26. Bundeswettbewerb Informatik, 2. Runde

Lösungshinweise und Bewertungskriterien

Allgemeines

Es ist immer wieder bewundernswert, wie viel an Ideen und Wissen, an Fleiß und Durchhaltevermögen in den Einsendungen zur zweiten Runde eines Bundeswettbewerbs Informatik steckt. Um aber die Allerbesten für die Endrunde zu bestimmen, müssen wir Ihre Arbeit kritisch begutachten und hohe Anforderungen stellen. Von daher sind Punktabzüge die Regel und Bewertungen über das Soll (5 Punkte) hinaus die Ausnahme, insbesondere bei den schwierigen Aufgaben dieses Jahres. Lassen Sie sich davon nicht entmutigen! Wie auch immer Ihre Einsendung bewertet wurde: Allein durch die Arbeit an den Aufgaben und den Einsendungen hat jede Teilnehmerin und jeder Teilnehmer einiges dazu gelernt; den Wert dieses Effektes sollten Sie nicht unterschätzen.

Bevor Sie sich in die Lösungshinweise vertiefen, lesen Sie doch bitte kurz die folgenden Anmerkungen zu Einsendungen und den beiliegenden Unterlagen durch.

Terminprobleme Einige Einsender gestehen ganz offen, dass Ihnen die Zeit zum Einsendeschluss hin knapp geworden ist, worauf wir bei der Bewertung leider keine Rücksicht nehmen können. Abiturienten macht der Terminkonflikt mit der Abiturvorbereitung Probleme. Der ist für eine erfolgreiche Teilnahme sicher nicht ideal. In der zweiten Jahreshälfte läuft aber die zweite Runde des Mathewettbewerbs, dem wir keine Konkurrenz machen wollen. Also bleibt uns nur die erste Jahreshälfte. Aber: Sie haben etwa vier Monate Bearbeitungszeit für die zweite BWINF-Runde. Rechtzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, Konflikte mit dem Abitur zu vermeiden.

Dokumentation Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und die Umsetzung in Software fließen lassen. Doch ohne eine gute Beschreibung der Lösungsideen, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine gute Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (die die verschiedenen bei der Lösung des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung wenig wert. Bewerterinnen und Bewerter können die Qualität Ihrer Einsendung nur anhand dieser Informationen vernünftig einschätzen. Mängel können nur selten durch gründliches Testen der eingesandten Programme ausgeglichen werden – wenn diese denn überhaupt ausgeführt werden können: Hier gibt es häufig Probleme, die meist vermieden werden könnten, wenn Lösungsprogramme vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner getestet würden. Insgesamt sollte die Erstellung des schriftlichen Materials die Programmierarbeit begleiten oder ihr teilweise sogar vorangehen: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt keine saubere Umsetzung in welche Programmiersprache auch immer hin.

Bewertungsbögen Kein Kreuz in einer Zeile bedeutet, dass die genannte Anforderung den Erwartungen entsprechend erfüllt wurde. Vermerkt wird also in der Regel nur, wenn davon abgewichen wurde – nach oben oder nach unten. Ein Kreuz in der Spalte „+“ bedeutet Zusatzpunkte, ein Kreuz unter „–“ bedeutet Minuspunkte für Fehlendes oder Unzulängliches. Die Schattierung eines Feldes bedeutet, dass die entsprechenden Plus- bzw. Minuspunkte in der Regel nicht vergeben wurden.

Lösungshinweise Bei den folgenden Erläuterungen handelt es sich um Vorschläge, nicht um die einzigen Lösungswege, die wir gelten ließen. Wir akzeptieren in der Regel alle Ansätze, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind. Einige Dinge gibt es allerdings, die – unabhängig vom gewählten Lösungsweg – auf jeden Fall diskutiert werden müssen. Zu jeder Aufgabe gibt es deshalb einen Abschnitt, indem gesagt wird, worauf bei der Bewertung letztlich geachtet wurde, zusätzlich zu den grundlegenden Anforderungen an Dokumentation und Quellcode.

Aufgabe 1: Berufsinformationstag

1.1 Voraussetzungen

Lehrer Lämpel möchte ein Werkzeug zur (ganzzahligen) linearen Programmierung benutzen, um einen möglichst „guten“ Ablaufplan für den Berufsinformationstag zu erhalten (eine genauere Diskussion darüber, was „gut“ heißt, folgt später). Er muss also für die bekannten Daten (Anzahl Schüler, Wünsche der Schüler, Beschränkungen der Zuhörermengen usw.) ein Programm erstellen, welches ihm eine Lösung für sein Problem liefert. Daher ist es am besten, einmal kurz zu definieren, was ein derartiges Programm eigentlich ist:

Das Wort *Programm* bedeutet hier nicht wie üblich eine Reihe von Anweisungen die nacheinander ausgeführt werden, sondern stellt eher eine Formulierung eines Problems dar, dessen Lösung mit externen Werkzeugen gefunden wird, über deren Funktionsweise nichts bekannt sein muss. Ein „Programm“ sagt also hier eher, **was** zu tun ist, nicht aber, **wie** man es tut.

Ein Problem besteht üblicherweise in der Maximierung (oder Minimierung) einer Zielfunktion, wobei die zur Verfügung stehenden Variablen nicht beliebige Werte annehmen dürfen, sondern ganz bestimmten Nebenbedingungen unterworfen sind.

Formal kann man ein lineares Optimierungsproblem (LP) wie folgt beschreiben[1]:

Maximiere (oder Minimiere)

$$z = c_1 x_{m+1} + c_2 x_{m+2} + \cdots + c_n x_{m+n} + c_0 \tag{1.1}$$

unter den linearen Nebenbedingungen

$$\begin{aligned} x_1 &= a_{11} x_{m+1} + a_{12} x_{m+2} + \cdots + a_{1n} x_{m+n} \\ x_2 &= a_{21} x_{m+1} + a_{22} x_{m+2} + \cdots + a_{2n} x_{m+n} \\ &\cdots\cdots\cdots\cdots\cdots \\ x_m &= a_{m1} x_{m+1} + a_{m2} x_{m+2} + \cdots + a_{mn} x_{m+n} \end{aligned} \tag{1.2}$$

und den folgenden Grenzen der Variablen

$$\begin{aligned} l_1 \leq\ & x_1 \ \leq u_1 \\ l_2 \leq\ & x_2 \ \leq u_2 \\ & \cdots\cdots \\ l_{m+n} \leq\ & x_{m+n} \leq u_{m+n} \end{aligned} \tag{1.3}$$

wobei $x_1, x_2, \ldots, x_m$ Schlupfvariablen; $x_{m+1}, x_{m+2}, \ldots, x_{m+n}$ strukturelle Variablen; z die Zielfunktion; $c_1, c_2, \ldots, c_n$ Koeffizienten in der Zielfunktion; c_0 den konstanten Term in der Zielfunktion; $a_{11}, a_{12}, \ldots, a_{mn}$ Koeffizienten in den linearen Nebenbedungungen; $l_1, l_2, \ldots, l_{m+n}$

[1] Entnommen aus der Dokumentation des GNU Linear Programming Kits.

untere Grenzen für die Variablen und $u_1, u_2, \ldots, u_{m+n}$ obere Grenzen für die Variablen darstellen.

