurde, eigentlich fehl am Platz sind, so wollen wir hier doch auf ein paar eingehen und die Ergebnisse vergleichen.
„Wenn man 2 Karten wählt, wählt man eine davon immer maximal.“ (keine vollständige Strategie) versagt z.B. für $H = \{1, \ldots, 7\}$ und $W = 11$: Es wählt $\{4, 7\}$ als Zug (Erwartungswert in dieser Situation 37,6995), aber $\{5, 6\}$ (Erwartungswert in dieser Situation 38,3857) wäre optimal gewesen.
Eine Reihe von Strategien versagt für $H = \{1, 2, 3, 4\}$ und $W = 4$: Sie wählen $\{4\}$ als Zug (Erwartungswert in dieser Situation 40,142), aber $\{1, 3\}$ (Erwartungswert in dieser Situation 40,2361) wäre optimal gewesen. Zu diesen Stategien zählen:

- „Man zieht so, daß im nächsten Zug noch möglichst viele der Augensummen mit offenen Karten darstellbar sind[8].“ hat einen Erwartungswert von 32,7925.
- „Man zieht so, daß im nächsten Zug noch möglichst wahrscheinliche Augensummen (siehe Tabelle 1) mit offenen Karten darstellbar sind[9].“ hat einen Erwartungswert von 32,8734.
- „Wenn W gewürfelt wurde und Karte W offen ist, wähle sie. Sonst wähle das Paar Karten, dessen Differenz maximal ist.“ hat einen Erwartungswert von 32,9157.

Unsere (optimale) hat einen Erwartungswert von 33,0361.

[8] Bei mehreren Möglichkeiten wird vorzugsweise eine einzelne Karte verwendet, oder bei Paaren das mit der größeren Differenz.

[9] Ebenso.

Mit Randomisierung kann natürlich das Ergebnis zufällig **nach oben abweichen**. Dies sollte sich bei häufiger Wiederholung des Experimentes und Durchschnittsbildung jedoch aufheben: Die Wahrscheinlichkeit, daß das beobachtete Ergebnis vom theoretischen Grenzwert nach N Wiederholungen um mehr als ϵ abweicht, liegt in $\mathcal{O}(\rho(\epsilon)^N)$, wobei $\rho(\epsilon) < 1$. (allgemeiner siehe auch: Gesetz der großen Zahlen)

Das ist doch eine Lösung auf Universitätsniveau

Niveau unserer Lösung

Das ist **teilweise** richtig: Beim Erstellen wurde viel Zeit auf möglichst knappe Sprache, klare Definitionen und richtige Begrifflichkeit verwendet. Alle vorkommenden mathematischen Begriffe werden in ihrer üblichen Definition verwendet.

Wir haben uns sehr bemüht, das Niveau[10] der Lösung möglichst gering zu halten, also keine unnötigen Konzepte zu verwenden (wie etwa „bedingter Erwartungswert“) — auch dann, wenn dies die Lösung erheblich verkürzt und somit ihr Verständnis für den Leser, der mit diesen Begriffen vertraut ist, sehr vereinfacht hätte. Eine echte Lösung auf Universitätsniveau hätte dies nicht getan: Von den Seiten 1–3 wären lediglich die Formeln und Definitionen übrig geblieben. Zusammen passen diese wohl etwa auf eine einzige Seite.

Auch wenn die im ersten Abschnitt genannten Eigenschaften (knappe Sprache, etc.) keine Voraussetzung für eine richtige Lösung sind, so ist doch anzumerken, dass sie jedem Leser beim ersten Lesen extrem beim Verständnis helfen. **Und so auch dem Bewerter!** Natürlich ist klar, dass jeder Teilnehmer nur Konzepte verwenden kann, die er kennt. Das Niveau einer Einsendung hängt somit zum einen sehr von der Klassenstufe des Teilnehmers ab, aber auch von seinem sonstigen Wissen. Wir hoffen, dass dieses Dokument ein paar neue Konzepte vermitteln kann und dem Leser Anreize gibt, sein Wissen in anderen Quellen zu vertiefen.

Ist diese Lösung nicht zu mathematisch?

Zunächst einmal ist die Aufgabe selbst sehr mathematisch. Auch wenn in der Aufgabenstellung keine Fachbegriffe verwendet werden, so greift unsere Lösung das natürlich auf. **Die Formeln sind nicht notwendig, um eine korrekte Lösung zu erstellen!** Sie und die Definitionen sind aber die Essenz dessen, was hinter einer jeden richtigen Lösung stecken muss: Um Ideen zu vermitteln, muss man sich erst auf die Bedeutung der verwendeten Begriffe einigen, sonst redet man aneinander vorbei. Der verwendete Formalismus hilft, sowohl dem Verfasser beim Schreiben als auch dem Leser beim Verstehen, sich dieser Problematik bewusst zu werden und sie zu lösen.

Es gibt viele Aufgaben beim BWINF, die weit weniger bis gar nicht mathematisch sind.

Ein weiterer source code als Lösung

Ohne Kommentare, und daher noch nicht wirklich BWINF-tauglich, aber schön kurz berechnet sie den oben erläuterten Graphen bottom-up (also von unten) statt top-down (von oben): anders als unsere Lösung oben. Das Programm ist mit

`java Main` *(binär codiertes H) (W)*

aufzurufen, also für $H = \{1, 3, 4\}$ und $W = 4$:

```
java Main 000001101 4
```

Das binär codierte H, also der Schlüssel der Knoten im Graphen, wird hier als Array-Index verwendet, womit man sich die `map` spart, die das Programm aus der Einsendung zu diesem Zweck braucht.

```
1  public class Main {
2    public static void main(String[] args) {
3      int bin[] = new int[13];
4      for (int i = 1; i <= 6; i++)
5        for (int j = 1; j <= 6; j++)
6          bin[i+j]++;
```