Die Grenzen für die Variablen können endlich, aber auch unendlich sein; außerdem ist es möglich dass untere und obere Grenzen übereinstimmen. Daher sind folgende Typen von Variablen möglich:

Grenzen	Typ der Variable
$-\infty < x_k < +\infty$	freie (unbeschränkte) Variable
$l_k \leq x_k < +\infty$	nach unten beschränkte Variable
$-\infty < x_k \leq u_k$	nach oben beschränkte Variable
$l_k \leq x_k \leq u_k$	(doppelt) beschränkte Variable
$l_k = x_k = u_k$	fixierte Variable

Um das LP-Problem (1.1)–(1.3) zu lösen, müssen Werte für die Schlupf- und strukturellen Variablen gefunden werden, so dass:

- die linearen Nebenbedingungen (1.2) erfüllt sind und
- alle Variablen und Nebenbedingungen die Grenzen (1.3) erfüllen und dabei
- den größtmöglichen (oder kleinstmöglichen) Wert der Zielfunktion liefern.

Ein ganzzahliges lineares Optimierungsproblem (*Mixed integer linear programming, kurz: (MIP)*) ist ein gewöhnliches lineares Optimierungsproblem, bei dem zusätzlich gefordert wird, dass einige der Variablen nur ganze Zahlen annehmen dürfen. Auf den ersten Blick erscheint dieses Problem nicht wesentlich schwieriger, ist aber sowohl theoretisch als auch praktisch sehr viel härter.

1.2 Lösungsidee

Voraussetzungen

Um die Anforderungen an den Ablaufplan des Berufsinformationstags als LP/MIP-Problem zu formulieren, benötigt man zunächst einmal eine Übersicht, welche Daten zu Beginn der Planung vorliegen:

Grundlegende, vorhandene Daten sind:

N	=	Anzahl der teilnehmenden Schüler (zwischen 80 und 240)
M	=	Anzahl der verschiedenen Vorträge (hier 16)
D	=	Anzahl der Durchgänge (hier 4)
W	=	Maximale Anzahl der Wünsche pro Schüler (hier 6)

Das heißt, jeder Schüler bekommt eine Nummer zwischen 1 und N, und jedem Vortrag wird eine Nummer zwischen 1 und M gegeben.

Jeder Schüler gibt jetzt bis zu W Wünsche ab, das heißt es existiert ein Wunschzettel WZ mit $N \cdot W$ Einträgen der folgenden Art:

$$\mathrm{WZ}(i,w) = \begin{cases} j & \text{falls Schüler } i \text{ als } w\text{-ten Wunsch Vortrag } j \text{ angegeben hat} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

für $i = 1,\ldots,N$, $w = 1,\ldots,W$. Außerdem haben die Eltern angegeben, welche Kapazitätsgrenzen jeder ihrer Vorträge hat, es existiert also eine Tabelle KG mit

$$\mathrm{KG}(j,1) = \text{Untergrenze für Vortrag } j$$
$$\mathrm{KG}(j,2) = \text{Obergrenze für Vortrag } j$$

welche für jeden Vortrag Unter- und Obergrenze enthält, die für eine erfolgreiche Einrichtung des Vortrages nötig sind.

Modellierung des Berufsinformationstages

Dies waren bisher nur Voraussetzungen. Jetzt müssen dagegen Variablen eingeführt werden, deren Werte angeben, wie der Tag ablaufen wird. Man definiert also $N \cdot M \cdot D$ Variablen $S(i,j,k)$ mit

$$S(i,j,k) = \begin{cases} 1 & \text{Schüler } i \text{ besucht Vortrag } j \text{ in Durchgang } k \\ 0 & \text{sonst} \end{cases} \tag{1.4}$$

für $i = 1\ldots N$, $j = 1\ldots M$, $k = 1\ldots D$. Nach Abschluss der linearen Optimierung kann also aus den Variablen mit $S(i,j,k) = 1$ abgelesen werden, welcher Schüler welchen Vortrag wann besucht. Jede Variable $S(i,j,k)$ kann nur die Werte 0 und 1 annehmen, d.h. es gelten die Bedingungen

$$0 \le S(i,j,k) \le 1 \qquad \text{(Variablengrenzen)} \tag{1.5}$$
$$S(i,j,k) \in \mathbb{N} \qquad \text{(Ganzzahligkeit)} \tag{1.6}$$

welche in der Form (1.3) angegeben sind.

Bedingungen für zulässige Belegungen

In der Theorie sind die Variablen $S(i,j,k)$ zunächst einmal voneinander unabhängig, d.h. jede kann entweder den Wert 0 oder 1 annehmen, egal welche Werte die anderen Variablen besitzen. Bei beliebigen Belegungen entstehen aber Konstellationen, die entweder

- physikalisch unmöglich sind,
- nicht sinnvoll sind, weil sich jemand nicht an diesen Plan halten wird, oder
- den geforderten Voraussetzungen widersprechen.

Um diese Belegungen auszuschließen, sind also weitere Nebenbedingungen nötig.