[10] Niveau ist nicht Qualität.

```
 7
 8       int MAX = 1 << 10;
 9
10       double avg_gain[] = new double[MAX];
11       int best_move[][] = new int[MAX][13];
12
13       for (int i = 2; i < MAX; i++) {
14         for (int j = 2; j <= 12; j++) {
15           double prob = bin[j] / 36.0;
16           double val = -1;
17           if (j <= 9 && (i & 1 << j - 1) != 0) {
18             double gain = avg_gain[i ^ 1 << j-1];
19             if (gain > val) {
20               val = gain;
21               best_move[i][j] = j;
22             }
23           }
24           for (int k = 1; k <= 9 && k < j; k++)
25             if (2 * k != j && j - k <= 9 && (i & 1 << k - 1) != 0
26                 && (i & 1 << j - k - 1) != 0) {
27               double gain = avg_gain[i ^ 1 << k - 1 ^ 1 << j - k - 1];
28               if (gain > val) {
29                 val = gain;
30                 best_move[i][j] = k;
31               }
32             }
33           avg_gain[i] += val == -1 ? 0 : prob * (val + j);
34         }
35       }
36
37       System.out.println("average gain: " + avg_gain[MAX-1]);
38
39       if (args.length > 1) {
40         int state = Integer.parseInt(args[0], 2),
41             dice = Integer.parseInt(args[1]);
42         System.out.println("best move for you: turn card " +
43             best_move[state][dice] +
44             (best_move[state][dice] != dice ? " and " +
45              (dice - best_move[state][dice]) : ""));
46       }
47     }
48   }
```

7 Generelle Kommentare

Seitenkopf

Der Seitenkopf sollte die Verwaltungsnummer (z.B. 29.123.01), eventuell den Namen des Autors, sowie den Aufgabenname oder die Aufgabennummer enthalten. In der ersten Runde kann der letzte Teil „.01“ weggelassen werden, da damit die einzelne Person in einer Gruppe identifiziert wird. Die Verwaltungsnummer bekommt man entweder vor Einsendeschluss der ersten Runde über die Homepage `http://www.bwinf.de` oder automatisch danach zugeordnet. Bei Gruppeneinsendungen, also in der ersten Runde, genügt die Verwaltungsnummer eines Gruppenmitglieds.

Seitenfuß

Im Seitenfuß sollte die Seitennummer stehen. Bei mehreren Aufgaben sollte man sie am besten durchgehend nummerieren. Gegebenenfalls kann man Blätter ja auch mehrfach durch den Drucker ziehen, um nachträglich Seitenkopf und -fuß aufzudrucken, wenn man seine Software nicht ordentlich konfiguriert bekommt.

kein Abschnitt „Allgemeines“

Der Inhalt einer allgemeinen Einführung ist in den allermeisten Fällen von keiner Relevanz oder gehört nicht an diese Stelle. Relevanten Inhalt sollte man an die richtige Stelle in der Lösung bringen, unrelevanten Inhalt sollte man weglassen.

Aufgabenstellung nicht abschreiben

Es ist nicht notwendig, die Aufgabenstellung abzuschreiben: die Bewerter kennen die Aufgabe und haben zahlreiche Ausdrucke zur Verfügung. Gleichzeitig gilt aber auch:

offene Punkte in der Aufgabenstellung

Die eigene Interpretation der Aufgabenstellung muss deutlich dokumentiert werden! Besonders in der zweiten Runde kann es vorkommen, dass die Aufgabenstellungen nicht eindeutig sind und von den Teilnehmern unter anderem eine sinnvolle Interpretation erwartet wird. Wenn das der Fall ist, muss die genaue Interpretation klar und deutlich

dokumentiert werden. Es gilt: offene Fragen sollten so beantwortet werden, dass sie die Aufgabe nicht trivial werden lassen, aber auch nicht unnötig verkomplizieren.

kurze Sätze

Am besten verwendet man kurze, einfache Sätze. Die Dokumentation soll für einen Bewerter leicht zu lesen sein. Das kann man z.B. dadurch erreichen, dass man keine langen, verschachtelten Sätze verwendet und indem man Zusammenhänge nicht komplizierter darstellt als sie sind.

Begriffe definieren und konsequent verwenden

Die Sachverhalte, die die Lösung behandelt, also zum Beispiel Algorithmen, Objekte oder Gegenstände aus der Aufgabenstellung, sollten klar definiert werden, sofern sie nicht selbstverständlich sind. Für jeden Sachverhalt sollte genau ein Begriff verwendet werden — und auch nur dieser sollte dann konsequent verwendet werden, um diesen Sachverhalt zu bezeichnen. Die Wiederholung desselben Begriffes mag zwar langweilig erscheinen, hilft aber enorm beim Verständnis. (Bei der Einsendung handelt es sich nicht um einen Deutschaufsatz, sondern um einen wissenschaftlichen Text.)

korrekte Rechtschreibung und Grammatik

Man sollte Schreib- oder Grammatikfehler vermeiden, soweit es geht. (Das sollte sich eigentlich von selbst verstehen.) Jahrelange Erfahrung eines Autors mit diesem Punkt ist, dass man am besten die komplette Einsendung vor dem Einsenden in gedruckter Fassung einmal auf Rechtschreibung und einmal auf Grammatik durchliest, oder Bekannte darum bittet, dies zu tun.

übersichtliches Layout

Die Gliederung sollte übersichtlich sein. Zum Beispiel sollten Überschriften einheitlich formatiert sein und die Struktur der Dokumentation betonen. LaTeX eignet sich unserer Meinung und Erfahrung nach hervorragend um Lösungen zu erstellen, weil allen Bewertern das Schriftbild bereits bekannt ist und in LaTeX gesetzte Texte sich durch die automatische Formatierung sehr angenehm lesen lassen. Es ist natürlich nicht notwendig, dass alle Diagramme in LaTeX erstellt werden. In Microsoft Word und OpenOffice empfiehlt es sich, sich weitgehend an die vordefinierten Schriftstile zu halten.

Das Aufgabenblatt komplett lesen

Wirklich genau von vorne bis hinten sollte man das Aufgabenblatt mindestens zweimal lesen: Ganz am Anfang, wenn man es bekommen hat, und bevor man die Einsendung ausdruckt, um sie abzuschicken. Es enthält viele wichtige Hinweise und Regeln, die leicht überlesen oder wieder vergessen werden.

Länge der Lösung

Die Dokumentation sollte möglichst kurz sein, aber natürlich alles Relevante enthalten. Es lohnt sich sehr, Arbeit in die Verkürzung oder Verbesserung des Textes zu stecken. Einen Richtwert für die Länge anzugeben ist schwierig, weil die Informationsdichte je nach Autor extrem schwankt. Diese sollte möglichst hoch sein, solange der Text noch verständlich ist. Sicherlich sollte die Dokumentation ohne Quelltext für eine Erstrundenaufgabe nicht länger als 20 Seiten sein, für die zweite Runde sicher nicht über 40 Seiten. Dies sollen *Obergrenzen* sein, weniger ist besser! Unsere Lösung ist mit 6 Seiten lang, da sie versucht, möglichst wenig Begriffe vorauszusetzen. Wie oben erwähnt, wäre man sonst wohl mit 4 Seiten ausgekommen.