Beispiel 1.1. *Was bedeutet die Belegung*

$$S(12,4,3) = 1$$
$$S(12,10,3) = 1$$

in der Realität? Nach der Definition (1.4) *heißt dies „Schüler 12 besucht Vortrag 4 in Durchgang 3“ und „Schüler 12 besucht Vortrag 10 in Durchgang 3“.*

Diese Belegung bedeutet also, dass ein Schüler ***gleichzeitig*** *zwei verschiedene Vorträge besucht, was physikalisch unmöglich ist. Daher müssen Bedingungen her, die derartige Belegungen verhindern.*

Das Beispiel zeigt, dass von den insgesamt M Variablen $S(i,1,k), S(i,2,k), \ldots, S(i,M,k)$ immer maximal eine einzige den Wert 1 haben kann, ansonsten würde ein Schüler zwei Vorträge gleichzeitg besuchen müssen. Diese Bedingung muss wieder in der Form (1.2), (1.3) formuliert werden. Glücklicherweise ist diese Bedingung leicht umzusetzen: Alle zulässigen Belegungen der Variablen ergeben in der Summe entweder 0 oder 1, nicht mehr. Damit erhält man die Bedingungen

$$0 \le \sum_{j=1}^{M} S(i,j,k) \le 1, \qquad i = 1\ldots N, \quad k = 1\ldots D \tag{B.1}$$

Man hat insgesamt $N \cdot D$ Bedingungen, weil eine solche Forderung für jeden Schüler in jedem einzelnen Durchgang gilt. Ausgeschrieben erhält man

$$\begin{array}{rcccl} 0 & \le & S(1,1,1)+\ldots+S(1,M,1) & \le & 1 \\ 0 & \le & S(1,1,2)+\ldots+S(1,M,2) & \le & 1 \\ & & \vdots & & \\ 0 & \le & S(1,1,D)+\ldots+S(1,M,D) & \le & 1 \\ 0 & \le & S(2,1,1)+\ldots+S(2,M,1) & \le & 1 \\ & & \vdots & & \\ 0 & \le & S(N,1,D)+\ldots+S(N,M,D) & \le & 1 \end{array}$$

Beispiel 1.2. *Was bedeutet die Belegung*

$$S(31,7,1) = 1$$
$$S(31,7,4) = 1$$

in der Realität? Nach der Definition (1.4) *heißt dies „Schüler 31 besucht Vortrag 7 in Durchgang 1“ und „Schüler 31 besucht Vortrag 7 in Durchgang 4“.*

Diese Belegung bedeutet also, dass ein Schüler einen Vortrag ***doppelt*** *besucht, was eigentlich kein Schüler tun wird. Wäre er zweimal für den gleichen Vortrag eingeplant, würde er wahrscheinlich einen der beiden Termine nicht wahrnehmen, was möglicherweise dazu führen könnte dass die Mindestanzahl der Teilnehmer dieses Vortrages an diesem Termin unterschritten werden könnte. Daher müssen Bedingungen her, die derartige Belegungen verhindern.*

Dieses Beispiel zeigt, dass von den insgesamt D Variablen $S(i,j,1), S(i,j,2), \ldots, S(i,j,D)$ maximal eine den Wert 1 haben darf, ansonsten würde der Ablaufplan vermutlich von Schüler i nicht eingehalten. Genau wie oben in der Bedingung (B.1) kann nun für jeden Schüler und jeden Vortrag gefordert werden

$$0 \leq \sum_{k=1}^{D} S(i,j,k) \leq 1, \qquad i = 1 \ldots N, \quad j = 1 \ldots M \tag{B.2}$$

mit der ausführlichen Schreibweise

$$\begin{array}{rcrcl}
0 & \leq & S(1,1,1) + \ldots + S(1,1,D) & \leq & 1 \\
0 & \leq & S(1,2,1) + \ldots + S(1,2,D) & \leq & 1 \\
 & & \vdots & & \\
0 & \leq & S(1,M,1) + \ldots + S(1,M,D) & \leq & 1 \\
0 & \leq & S(2,1,1) + \ldots + S(2,1,D) & \leq & 1 \\
 & & \vdots & & \\
0 & \leq & S(N,M,1) + \ldots + S(N,M,D) & \leq & 1
\end{array}$$

In die gleiche Richtung geht die Bedingung, dass keinem Schüler ein Vortrag zugeteilt werden sollte, den er sich nicht gewünscht hat. Also lässt sich als Erweiterung von (1.5) fordern

$$0 \leq S(i,j,k) \leq \begin{cases} 1 & \text{falls } \mathrm{WZ}(i,w) = j \text{ für ein } w \in \{1,\ldots,W\} \\ 0 & \text{sonst} \end{cases} \tag{B.3}$$

für $i = 1,\ldots,N$, $j = 1,\ldots,M$, $k = 1,\ldots,D$. Falls der Vortrag j nirgendwo bei Schüler i in der Liste der Wünsche auftaucht, besagt diese Bedingung dass

$$0 \leq S(i,j,k) \leq 0,$$

also

$$S(i,j,k) = 0,$$

d.h. in keinem einzigen Durchgang wird der Schüler den Vortrag besuchen müssen.

Jetzt fehlen noch die Bedingungen für die Mindest- und Höchstteilnehmerzahl. Die Teilnehmerzahlen $\mathrm{Teilnehmer}(j,k)$, die Vortrag j im k-ten Durchgang besuchen, kann mittels der Gleichungen

$$\mathrm{Teilnehmer}(j,k) = \sum_{i=1}^{N} S(i,j,k), \qquad j = 1,\ldots,M, \quad k = 1,\ldots,D \tag{1.7}$$

berechnet werden. Jetzt ist es auf den ersten Blick verführerisch einfach,

$$\mathrm{KG}(j,1) \leq \mathrm{Teilnehmer}(j,k) \leq \mathrm{KG}(j,2)$$

für alle $j = 1,\ldots,M, k = 1,\ldots,D$ zu fordern; leider fordert man hiermit aber zu viel. Die Bedingung sagt nämlich aus, dass **jeder** Vortrag in **jedem** Durchgang die Kapazitätsgrenzen erfüllen muss, insbesondere die Mindestteilnehmerzahl. Jeder Vortrag **muss** also immer stattfinden und dabei immer gut besucht sein. Dass dies in der Realität im Allgemeinen unmöglich ist, kann man sich leicht klarmachen, wenn man annimmt, dass einfach kein Schüler sich jemals Vortrag 1 wünscht. Mit dieser Bedingung wird das ganze Problem leicht unlösbar, denn für eine Lösung des MIP-Problems müssen *alle* Ungleichungen aus (1.3) erfüllt sein. Irgendwie muss also die Bedingung „Entweder innerhalb der zulässigen Grenzen, oder komplett gleich Null“ für die Teilnehmerzahl ausgedrückt werden.

Man fordert

$$\mathrm{FS}(j,k) \cdot \mathrm{KG}(j,1) \leq \mathrm{Teilnehmer}(j,k) \leq \mathrm{FS}(j,k) \cdot \mathrm{KG}(j,2) \tag{1.8}$$

für alle $j = 1,\ldots,M, k = 1,\ldots,D$. Dabei ist $\mathrm{FS}(j,k)$ die Kurzform von $\mathrm{Findet\_statt}(j,k)$, einer binären Variable, die angibt, ob Vortrag j in Durchgang k überhaupt stattfindet. Falls sie Null ist, muss dies auch für die Teilnehmerzahl gelten, ist sie Eins, muss die Teilnehmerzahl die Kapazitätsgrenzen in beide Richtungen erfüllen.