Zeit für Programmieren vs. Dokumentieren

Die Dokumentation zu erstellen braucht mindestens so viel Zeit, wie das Programmieren gedauert hat, wenn nicht ein Vielfaches davon. Ist man vorher fertig, macht man ziemlich sicher etwas falsch.

Ist der erste Entwurf der Dokumentation fertig, bekommt man häufig eine Idee, wie man die Lösung noch besser darstellen könnte. Es lohnt sich, das dann zu tun, also den ersten Entwurf dann weitgehend zu überarbeiten. Dafür sollte man Zeit einplanen.

Es empfiehlt sich, so bald wie möglich nach dem Erhalt der Aufgaben mit dem Bearbeiten, also dem Nachdenken, Programmieren und Dokumentieren, anzufangen. Die Erfahrung zeigt, daß Torschlusspanik zwar eine Motivation zur Arbeit ist — aber leider in der Regel auch ein Garant für schlechte Dokumentation.

8 Gliederung

(kein) Inhaltsverzeichnis

Ein Inhaltsverzeichnis ist eigentlich nicht notwendig, vor allem, wenn man sich an die

Standardstruktur hält. Die Standardstruktur und deren Inhalte werden am besten durch unsere Einsendung am Beginn dieses Dokumentes vermittelt. Wir listen hier zunächst die Teile auf, bevor wir unten genauer auf sie eingehen:

1. **Lösungsidee**: Hier sollte eben die Idee der Lösung beschrieben werden. Dazu gehört meistens die Abstraktion des Problems, die Idee selbst und die Argumentation, wieso die Idee richtig ist.

2. **Programm-Dokumentation**: Nachdem unter „Lösungsidee“ das Problem theoretisch gelöst wurde, muss die Lösung nun durch ein Programm realisiert werden. Diese Transformation von Idee ins Programm soll hier vermittelt werden. Beim Verständnis kann Pseudocode oder zum Teil auch ein UML-Diagramm enorm helfen.

3. **Programm-Ablaufprotokoll**: Hier soll dokumentiert werden, wie das Programm sich unter verschiedenen Eingaben verhält. Die Eingaben sind am besten so gewählt, sodass der Leser nach diesem Abschnitt überzeugt ist, dass das Programm korrekt funktioniert und die relevanten Fälle lösen kann (z.B. für $N = 1000000$).

4. **Quelltext**: Die *relevanten* Teile des Programms.

Nicht bei allen Aufgaben ist es möglich, dieses Schema anzuwenden, um eine optimale Lösungsstruktur zu erreichen.

Bezeichner erst in der Programm-Dokumentation

Variablen- oder Funktionsnamen haben **nichts** in der Lösungsidee zu suchen! Die Lösungsidee soll **abstrakt** die Lösung beschreiben. Das Programm ist lediglich **eine mögliche** Umsetzung der Idee.

Dies zwingt zum einen dazu, dass die Idee eigenständig formuliert sein muss, sich also nicht hinter Aussagen wie „XY wird dann in der Funktion `xy()` berechnet“ verstecken kann. Das hilft beim Verständnis, denn normalerweise wird man die Lösung von vorne nach hinten lesen, und wie `xy()` funktioniert, ist zu dem Zeitpunkt, an dem dieser Satz gelesen wird, weder bekannt noch relevant. Zum anderen ist der Name der Funktion (`xy()`) nur von sehr geringem Interesse: Ist die Lösungsidee richtig, und arbeitet das Programm korrekt (das sind Dinge die den Bewerter interessieren), so ist wohl auch `xy()` (oder wie auch immer die Funktion heißen mag) korrekt.

9 Lösungsidee

Problem formalisieren, Mehrdeutigkeiten entscheiden

Der erste Teil der Lösungsidee sollte üblicherweise darin bestehen, das gegebene Problem zu formalisieren. Dazu gehört auch, dass eventuelle Uneindeutigkeiten in der Aufgabenstellung entschieden und klar dokumentiert werden. Das Ergebnis sollte eine formale, mathematische Darstellung des Problems sein – so einfach wie möglich und nicht unnötig kompliziert. Es geht nicht darum, irgendjemanden davon zu überzeugen, dass das Problem schwierig ist. Vielmehr glänzen gute Lösungen dadurch, dass sie das Problem erstaunlich einfach darstellen. Gute, veranschaulichende Beispiele sind natürlich erlaubt und erwünscht.

Bei Unklarheiten oder Fragen, sollte man als allererstes seinen gesunden Menschenverstand anwenden. Mit dem „BWINF-Dreisprung: Überlegen, Entscheiden, Dokumentieren“ kann man nie etwas falsch machen.

Durch die Entscheidung sollte die Aufgabe natürlich weder trivial noch unnötig kompliziert werden. Eventuell lohnt sich ein Blick in die Sektion „Häufig gestellte Fragen“ (englisch: FAQ) auf `http://www.bwinf.de`.

Definitionen

Unsere Lösung beginnt zum Beispiel damit, dass sie die Begriffe *Karte*, *Kartenmenge*, und die Funktion $P(H)$ definiert:

Eine *Karte* wird durch eine natürliche Zahl von 1 bis 9 dargestellt. Eine *Kartenmenge* ist eine Menge von Karten. Ein Spielzustand wird eindeutig durch die Kartenmenge der noch nicht umgedrehten Karten beschrieben. Diese Karten bezeichnen wir als die *offenen Karten*. Wir schreiben $P(H)$ für die erwartete Punktzahl einer Kartenmenge H, wenn genau die Karten in H offen sind und der Spieler „optimal“ spielt.

Zur Veranschaulichung geben wir ein Beispiel:

Zum Beispiel ist $P(\{2\}) = 43 + 2 \cdot \frac{1}{36}$, da dem Spieler bereits $43 = 1 + 3 + 4 + 5 + 6 + 7 + 8 + 9$ Punkte sicher sind und er zwei weitere Punkte genau dann erhält, wenn als nächstes eine Zwei gewürfelt wird ($= 2 \times \boxdot$), was mit einer Wahrscheinlichkeit von $\frac{1}{36}$ geschieht, er also die verbleibende Karte 2 umdrehen kann.

Anschließend wird der Begriff *optimal spielen* definiert, da er aus der Aufgabenstellung nicht klar hervorgeht. Die gewählte Definition ist allerdings die einzig sinnvolle[11]:

Optimal zu spielen bedeutet, immer genau die Karten herumzudrehen, die die danach zu erwartetende Punktzahl maximieren (unter der Annahme, dass man weiterhin optimal spielt).