Damit sind alle Einschränkungen an einen Ablaufplan formuliert worden, jede zulässige Belegung der Variablen liefert einen gültigen, sinnvollen Ablaufplan für den Berufsinformationstag. Das Problem ist außerdem immer lösbar, denn für alle Werte von N, M, D, W ist

$$S(i,j,k) = 0 \tag{1.9}$$

$$\mathrm{Teilnehmer}(j,k) = 0 \tag{1.10}$$

$$\mathrm{FS}(j,k) = 0 \tag{1.11}$$

eine Belegung aller Variablen, die

- alle linearen Nebenbedingungen erfüllt,
- alle Grenzen der Variablen beachtet und
- die Ganzzahligkeit bestimmter Variablen respektiert.

Nur ist dieser Plan sicher nicht sinnvoll, kommt er doch einer Absage des kompletten Tages gleich. Um sinnvollere Belegungen der Variablen zu forcieren, braucht man eine Zielfunktion, deren Maximierung sinnvollere Belegungen bevorzugt.

Was macht also einen „guten“ Ablaufplan aus?

Kriterien für einen „guten“ Ablaufplan

Im Sinne eines MIP/LP-Problems muss nun eine Zielfunktion angegeben werden, die linear von den vorkommenden Variablen abhängt.

Sucht man eine Zielfunktion, kann man ganz allgemein erst einmal die Aufgabe

$$\text{Maximiere : (Wert positiver Aspekte)} - \text{(Wert negativer Aspekte)}$$

formulieren, die auf das jeweilige Problem anzupassen ist.

Ein offensichtliches Ziel der Veranstalter des Berufsinformationstags ist es, möglichst viele Wünsche der Schüler zu erfüllen. Man erhält als direkte Formulierung:

$$\text{Maximiere : } \sum_{\substack{1\le i\le N\\ 1\le j\le M\\ 1\le k\le D}} S(i,j,k) \tag{1.12}$$

Das folgende konstruierte Beispiel zeigt ein Dilemma dieser Zielfunktion auf:

Beispiel 1.3. *$N = 13$ Schüler äußern die Wünsche in der folgenden Tabelle, es gibt $M = 6$ mögliche Vorträge die in $D = 3$ Durchgängen stattfinden können.*

Schüler	*W1*	*W2*	*W3*	*W4*
1	*1*	.	.	.
2	*3*	*1*	.	.
3	*3*	*1*	.	.
4	*3*	*1*	*2*	.
5	*3*	*1*	*2*	.
6	*3*	*1*	*2*	.
7	*3*	*1*	*2*	.
8	*3*	*1*	*2*	.
9	*3*	*1*	*2*	.
10	*4*	*1*	*2*	*6*
11	*4*	*1*	*6*	*2*
12	*4*	*1*	*2*	*5*
13	*1*	*2*	*5*	.

Die Kapazitätsgrenzen der Vorträge in der folgenden Tabelle sind zu beiden Seiten genau so groß gewählt wie es Schüler gibt, die sich den jeweiligen Vortrag anhören wollen. Entweder gehen also alle in einen Vortrag oder keiner.

Vortrag j	*KG(j,1)*	*KG(j,2)*
1	*13*	*13*
2	*10*	*10*
3	*8*	*8*
4	*3*	*3*
5	*2*	*2*
6	*2*	*2*

Man erkennt, dass die Vorträge 1–3 jeweils nicht gleichzeitig stattfinden können, weil sich manche Schüler für jeden dieser Vorträge interessieren. Jeder dieser Vorträge kann (und muss) eingerichtet werden, weil in jedem allein schon mehr Schüler unterkommen, als in den Vorträgen 4–6 zusammen. Werden die Vorträge 1–3 also in den Durchgängen 1–3 angeboten, erhält man die folgende Tabelle, in der die noch nicht berücksichtigten Wünsche stehen.

Schüler	*W1*	*W2*	*W3*	*W4*
1	.	.	.	.
2	.	.	.	.
3	.	.	.	.
4	.	.	.	.
5	.	.	.	.
6	.	.	.	.
7	.	.	.	.
8	.	.	.	.
9	.	.	.	.
10	*4*	.	.	*6*
11	*4*	.	*6*	.
12	*4*	.	.	*5*
13	.	.	*5*	.

Nun ist es möglich (immer parallel zu Vortrag 3), entweder Vortrag 4 für die Schüler 10, 11 und 12 einzurichten oder aber die Vorträge 5 und 6 für die Schüler 10–14. Welchen Weg geht dabei die Maximierung der Zielfunktion (1.12)*?*

Weil vier weitere Schüler in Vorträgen besser sind als drei, richtet das Programm nach (1.12) *die Vorträge 5 und 6 ein.*

Betrachtet man aber im Beispiel, welche Wünsche erfüllt wurden, ergibt sich die Frage:

> „Was ist besser? Sollen lieber zwei dritte und zwei vierte Wünsche erfüllt werden oder drei erste Wünsche?“

Da die Schüler die wichtigsten Wünsche zuerst angeben sollten, bietet es sich an, vordere Wünsche bevorzugt zu erfüllen, auch wenn damit wie gesehen evtl. nicht so viele Schüler wie möglich auf die Vorträge verteilt werden. Ein Ausweg ist es, in der Zielfunktion (1.12) jeden

Summand unterschiedlich stark zu gewichten, entsprechend der Reihenfolge der Wünsche jedes einzelnen Schülers:

$$\text{Maximiere}: \sum_{\substack{1\le i\le N\\ 1\le j\le M\\ 1\le k\le D}} w(i,j)\cdot S(i,j,k) \tag{1.13}$$

Eine mögliche Gewichtung wäre zum Beispiel die folgende:

Wunsch l	1.	2.	3.	4.	5.	6.
Gewicht $g(l)$	3.0	2.5	2.0	1.5	1.2	1.0

Das Verhältnis jeweils zweier Zahlen zeigt an, welche Wünsche das Programm erfüllt, wenn es die Wahl hat. Zum Beispiel ist $\frac{3.0}{2.5}=\frac{6}{5}$, d.h. fünf erste Wünsche entsprechen genau sechs zweiten Wünschen, wohingegen ein erster Wunsch sogar so viel wert ist wie drei sechste Wünsche. Die Gewichte in (1.13) leiten sich dann wie folgt daraus ab:

$$w(i,j)=\begin{cases} g(l) & \text{falls Schüler } i \text{ Vortrag } j \text{ als } l\text{-ten Wunsch hat}\\ 0 & \text{sonst}\end{cases}$$

Damit würde im obigen Beispiel 1.3 Vortrag 4 eingerichtet, denn drei erfüllte erste Wünsche ($3\times 3.0=9.0$ Punkte) wiegen in (1.13) schwerer als zwei dritte und zwei vierte Wünsche ($2\times 2.0+2\times 1.5=7.0$ Punkte).

Einschränkung der Möglichkeiten

Man kann beobachten, dass mit einem gültigen Plan auch jede Permutation der Durchgänge einen gültigen Plan liefert. Es lassen sich sinnvolle einschränkende Bedingungen dazu finden, in welcher Reihenfolge die Durchgänge stattzufinden haben. Zum Beispiel gibt es

- die Möglichkeit, dass die Anzahl der beteiligten Schüler in jedem weiteren Durchgang nicht größer wird: schließlich nimmt deren Konzentration im Laufe des Tages immer mehr ab, außerdem macht es Sinn nach der Begrüßung direkt mit möglichst vielen Schülern zu starten. Dann erhält man folgende Ungleichungen (im Fall von vier Durchgängen):

$$\sum_{\substack{1\le i\le N\\ 1\le j\le M}} S(i,j,1) \ge \sum_{\substack{1\le i\le N\\ 1\le j\le M}} S(i,j,2) \ge \sum_{\substack{1\le i\le N\\ 1\le j\le M}} S(i,j,3) \ge \sum_{\substack{1\le i\le N\\ 1\le j\le M}} S(i,j,4) \tag{1.14}$$

- die Möglichkeit, dass die Anzahl der vortragenden Elternteile in jedem weiteren Durchgang nicht größer wird: schließlich ist man auch im nächsten Jahr auf deren Kooperation angewiesen, und Elternteile, die früh wieder nach Hause können (bei der Begrüßung werden wohl alle anwesend sein), sind glücklicher bezüglich ihres Zeitaufwandes. Die Ungleichungen lauten

$$\sum_{1\le j\le M} \mathrm{FS}(j,1) \ge \sum_{1\le j\le M} \mathrm{FS}(j,2) \ge \sum_{1\le j\le M} \mathrm{FS}(j,3) \ge \sum_{1\le j\le M} \mathrm{FS}(j,4) \tag{1.15}$$

Beide einschränkenden Bedingungen legen die Reihenfolge der Durchgänge bei einem optimalen Ablaufplan für sich genommen jeweils ziemlich fest; dagegen ist es möglich, dass, sollten beide Bedingungen gefordert werden, der optimale Plan nicht durchzuführen ist. In diesem Fall sorgen beide Bedingungen in Kombination dafür, dass das Ergebnis der Maximierung ein nicht optimaler Plan ist.

Der Fehler, dass durch Nebenbedingungen das Problem überbestimmt und eine nicht optimale Lösung gefunden wird oder das Problem sogar komplett unlösbar wird, kann bei linearer Optimierung leicht auftreten und ist durch sorgfältiges Aufstellen der Ungleichungen zu vermeiden.

Die folgenden Ungleichungen (jetzt einmal für den festen Fall von $D=4$ Durchgängen formuliert) dagegen schränken im Allgemeinen die Möglichkeiten bereits für sich genommen ein, die restlichen Lösungen haben aber dafür aber bestimmte Eigenschaften:

- Die Formulierung „eine Veranstaltung auch mehrmals nacheinander durchzuführen“ ist als „unmittelbar hintereinander“ zu verstehen. Für einen festen Vortrag j ist bei den drei aufeinanderfolgenden Variablen

$$\big(\mathrm{FS}(j,k),\mathrm{FS}(j,k+1),\mathrm{FS}(j,k+2)\big)$$

jede Belegung erlaubt, außer der Kombination $(1,0,1)$, bei der Viererkombination

$$\big(\mathrm{FS}(j,k),\mathrm{FS}(j,k+1),\mathrm{FS}(j,k+2),\mathrm{FS}(j,k+3)\big)$$

jede Kombination außer $(1,0,0,1)$. Diese würden nämlich Pausen für ein Elternteil bedeuten, die sich mit den Ungleichungen

$$\mathrm{FS}(j,1)-\mathrm{FS}(j,2)+\mathrm{FS}(j,3)\le 1 \tag{1.16}$$
$$\mathrm{FS}(j,2)-\mathrm{FS}(j,3)+\mathrm{FS}(j,4)\le 1 \tag{1.17}$$
$$\mathrm{FS}(j,1)-\mathrm{FS}(j,2)-\mathrm{FS}(j,3)+\mathrm{FS}(j,4)\le 2 \tag{1.18}$$

eliminieren lassen.

- Schüler sollten keine Pausen in ihrem Plan haben, es lassen sich ganz ähnliche Ungleichungen wie bei den Eltern aufstellen.

1.3 Implementierung

Im Allgemeinen wird aus nahe liegenden Gründen in den Lösungshinweisen auf konkrete Codebeispiele verzichtet. Auf Grund der besonderen Art der Aufgabenstellung folgt hier trotzdem eine beispielhafte Implementierung der Lösungsidee in der GNU MathProg-Sprache. Die meisten Lösungsideen sind im Code nachzuvollziehen, teilweise sind Sektionen nach Bedarf auskommentiert. Soll kein Standalone-Solver verwendet werden, übergibt man die Modellierung des Problems mit Hilfe von Schnittstellenfunktionen an das als Bibliothek eingebundene Lösungssystem.

Aufteilung

Das Problem wird in Modell- und Datensektionen gespalten, die beide in unterschiedlichen Dateien zu finden sind. Damit können flexibel unterschiedliche Datensätze verarbeitet werden.

Der Aufruf des Standalone-Solvers `glpsol` lautet dann z.B. folgendermaßen:

```
glpsol --math problem.mod --data demo.dat
```

In der Praxis hat sich für das gegebene Problem bewährt, die zusätzlichen Parameter `--intopt` (verbesserte interne Algorithmen) und `--bestp` (günstige Branch-and-Bound-Methode) zu verwenden, gepaart unter Umständen für größere Schülerzahlen mit der Option `--tmlim t` (nach t Sekunden abbrechen und die beste bisher gefundene MIP-Lösung ausgeben).