Schließlich drücken wir die Fragestellung in unserem neu eingeführten Formalismus aus:

Mit diesen Definitionen kann die **Aufgabenstellung** wie folgt umformuliert werden:

1. Gegeben H und W, für welches $K \in Z(H, W)$ ist $P(H, W) = P(H \setminus K)$?
2. Was ist der Wert von $P(\{1, \ldots, 9\})$?

Algorithmus und seine Korrektheit

Als nächstes müssen die Algorithmen beschrieben werden, die das Problem lösen. Dazu gehört auch immer ein Korrektheitsargument (oder besser noch: ein Beweis). Insbesondere in der zweiten Runde kommt es gelegentlich vor, dass keine effiziente oder vollständig korrekte Lösung für das gegebene Problem bekannt ist. In einem solchen Fall ist es nötig, die Schwachstellen oder eingegangenen Kompromisse der vorliegenden Lösung klar zu dokumentieren. Eine undokumentierte Schwachstelle bedeutet oftmals Punktabzug.

Es ist einerseits wichtig, die Algorithmen in Worten so zu beschreiben, dass der Leser die richtige Intuition und Vorstellung bekommt. Andererseits sollten die Beschreibungen so präzise sein, dass der Leser den Algorithmus ohne viel nachzudenken selbst implementieren kann. Für eine präzise Beschreibung eignet sich Pseudocode besonders gut. Triviale Schritte im Algorithmus müssen natürlich nicht als Pseudocode ausgeführt werden. Wie immer gilt: weniger ist mehr. Echter Quelltext oder Implementationsdetails haben in der Lösungsidee nichts zu suchen.

Es kann interessant sein, alternative Lösungsverfahren zu kurz zu beschreiben und zu erklären, weshalb diese nicht verwendet worden sind. Wenn dies von der Aufgabenstellung impliziert ist, kann es auch nötig sein, mehrere Lösungsverfahren zu implementieren und diese zu vergleichen. In einem solchen Fall ist es sehr wichtig, die Vergleichsergebnisse *übersichtlich* im Kapitel „Programm-Ablaufprotokolle“ zu präsentieren (zum Beispiel in einer Tabelle) und zu interpretieren. Wenn die vergleichsrelevanten Ergebnisse auf mehreren Seiten verteilt sind, wird sich kaum ein Bewerter die Mühe machen und diese zusammensuchen.

[11] Man könnte auch versuchen für den Worst-Case (`http://de.wikipedia.org/wiki/Worst_case#Informatik`) zu optimieren. Nach genauerer Betrachtung ist dies jedoch uninteressant.

9.1 Theoretische Analyse

Laufzeitkomplexität

Nachdem die Lösungsidee beschrieben wurde, muss in den allermeisten Fällen eine theoretische Analyse folgen. Oft sollte hier die Laufzeitkomplexität des Algorithmus genannt und begründet werden.[12] Auch die Speicherkomplexität ist interessant, falls diese nicht trivial ist. Wenn nachgewiesen werden kann, dass das vorliegende Problem NP-vollständig ist, ist dies meistens Extrapunkte wert. Die Analyse sollte knapp und richtig sein: es ist nicht hilfreich, wenn die Anzahl der Zyklen auf einer selbst-definierten Rechnerarchitektur auf zehn Seiten genau berechnet wird. Vielmehr sollten eine oder mehrere Variablen definiert werden, die die Größe der Eingabe beschreiben. Mithilfe dieser Variablen kann die Laufzeit- bzw. Speicherkomplexität dann in $\mathcal{O}$-Notation[13] angegeben werden.

Wenn man von der Laufzeit einer „durchschnittlichen Eingabe“ sprechen möchte (dieser Begriff muss natürlich definiert sein) und die Analyse dieser Laufzeit zu kompliziert ist, kann man mithilfe von Zeitmessungen eine Hypothese aufstellen (diese sollte selbstverständlich auch als reine Hypothese dargestellt werden). In der ersten Runde sind solche Messungen selten sinnvoll.

Nicht nur in der theoretischen Analyse ist es erlaubt und sinnvoll, mathematische Symbole wie $\sum$ oder $\prod$ zu verwenden, wenn ein Zusammenhang nicht einfacher dargestellt werden kann – Ausdrücke mit diesen Symbolen sind meistens leichter zu verstehen als andere Konstruktionen oder umständliche Sätze, die den Zusammenhang beschreiben. Oft können (und sollten) kompliziertere Ausdrücke mit einem einfachen Beispiel erläutert werden. Zum Beispiel führen wir die Laufzeitanalyse zunächst anhand der fest vorgegebenen Eingabegrößen vor, bevor wir sie auf andere Eingaben verallgemeinern:

> Schließlich lässt sich noch beobachten, dass es für die Funktion P nur $512 = 2^9$ verschiedene Eingabewerte gibt (alle möglichen Teilmengen von $\{1, \ldots, 9\}$). Der Rückgabewert kann nach der ersten Berechnung also gespeichert werden, was die Laufzeit der Berechnungen erheblich verbessert. Auf diese Art und Weise wird $P(H)$ nur höchstens 512 Mal berechnet. Dementsprechend wird $P(H, k)$ höchstens $11 \cdot 512 = 5632$ Mal berechnet.

10 Programm-Dokumentation

Bezug zwischen Lösungsidee und Implementation

In der Programm-Dokumentation geht es darum, den Bezug zwischen der Lösungsidee und der Implementation herzustellen. Nur selten sollte hier der tatsächliche Quelltext abgedruckt sein. Auf gar keinen Fall soll der Code Zeile für Zeile erklärt werden. Wenn der Quelltext nicht selbst-erklärend ist (und damit ist nicht eine Flut von Kommentaren gemeint!), ist er oft in schlechtem Zustand. In den allermeisten Fällen reicht es anzugeben, in welchen Methoden die Algorithmen aus der Lösungsidee implementiert sind.

Dies ist der richtige Abschnitt, um darauf einzugehen, wie bestimmte Konzepte der Lösungsidee elegant im Programm modelliert wurden (in diesem Fall dürfen natürlich auch kurze Ausschnitte des Quelltextes gezeigt werden).

In unserer Lösung erklären wir zum Beispiel, wie wir Kartenmengen im Programm darstellen:

[12] Auch wenn dies in der ersten Wettbewerbsrunde keine Voraussetzung ist, ist es dennoch gut, das zu tun.