Bindet man dagegen Bibliotheken ein, sollten diese Optionen im Programm gesetzt werden.

Listings

```
/*
 *   Datei: problem.mod
 */

param ANZ_SCHUELER,    integer, > 0;
param ANZ_VORTRAEGE,   integer, > 0;
param ANZ_DURCHGAENGE, integer, > 0;
param ANZ_WUENSCHE,    integer, > 0;

set Schueler    := 1..ANZ_SCHUELER;
set Vortraege   := 1..ANZ_VORTRAEGE;
set Durchgaenge := 1..ANZ_DURCHGAENGE;
set Wuensche    := 1..ANZ_WUENSCHE;

param GRENZEN{j in Vortraege, l in 1..2 }, integer, > 0;
```



```
param WEIGHTS{l in Wuensche};

param WUNSCHZETTEL{i in Schueler, l in Wuensche}, integer default 0;

param Multiplikator{i in Schueler, j in Vortraege} :=
sum{l in Wuensche} (
  if WUNSCHZETTEL[i,l]=j then
    WEIGHTS[l]
  else
    0
);

var  S       {i in Schueler, j in Vortraege, k in Durchgaenge }, binary;
s.t. s_cond1 {i in Schueler, j in Vortraege } :
        sum{k in Durchgaenge} S[i,j,k] <= 1;
s.t. s_cond2 {i in Schueler, k in Durchgaenge } :
        sum{j in Vortraege} S[i,j,k] <= 1;
s.t. s_cond3 {i in Schueler, j in Vortraege, k in Durchgaenge :
                not (exists{l in Wuensche} WUNSCHZETTEL[i,l]=j) } :
        S[i,j,k] = 0;

/*
 * Teilnehmer[j,k] gibt an, wieviele Schüler Vortrag j in Durchgang k
 * besuchen, diese Zahl wird den Mindest- und Höchstangaben genügen
 * müssen, oder Null sein
 */
var  Teilnehmer      {j in Vortraege, k in Durchgaenge}, integer;
s.t. teilnehmer_cond {j in Vortraege, k in Durchgaenge} :
        Teilnehmer[j,k] = sum{i in Schueler} S[i,j,k];

/*
 * FS[j,k] gibt an, ob Vortrag j in Durchgang k überhaupt stattfindet
 */
var  FS          {j in Vortraege, k in Durchgaenge}, binary;
s.t. grenz_cond1 {j in Vortraege, k in Durchgaenge} :
        FS[j,k]*GRENZEN[j,1] <= Teilnehmer[j,k];
s.t. grenz_cond2 {j in Vortraege, k in Durchgaenge} :
        Teilnehmer[j,k] <= FS[j,k]*GRENZEN[j,2];

/*
 * Bedingungen für die Sortierung der Durchgänge nach bestimmten Kriterien
 */
s.t. decreasing_children_cond{k in 1..(ANZ_DURCHGAENGE-1)} :
        sum{i in Schueler,j in Vortraege} S[i,j,k] >=
        sum{i in Schueler,j in Vortraege} S[i,j,k+1];
```

```
# s.t. decreasing_parents_cond{k in 1..(ANZ_DURCHGAENGE-1)} :
#        sum{j in Vortraege} FS[j,k] >= sum{j in Vortraege} FS[j,k+1];


/*
 * Zielfunktion
 */
maximize obj: sum{i in Schueler, j in Vortraege, k in Durchgaenge}
         Multiplikator[i,j] * S[i,j,k];

solve;

/********** print section **********/

printf "\n";
printf "*************************\n";
printf "* Terminpläne der Schüler *\n";
printf "*************************\n";
printf "           ";
for {k in Durchgaenge} printf "  D%02d",k;
printf "    | ";
for {l in Wuensche} printf "  W%d",l;
printf "\n";
for {i in Schueler} {
  printf "Schüler %3d:", i;
  for {k in Durchgaenge} printf "   %2d", sum{j in Vortraege} j*S[i,j,k];
  printf "    | ";
  for {l in Wuensche} printf "  %2s", if WUNSCHZETTEL[i,l]>0
                                      then WUNSCHZETTEL[i,l]
                                      else "";
  printf("\n");
}
printf "\n";
printf "*************************\n";
printf "* Belegung der Vorträge *\n";
printf "*************************\n";
printf "           ";
for {k in Durchgaenge} printf "  D%02d",k;
printf "\n";
for {j in Vortraege} {
  printf "Vortrag %2d:", j;
  for {k in Durchgaenge} printf "   %2d", sum{i in Schueler} S[i,j,k];
  printf "    ( %2d <= X <= %2d ) (%d Wünsche)\n",GRENZEN[j,1],GRENZEN[j,2],
         sum{i in Schueler, l in Wuensche:WUNSCHZETTEL[i,l]=j} 1; }
printf "           ";
for {k in Durchgaenge} printf "-----",k;
```



```
printf "\n";
printf "Schüler:   ";
for {k in Durchgaenge}
  printf "   %2d", sum{i in Schueler, j in Vortraege} S[i,j,k];
printf "\n";
printf "Eltern:    ";
for {k in Durchgaenge} printf "   %2d", sum{j in Vortraege} FS[j,k];
printf "\n";

printf "\n";
printf "*********************************\n";
printf "* Maximalwert der Zielfunktion *\n";
printf "*********************************\n";
printf "%f\n", sum{i in Schueler, j in Vortraege, k in Durchgaenge}
         Multiplikator[i,j] * S[i,j,k];
end;
/*
 *   Ende: problem.mod
 */
```

Ein Datenfile hat dann z.B. folgendes Aussehen:

```
/*
 *   Datei: demo.dat
 */

data;

param ANZ_SCHUELER    := 13;
param ANZ_VORTRAEGE   := 6;
param ANZ_DURCHGAENGE := 3;
param ANZ_WUENSCHE    := 4;

param WUNSCHZETTEL
 : 1 2 3 4 :=
 1 1 . . .
 2 3 1 . .
 3 3 1 . .
 4 3 1 2 .
 5 3 1 2 .
 6 3 1 2 .
 7 3 1 2 .
 8 3 1 2 .
 9 3 1 2 .
```

```
10 4 1 2 6
11 4 1 6 2
12 4 1 2 5
13 1 2 5 .;

param GRENZEN
:   1     2 :=
1  13    99
2  10    99
3   8    99
4   3    99
5   2    99
6   2    99;

param WEIGHTS := 1 2.00000
                 2 1.66666
                 3 1.33333
                 4 1.00000;

end;
/*
 *   Ende: demo.dat
 */
```