[13] `http://de.wikipedia.org/wiki/Landau-Symbole`

> Zum Speichern von Kartenmengen haben wir eine Datenstruktur entwickelt, die nur mit bis zu 32 Karten funktioniert, dafür aber sehr effizient ist. Dazu wird eine Kartenmenge als Bitvektor dargestellt. Die Datenstruktur ist mit der Klasse Cards abstrahiert. Intern wird ein **unsigned int** verwendet, in dem das i-te Bit genau dann gesetzt ist, wenn die Karte mit dem Wert i in der Kartenmenge vorhanden ist.

Danach erklären wir kurz, wo die beiden Funktionen aus der Lösungsidee zu finden sind:

> Die Funktion $P(H)$ aus der Lösungsidee ist als ExpectedScore implementiert. Für das Zwischenspeichern der Rückgabewerte wird eine std::map verwendet (das Abspeichern und Auslesen kostet hier also $\mathcal{O}(2^n) = \mathcal{O}(n)$ statt $\mathcal{O}(1)$ Zeit).

Wenn euer Programm (ohne Benutzeroberfläche) länger als wenige hundert Zeilen ist, kann es für den Bewerter hilfreich sein, ein UML Klassendiagramm[14] oder ein Flussdiagramm[15] zu sehen. Solche Diagramme haben allerdings nur ihren Platz in der Dokumentation, wenn sie zum Verständnis beitragen und sich auf relevanten Code beziehen. Es wird kein Klassendiagramm benötigt, was zeigt, welches die privaten Felder eurer C++ Klasse sind (Das können die Bewerter auch im Quelltext nachlesen.), und auch keines, das darstellt, dass euer Hauptfenster 5 Buttons enthält.

kein Benutzerhandbuch schreiben

Keinesfalls sollte der Name „Programm-Dokumentation“ als Aufforderung gesehen werden, ein Benutzerhandbuch zu schreiben. Es sollte davon ausgegangen werden, dass mögliche Leser bzw. Bediener sowohl mit dem Programmieren als auch mit der konkreten Aufgabenstellung vertraut sind. Ist darüberhinaus eine Benutzerdokumentation notwendig, so ist die Benutzeroberfläche offenbar verbesserbar.

11 Programm-Ablaufprotokoll

3–5 Beispiele

Zu jeder Lösung gehören 3 bis 5 Ablaufprotokolle. Auch wenn 3 Beispieleingaben in der Aufgabe vorgegeben sind, sollte man zusätzlich noch 3 Eingaben selbst entwerfen, dann erklären, wieso man was wie gewählt hat, und das Ergebnis kommentieren. Mit der Wahl der Eingaben sollte man möglichst viele Fälle abdecken (solche, die etwa im alltäglichen Betrieb zu erwarten sind, aber auch solche, die Randwerte oder Sonderfälle darstellen, und das Programm vielleicht an seine Grenzen bringen).

Schwächen kommentieren

Schwächen des Programms sollte man kommentieren. (Das wird positiv gewertet, das Weglassen solcher Kommentare negativ!). Bringt das Programm etwa nur bei jedem zehnten Aufruf ein gutes Ergebnis, so sollte man dokumentieren, dass man das Programm für die Beispiele zehnmal hat laufen lassen, oder eben gleich das Programm ändern, so dass es den Algorithmus zehnmal ausführt (das gehört dann aber natürlich schon entsprechend in der Lösungsidee dokumentiert).

Protokolle ggf. kürzen

Die „Ablaufprotokolle“ sollen *sinnvoll* darstellen, was bei einem Programmablauf passiert. Handelt es sich bei der Aufgabe etwa um ein Computerspiel, bei dem 1000 Züge gemacht werden, so ist eine Liste der Züge sicherlich nicht sinnvoll. (Die Einsendung wird von Menschen gelesen, nicht von Computern!) Stattdessen sollten vielleicht einzelne Stellungen mit jeweils nächsten paar Zügen ausgegeben werden, anhand derer dann dokumentiert wird, wie sich die implementierte Methode stategisch auswirkt.

In unserer Einsendung haben wir lediglich die Interaktion auf der Konsole ausgegeben. Für die erste Runde ist dies meistens ausreichend (die Art und Weise, wie P für jeden Augensummen- bzw. Kartenknoten berechnet wurde, ist ja für alle solche Knoten jeweils gleich), in der zweiten Runde des Wettbewerbs sollte das Programm sicherlich

[14] `http://en.wikipedia.org/wiki/Unified_Modeling_Language`
[15] `http://en.wikipedia.org/wiki/Flowchart`

etwas mitteilsamer sein. (Zu) lange Ausgaben sollte man gegebenenfalls manuell sinnvoll kürzen.

Keinen schwarzen Hintergrund

Beim Drucken von Ablaufprotokollen kann man umweltfreundlich sein und Geld sparen, wenn man Screenshots von schwarzen Konsolenfenstern invertiert. Das sollte man daher tun, es ändert nichts an der Sache.

12 Programm-Quellcode

Relevant vs. unrelevant

Der relevante Teil des Programm-Quellcodes muss in ausgedruckter Form vorliegen. (Auf den Datenträger gehen wir später ein.) Die Unterscheidung zwischen relevantem und unrelevantem Code entscheidet sich an der Frage, was in der Aufgabenstellung gefragt ist. In unserem Beispiel war die Eingabe der Spielsituation, die wir textbasiert über die Konsole realisiert haben, nicht relevant. Daher haben wir ihren Code in separate Dateien ausgelagert und diese nicht abgedruckt. Diese Trennung unterscheidet aber nicht nur, was abgedruckt wird, sondern ist auch gutes Softwaredesign: Wir haben ein Interface (zu deutsch: Schnittstelle) entwickelt, die den Lösungsalgorithmus von seiner Verwendung isoliert. Das Interface selber, in drehzahl_strategy.h definiert,

Modular programmieren

```
1 #include "cards.h"
2
3 // Computes the expected final score for a given hand.
4 double ExpectedScore(const Cards& hand);
5
6 // Computes what cards to play next, given a hand and the sum of rolled dice.
7 // Returns an empty set of cards if no move is possible.
8 Cards BestNextMove(const Cards& hand, int roll);
```

haben wir nicht abgedruckt, weil es schon in der Programm-Dokumentation beschrieben und einfach aus der bereits abgedruckten drehzahl_strategy.cpp ableitbar war. Solche Interfaces machen das gesamte Programm testbarer, einfacher und wartbarer, und helfen sowohl beim Lösen des Problems, da man ein großes Problem in zwei kleine unterteilt, die man unabhängig voneinander lösen kann, als auch später beim Kommunizieren bzw. Dokumentieren der eigenen Lösung, da alle so isolierten Komponenten unabhängig voneinander sind — sofern die Interfaces exakt spezifiziert sind[16].