Die Ausgabe des Befehls `glpsol --math problem.mod --data demo.dat` liefert dann die folgende Ausgabe:

```
*************************
* Terminpläne der Schüler *
*************************
              D01  D02  D03     |   W1  W2  W3  W4
Schüler   1:    1    0    0     |    1
Schüler   2:    1    3    0     |    3   1
Schüler   3:    1    3    0     |    3   1
Schüler   4:    1    3    2     |    3   1   2
Schüler   5:    1    3    2     |    3   1   2
Schüler   6:    1    3    2     |    3   1   2
Schüler   7:    1    3    2     |    3   1   2
Schüler   8:    1    3    2     |    3   1   2
Schüler   9:    1    3    2     |    3   1   2
Schüler  10:    1    4    2     |    4   1   2   6
Schüler  11:    1    4    2     |    4   1   6   2
Schüler  12:    1    4    2     |    4   1   2   5
Schüler  13:    1    0    2     |    1   2   5
```



```
************************
* Belegung der Vorträge *
************************
             D01  D02  D03
Vortrag  1:   13    0    0    ( 13 <= X <= 99 ) (13 Wünsche)
Vortrag  2:    0    0   10    ( 10 <= X <= 99 ) (10 Wünsche)
Vortrag  3:    0    8    0    (  8 <= X <= 99 ) (8 Wünsche)
Vortrag  4:    0    3    0    (  3 <= X <= 99 ) (3 Wünsche)
Vortrag  5:    0    0    0    (  2 <= X <= 99 ) (2 Wünsche)
Vortrag  6:    0    0    0    (  2 <= X <= 99 ) (2 Wünsche)
           ---------------
Schüler:      13   11   10
Eltern:        1    2    1

*******************************
* Maximalwert der Zielfunktion *
*******************************
57.666560
```

Die Ergebnisse stimmen mit den von Hand für Beispiel 1.3 vorausgesagten Ergebnissen überein: Das Programm bevorzugt die Durchführung von Vortrag 4 gegenüber den Vorträgen 5 und 6; außerdem wurden die Vorträge so auf die Durchgänge verteilt, dass die Schülerzahlen immer kleiner werden, die Zahl der eingesetzten Eltern dagegen nicht (diese Forderung war auskommentiert).

1.4 Bewertungskriterien

Optimalitätskriterien und Bedingungen Teilaufgabe 1 fordert die Entwicklung von Kriterien, die eine Lösung unseres Planungsproblems erfüllen sollte. Diese Kriterien sollen die verschiedenen Interessen der Beteiligten berücksichtigen; diese Interessen sind übersichtlich darzulegen. Teilaufgabe 2 fordert die Identifikation von Bedingungen für die Zulässigkeit einer Lösung; auch hier sollte eine übersichtliche, in sich widerspruchsfreie Liste von Bedingungen vorhanden sein. Insbesondere sollten durch die Bedingungen inhaltlich unmögliche oder nicht sinnvolle Lösungen (etwa die doppelte Zuweisung von Vorträgen an einen Schüler oder die Zuweisung von nicht gewünschten Vorträgen) ausgeschlossen sein.

Trennung von Modell/Programm und Daten Die Eingabedaten sind vom Programm (bzw. vom Modell) zu trennen, damit Lehrer Lämpel es auch im nächsten Jahr noch unverändert verwenden und auch an Kollegen mit ähnlichen Aufgaben weitergeben kann.

Modellierung Die Modellierung mit Hilfe von MathProg (für GLPK) oder einer anderen LP/MIP-Problembeschreibungssprache sollte grundsätzlich geeignet, gut erläutert und der zugehörige „Quellcode“ gut nachvollziehbar sein. Vor der Umsetzung der Kriterien und Bedingungen in die Beschreibungssprache (oder entsprechender Aufrufe der Schnittstellenfunktionen des Lösungssystems) sollten diese auch einigermaßen formal, also in mathematischer Notation, spezifiziert worden sein. Dies kann auch im Rahmen der Bearbeitung der Teilaufgaben 1 und 2 geschehen.
Das erstellte Modell muss zunächst zu zulässigen Lösungen gemäß der Bedingungen aus Teilaufgabe 2 führen. Diese müssen also korrekt umgesetzt sein. Ebenso müssen die in Teilaufgabe 1 definierten Kriterien berücksichtigt werden. Auch wenn Problembeschreibungen vom eigenen Programm automatisch generiert und direkt an das Lösungssystem weitergereicht werden, sollten Beispiele dafür dokumentiert sein.
Eine besondere Leistung ist es, Bedingungen an die Räume oder andere besondere Bedingungen aufzustellen und zu erfüllen.

Lösbarkeit Die Modellierung beeinflusst auch die Lösbarkeit des Problems. Zu viele Nebenbedingungen können das Problem unlösbar machen. Eine gute Bearbeitung muss sicherstellen (und idealerweise begründen), dass das Problem mit der gewählten Modellierung gelöst werden kann. Für die in der Aufgabenstellung genannten Zahlen muss eine Lösung gefunden werden.

Interface Die Aufgabenstellung spricht von einem leicht zu benutzenden Programm. Der Benutzer sollte also die Eingabedatei nicht selbst erstellen müssen, die Ausgabe sollte übersichtlich sein. Eine aufwändige grafische Schnittstelle ist nicht nötig, Ein- und Ausgabe können durchaus im Textformat erfolgen.

Beispiele Die Leistungsfähigkeit des Modells muss wie üblich mit Beispielen gezeigt werden. Darunter sollten auch kleinere, gut nachvollziehbare Beispiele sein. Idealerweise werden auch Lösungsvarianten anhand von Beispielen erläutert. Für die in der Augabenstellung genannten Zahlen sollte es selbstverständlich auch ein Beispiel geben (vgl. Bewertungskriterium „Lösbarkeit“).

Aufgabe 2: Blocksberg

Die Aufgabe ist dem *List-Update*-Problem nachempfunden, das im Bereich der so genannten *Online-Algorithmen* untersucht wird, also Algorithmen, deren Eingabedaten erst im Verlauf der Ausführung vollständig bekannt werden. Hier ist eine Übersetzung zwischen den beiden Domänen: Der Stapel von Blöcken ist die Liste von Elementen. Die von Bibi erstellte Liste ist die Folge der Anfragen. Ben ist der Online-Algorithmus, und Bibi ist der Offline-Algorithmus. Die Konstante c hat viel mit dem sogenannten *Kompetitivitätsfaktor* zu tun. Eine kurze Begriffsübersicht zum Thema Online-Algorithmus liefert Wikipedia[2]. Im Folgenden stehe A für Bibi und B für Ben.