Sind die Ein- und Ausgabe, die graphische Oberfläche (zum Beispiel Fensterklassen), Ereignisbehandlungsmethoden, das Lesen und Schreiben von Dateien, oder ähnliche Komponenten des Programms nicht ein zentraler Teil der Aufgabenstellung, so wäre es auch falsch, sie zu einem zentralen Teil der Lösung zu machen: Dieser Code ist oft recht technisch und an das verwendete System gekoppelt, zum Beispiel an Windowing Toolkits oder Runtime Libraries. Tragen diese Teile des Codes nichts zur Lösung des eigentlichen Problemes bei, so sind sie nicht relevant.

Aus dieser Erkenntnis folgt, dass solche Teile des Codes auch keinen Wert für die Lösung des Problems tragen. Man muss sie somit nicht nur nicht abdrucken, sondern eigentlich auch gar nicht schreiben: Mit der Konsoleeingabe haben wir in unserem Fall das Problem der Eingabe möglichst einfach gelöst: Eine graphische Oberfläche (englisch: graphical user interface, GUI) hätte keinen Mehrwert geschaffen, sondern lediglich den nicht-relevanten Codeteil verlängert.

Konzentration auf das Wesentliche

Schreibt man doch GUI-Code, so sollte man ihn, soweit es geht, vom Rest des Programms isolieren. Dazu bietet sich das „Model-View-Controller-Pattern“ an[17]: Es darf als bekannt vorausgesetzt werden, und die konsistente und konsequente Verwendung der Begriffe „Model“, „View“ und „Controller“ im Code und in der Dokumentation helfen nach der Erwähnung des Patterns wiederum enorm beim Verständnis.

MVC

[16] http://de.wikipedia.org/wiki/Schnittstelle_%28Programmierung%29#Technische_Details

[17] http://de.wikipedia.org/wiki/Model_View_Controller ist leider nur von begrenzter Qualität, [5] ist die Lektüre der Wahl.

Natürlich muss man an diesen Stellen ein bisschen mitdenken: Unrelevante Programmteile möglichst einfach zu realisieren heißt nicht, dass das Programm deswegen unbedienbar werden sollte: Zum Einlesen von 100 Zahlen pro Zahl jeweils ein Dialogfenster zu zeigen, ist sicher schlechter Stil, für den es Abzug gibt — auch wenn das die Länge des unrelevanten Codes vielleicht minimiert hätte.

monospace

Abgedruckter Code sollte in *monospaced font* gedruckt werden, also einer Schriftart, die jedem Zeichen die gleiche horizontale Breite beimisst. Beispiele für solche Schriftarten sind `tt` in TEX und `Courier New` in Windows.

Zeilenumbrüche

Automatische Zeilenumbrücke, also Zeilen, die länger als die Seitenbreite sind, sind im Code zu vermeiden. Kleiner als 7pt sollte der Druck auf keinen Fall werden. Ist eine Zeile doch mal zu lang — und bei gutem Code ist das äußerst selten der Fall — so bieten heutige Entwicklungsumgebungen meistens die Funktion an, die Anzahl der Zeichen pro Zeile zu begrenzen, und auch, den Code automatisch entsprechend umzubrechen. Entweder manuell oder automatisch sollte man also alle Zeilen an logischen Stellen brechen, wo es nötig ist. Das ist nicht nur guter Stil, sondern auch hilfreich für den Leser.

Gute Kommentare

Guter Code ist gut kommentiert. Das bedeutet, dass die Kommentare den Code perfekt *ergänzen*, und so die Kombination aus Code und Kommentaren alle Fragen beantworten, die jemand haben könnte, der die Datei liest.[18]

```
i++; // i wird um 1 erhoeht
```

In diesem üblichen schlechten Beispiel ergänzt der Kommentar den Code nicht. Es ist davon auszugehen, dass der Leser die Programmiersprache versteht.

```
71  result += Probability(roll) * current_score;  // Game over.
```

Dieser Kommentar ist ist im Kontext unserer Lösung exzellent: Er beschreibt, dass mit diesem Code-Pfad das Ende des Spiels erreicht wurde — also genau das, was wir im Spielverlaufsgraphen oben mit „Ende“ bezeichnet hatten.

Style Guides

Schließlich ist es noch sehr hilfreich, wenn der Quelltext in einem konsistenten Stil geschrieben ist. Als Grundlage für C++ Code kann man z.B. [16] und als Grundlage für Java Code [9] verwenden.

12.1 Datenträger

Der *gesamte* Programm-Quellcode muss auf dem Datenträger der Einsendung beiliegen.

Informationen, die zur Inbetriebnahme des Programms notwendig sind, gehören erwähnt (bzw. Dateien auf den Datenträger), und sollten so kurz wie möglich ausfallen. Es kann von einem üblichen Windows-, Linux- oder Mac-Systemen ausgegangen werden. dass heißt, dass man alle exotischen Bibliotheken beilegen sollte oder das Programm statisch linkt. Ein vom Datenträger direkt auf einem dieser Systeme ausführbares Executable ist perfekt.

In unserem Fall haben wir standardisiertes C++ und die C++ Standard Library[19] verwendet. Das Programm ist somit einfach unter jedem der drei erwähnten Betriebsysteme zu übersetzen: Wir konnten uns jegliche Information sparen.

Eine kurze Information im Programm-Ablaufprotokoll wie „das Programm wurde unter Perl 5.12.2 getestet,“ ist bei Skriptsprachen gut, oder auch natürlich dann notwendig, wenn nicht-standardisierte Funktionen verwendet werden.

Beispieldaten beifügen

Am besten werden die für die Beispiele verwendeten bzw. die von ihnen generierten Daten ebenfalls auf dem Datenträger beigelegt.

[18] Ein gutes Beispiel dafür, was sich Leser alles fragen können und was gute Dokumentation ist, sind die Java Collections, zum Beispiel `http://download-llnw.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html`. Als eine Sammlung von Klassen, die viele Algorithmen vor dem Benutzer verstecken, gibt es extrem viele Details, die es zu kommentieren gilt.

[19] `http://de.wikipedia.org/wiki/C%2B%2B-Standardbibliothek`

13 Erweiterungen

Mit sinnvollen Erweiterungen der Aufgabenstellung, und der zugehörigen Lösung, kann man sich Zusatzpunkte verdienen. Dies ist in der ersten Runde meistens nicht entscheidend, — und somit ist auch davon abzuraten — da man zusätzlichen Arbeitsaufwand lieber in die Qualität der Lösung der eigentlichen Aufgabe steckt, und somit die Punktgrenze zur zweiten Runde erreicht. In der zweiten Runde können gute Erweiterungen bereits gute Lösungen noch verbessern.

Erweiterungen sollte man nur angehen, wenn einem nichts mehr einfällt, was man an der eigentlichen Lösung verbessern könnte — und uns fällt da eigentlich immer was ein.

Entscheidend dafür, dass eine Erweiterung gut ist, ist, dass sie *im Sinne der Aufgabenstellung* sowie *keine Fleißarbeit*, sondern *kreativ* ist.

Im Sinne der Aufgabenstellung

Dem *Sinn der Aufgabenstellung* folgt man am besten entlang der kurzen Geschichte, in die die eigentliche Aufgabe meinstens eingebettet ist, sowie dem theoretischen Problem. In unserem Fall besteht die Geschichte aus den die Addition übenden Grundschülern, und das theoretische Problem aus dem optimal zu wählenden Spielzug.

keine Fleißarbeit

Es wäre nun *reine Fleißarbeit*, die Aufgabenstellung zu verallgemeinern, indem man *beliebig viele* (sagen wir n) *statt der zwei Würfel* verwendet. Man müsste zwar die Funktion `Probability()` und die 12 in $2 * n$ ändern, aber der Kern des Algorithmus und damit auch der der Lösung hätte sich kein bisschen verändert — Dies wäre also keine gute Erweiterung!

Multiplikation statt Addition würde den algorithmischen Kern ebenfalls nicht verändern. (Auch wenn es entlang der Geschichte gut gedacht wäre!)

Auf der anderen Seite wäre eine *3D-Simulation*, wie Karten auf dem Tisch liegen und von einer Geisterhand umgedreht werden, auch nicht gut: Es hat nichts mit dem theoretischen Problem zu tun.

Was eventuell interessant wäre, ist nicht nur die Anzahl n der Würfel variabel zu gestalten, sondern auch die der Karten (sagen wir m). Damit müsste man nämlich die Implementation der Klasse `Cards` ändern, die ja bisher nur für $m \leq 32$ funktioniert. Die neue Laufzeit dann zu analysieren wäre vielleicht schon eine eigene Erweiterung. Eventuell ist diese Laufzeit dann so groß, dass das Berechnen des kompletten Graphens (also die Laufzeit des in unserer Lösung dargestellten Algorithmus) zu langsam ist, um noch auf heutigen Rechnern praktikabel zu sein. Dann könnte man also eine Heuristik für die so geänderte Aufgabe implementieren, diese dann an Beispielen testen, und ihre Qualität an kleineren Beispielen noch mit unserer optimalen Lösung von oben vergleichen. Dies wäre eine sehr interessante Erweiterung — für die zweite Runde.

Alternativ könnte man sich vielleicht überlegen, den Würfel durch eine Art Gegner zu ersetzen, um den Schritt, in dem der Erwartungswert berechnet wird, zu verändern.

(Aufgaben der zweiten Runde bieten meistens auch mehr Erweiterungsmöglichkeiten als die der ersten.)

14 Bücher

Hier ist nochmal ein separater Absatz zur Literatur. Die meisten Bücher gibt es sowohl auf Deutsch und auf Englisch, zum Teil sowohl gebunden als auch als Taschenbuch. Wir persönlich ziehen englische Bücher immer vor, da viele Fachbegriffe keine (oder nur eine schlechte) deutsche Übersetzung haben. Langfristig muss man sich spätestens im Studium sowieso mit englicher Literatur anfreunden. Gebundene Bücher sind in der Regel teurer, halten aber mehr aus. Es lohnt sich, vor dem Kauf ein paar Vergleiche zu machen und auch die letzte Ausgabe zu suchen. (Achtung: [11] und [12] waren in einer früheren Ausgabe zusammen in *einem* Buch; es lohnt sich also auch, kurz ins Inhaltsverzeichnis zu gucken!)

Die günstige Alternative zum Kauf der Bücher ist meistens die nächste (Universitäts-)Bibliothek. Die meisten der Kataloge sind wohl online. Zum Teil wird zwischen Haupt-, Abteilungsbibliothek und Lehrbuchsammlung unterschieden.

Zu allen Themen lassen sich auch sehr gute andere Bücher als Quellen angeben. Wir erwähnen hier unsere Empfehlungen, in absteigender Wichtigkeit.

Algorithmen (Stichwort Internationale Informatikolympiade[20]) sind für viele Aufgaben essentiell. Eine sehr gute Einführung ist [11, 12] mit Beispielen in C++ oder dasselbe [13] mit Beispielen in Java. Für Fortgeschrittene bietet sich das etwas mathematische [2] an, oder das lexikonartige [14].

Für eine oberflächliche *Allgemeinbildung im Bereich der theoretischen Informatik* gibt es das Buch "The New Turing Omnibus" [3], das häufig den Teilnehmern der Endrunde empfohlen wird. Detaillierter und mit Beweisen ist ein Buch von Prof. Uwe Schöning [10], langjähriger Juror der Endrunde des BWINF.

Für *Software-Design* empfiehlt sich [5]: Es bildet ein gemeinsames Vokabular, mit dem sich zahlreiche Beziehungen zwischen Klassen im Code beschreiben lassen. Die Verwendung des Vokabulars hilft, selbst die Situation zu erkennen und somit auch besser von Anfang an zu strukurieren. Sie verkürzt die Dokumentation und hilft somit, Ideen an den Leser zu vermitteln (der das Buch oder zumindest die Begriffe hoffentlich auch kennt; bei den Bewertern ist davon auszugehen).

Sind in einer Aufgabe Klassenbeziehungen zu vermitteln (zB. wenn ein Datenbankschema entworfen werden soll), so eignet sich meistens ein *unified modelling language (UML) class diagram* exzellent dazu. Auch das hat den Vorteil, dass die Zeichen standardisiert sind und so mit wenig viel und genau vermittelt werden kann. Ein recht kurzes Buch dazu ist [4].

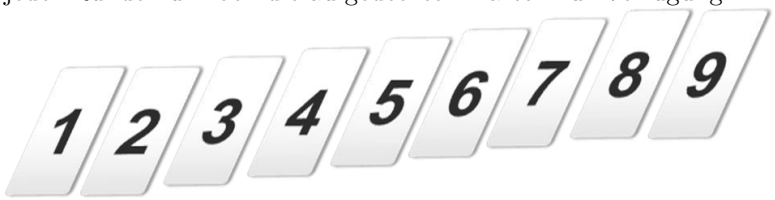
Allgemein empfiehlt sich nach den bereits genannten Büchern für *Java-Programmierer* [1] und für C++-Programmierer [7, 6, 8].

Von Tobias Thierer, einem ehemaligen Teilnehmer des BWINF und der Olympiadenlehrgänge, gibt es eine Webseite[15], die weitere Empfehlungen sowie einige kurze Kritiken enthält.

[20] siehe `http://www.ioi-training.de/wp/about/`

A Die Aufgabe: Drehzahl

Es gibt ein Kartenspiel, mit dem Grundschüler die Addition üben können. Auf dem Tisch liegen Karten, die mit den Zahlen 1 bis 9 beschriftet sind. Diese werden zu Beginn des Spiels aufgedeckt. Dann wird mit zwei Würfeln gewürfelt. Nach jedem Wurf muss der Spieler ein oder zwei beliebige Karten so umdrehen, dass die Summe der Zahlen auf den umgedrehten Karten der Augensumme der beiden Würfel entspricht. Es stehen in jeder Runde nur noch die aufgedeckten Karten zur Verfügung.



Wenn der Spieler keine Möglichkeit mehr hat, nach einem Wurf entsprechende Karten umzudrehen, wird die Summe der umgedrehten Karten als Bonuspunkte notiert. Ziel ist es, eine möglichst hohe Punktzahl zu erreichen.
Während des Spiels muss der Spieler immer wieder die Entscheidung fällen, welche Karten er umdreht. Würfelt er zu Beginn des Spiels z. B. die Augensumme 5, könnte er die Karten 1 und 4

oder die Karten 2 und 3

oder nur die Karte 5 umdrehen.

Aufgabe

Schreibe ein Programm, das dem Spieler in jeder Spielsituation zu seinem Wurf eine Kartenkombination vorschlägt, die zu einer möglichst hohen Endpunktzahl führt. Welche durchschnittliche Punktzahl pro Spiel kann ein Spieler erzielen, wenn er sehr viele Spiele mit stets optimalen Entscheidungen spielt?

B Mathematische Notationen

Hier wird kein Programm oder Pseudocode erklärt, sondern mathematische Notationen.

$D := 5$
: Eine *Definition*. D steht ab jetzt für den Wert 5.

$D = 5$
: Eine *Behauptung* (die vielleicht falsch ist, wie sich später herausstellt), ein *Schluß* oder eine *Aussage*. Ohne Kontext kann man nicht zwischen diesen Möglichkeiten unterscheiden.

$A := \{2, 3, 4, 5\}$
: Eine *Menge* von *Elementen*.

$3 \in A$
: 3 ist ein Element von A. 3 ist in A enthalten.

$B := \{\}$
: Die *leere Menge*. Zum Teil wird sie auch durch $\emptyset$ gekennzeichnet.

$C := \{3, 4\}$
: Eine andere Menge.

$D := \{5, 6\}$
: Und noch eine.

$E := \{A, C\} = \{\{2, 3, 4, 5\}, \{3, 4\}\}$
: Eine Menge von Mengen.

$C \subseteq A$
: C ist eine Teilmenge von A. D.h. alle Elemente in C sind auch in A.

$A \setminus C = \{2, 5\}$
: Die Menge der Elemente aus A ohne die aus C.

$|A| = 4$
: Die Menge A hat 4 Elemente.

$\mathbb{N} = \{0, 1, 2, \dots\}$
: Die natürlichen Zahlen.

$F := \{\, n \in \mathbb{N} \mid n \text{ gerade}\}$
: F ist *die Menge* der natürlichen Zahlen *mit der Eigenschaft,* dass sie gerade sind.

$G := \{M \in E \mid |M| = 2\} = \{\{3, 4\}\}$
: G ist die Menge der Elemente aus E, die 2 Elemente enthalten. Die Elemente aus E sind hier Mengen.

$\max\limits_{x \in A} x = 5$
: Das *Maximum* der Menge A.

$\max\limits_{x \in A} x^2 = 25$
: Das Maximum der Quadrate der Elemente aus A. Das Maximum wird bei $x = 5$ angenommen.

$\max\limits_{x \in A}(-x) = -2$
: Das Maximum von $-x$ für $x \in \{2, 3, 4, 5\}$.

$\max\limits_{x \in A}(x - 4)^2 = 4$
: Das Maximum wird bei $x = 2$ angenommen.

$\sum_{i \in A} i = 14$
: Die Summe aller Elemente in A.

$\sum_{i \in A} i^2 = 54$
: Die Summe aller Quadrate der Elemente in A.

Literatur

[1] Joshua Bloch. *Effective Java*. Prentice Hall, second edition, 2008.

[2] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. Mit Press, third edition, 2009.

[3] Alexander Dewdney. *The New Turing Omnibus: Sixty-Six Excursions in Computer Science*. Holt Paperbacks, 1993.

[4] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, third edition, 2003.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[6] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, 1996.

[7] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Design*. Addison-Wesley Professional, second edition, 1997.

[8] Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Professional, 2001.

[9] Achut Reddy. Java Coding Style Guide. http://developers.sun.com/sunstudio/products/archive/whitepapers/java-style.pdf, 2000. [Revision May 30, 2000].

[10] Uwe Schöning. *Theoretische Informatik - kurz gefasst*. Spektrum Akademischer Verlag, fifth edition, 2008.

[11] Robert Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley Professional, third edition, 1998.

[12] Robert Sedgewick. *Algorithms in C++, Part 5: Graph Algorithms*. Addison-Wesley Professional, third edition, 2002.

[13] Robert Sedgewick. *Algorithmen in Java. Teil 1-4: Grundlagen, Datenstrukturen, Sortieren, Suchen*. Pearson Studium, third edition, 2003.

[14] Steven Skieva. *The Algorithm Design Manual*. Springer, second edition, 2008.

[15] Tobias Thierer. Buchtipps. http://www.tobias-thierer.de/buchtipps.html.

[16] Benjy Weinberger, Craig Silverstein, Gregory Eitzmann, Mark Mentovai, and Tashana Landray. Google C++ Style Guide. http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml. [Revision 3.178].