2.1 Offene Stellen der Spielanleitung

Wodurch wird die Länge der Liste bestimmt? Ist sie fest vorgegeben, nach oben beschränkt oder wird sie B wenigstens vorher mitgeteilt? Alle Alternativen sind denkbar, aber am einfachsten ist sicherlich: vorgegeben und B nicht bekannt, denn dann müssen weder die Strategie von A noch die von B der Listenlänge viel Aufmerksamkeit widmen. Ist die Liste viel länger als die Anzahl der Blöcke? Wenn ja, ist auf beiden Seiten mehr Strategie möglich und das Spiel ist weniger ein Glücksspiel, daher ist diese Option zu bevorzugen.

Wie liegen die Blöcke am Anfang? Antwort: Wenn A das weiß, ist das völlig egal, da die Nummern nur der Identifikation dienen. O.B.d.A. kann man davon ausgehen, dass der k-te Block von oben die Nummer k trägt – was auch ein paar Zeilen Code spart. Daraus ergibt sich eine Beobachtung, für die ich allerdings keinen praktischen Nutzen finden konnte: Statt die Blöcke physisch zu verschieben, kann man auch durch entsprechendes Umbenennen der Zahlen in der Liste eine äquivalente Spielposition schaffen. B's Einflussmöglichkeiten werden dadurch verwandelt in eine eingeschränkte Veränderung der Liste und die Blöcke werden nicht mehr benötigt: eine Sichtweise, die manches vielleicht einfacher macht. Wenn A die Anfangsreihenfolge nicht kennen soll, ist sie nicht völlig egal. Erreichen ließe sich das, indem nach dem Erstellen der Liste die Blöcke in eine Zufallsreihenfolge gebracht werden oder B die Blöcke umsortieren darf. Der Effekt ist allerdings gering, weil durch die Anfangsposition für jeden Block die Wertung nur einmal beeinflusst wird.

Wie liegen die Blöcke vor dem zweiten Durchgang? So wie sie nach dem ersten lagen: In diesem Fall hat B, falls er die Länge der Liste kennt, die Möglichkeit, die Blöcke gegen Ende des ersten Durchgangs in eine Reihenfolge zu bringen, die ihm im zweiten Durchgang am Anfang mehr Punkte bringt. Der Vorteil ist aber klein und wirkt sich für jeden Block nur einmal aus. So wie sie vor dem ersten lagen: Macht die Sache einfacher und irgendwie fairer. Da der Effekt sowieso gering ist, bevorzuge ich diese Alternative.

[2] http://de.wikipedia.org/wiki/Online-Algorithmus

Gibt es eine oder mehrere Runden? Mehrere wären durchaus vorstellbar, aber die genaue Organisation eines Spielablaufs mit mehreren Runden lässt so viele Freiheiten, dass diese Möglichkeit als Erweiterung und nicht als Lücke in der Spielbeschreibung anzusehen ist. Ich mache mir mal keine weiteren Gedanken darüber.

2.2 Komponenten und Ausgaben des Programms

Bei der Programmierung des Spiels stellen sich u.a. folgende Fragen: Welche Komponenten gibt es, wie verläuft die Interaktion der Komponenten, und welche interessanten Ausgaben kann man sich anzeigen lassen? Zunächst scheint es naheliegend, je eine Komponente für A bzw. B zu realisieren. Eine vernünftige Implementierung sollte aber auf tiefer gehenden Überlegungen basieren. Insbesondere wird für eine schummelsichere Implementierung eine Spielleiter-Komponente benötigt, die jeweils den Kontrollfluss und eine Kopie der Daten an die Spieler übergibt und anschließend deren Antwort auf die Originaldaten anwendet und die Punkte zählt. Konkret bedeutet das, dass der Spielleiter zumindest den Stapel mit den Blöcken, die Punktzahlen von A und B sowie die Liste verwaltet, um Manipulationen vorzubeugen.

Das Spiel lässt sich in 4 Phasen unterteilen, für die Programmkomponenten zu realisieren sind (wenn man auf die Forderung der Schummelsicherheit verzichtet, wird der ganze Ablauf natürlich einfacher). Für diese Phasen wird beschrieben, was die einzelnen Komponenten leisten und wie der Kontrollfluss ist. Phasen 1 und 3 betreffen A; ob diese programmiertechnisch als zwei Komponenten realisiert oder in einer Komponente zusammengefasst werden, ist unwesentlich und wird auch von der verwendeten Programmiersprache abhängen.

Phase 1: A erzeugt eine Liste Je nachdem, ob die Länge der Liste und die Blockanzahl vor dem Spiel festgelegt wurden oder von A ausgesucht werden dürfen, werden die entsprechenden Zahlen an diese Komponente übergeben bzw. von ihr zurückgegeben. Auf jeden Fall enthält das Ergebnis die Liste, die angezeigt werden soll. Weil die Liste extrem lang und manchmal uninteressant sein kann, wäre es hilfreich, wenn man ihre Ausgabe auch unterdrücken kann. Interessant wäre es auch, die Liste mit Kommentaren zu versehen, welche beschreiben, warum eine bestimmte Zahlenkonstellation für besonders fies gehalten wird. Diese Kommentare können mit angezeigt und anschließend aus der Liste entfernt werden. Wenn der Spielleiter dem Listenerzeuger zutraut, dass er eine ungültige Liste erzeugen wird, kann auch noch überprüft werden, ob die Liste wirklich nur aus Zahlen aus $\{1, \ldots, \ell\}$ besteht.

Phase 2: B arbeitet die Liste ab Um zu verhindern, dass B heimlich doch sein Wissen über den Rest der Liste ausnutzt, muss der Kontrollfluss zwischen der B-Komponente und dem Spielleiter mehrere Male hin und her wechseln, wobei der Spielleiter jeweils A's Punktzahl feststellt und aufaddiert, B das nächste Listenelement mitteilt, B's Antwort entgegennimmt und überprüft, die Permutation der Blöcke entsprechend verändert und den Zyklus von vorne beginnt. B's Aufgabe besteht darin zu bestimmen, um wie viele Blöcke der aktuelle Block nach oben wandern soll, und dann den Kontrollfluss an den Spielleiter zurückzugeben. Da

der Spielleiter B misstraut und die Permutation der Blöcke lieber selbst vornimmt, kann man B diese Arbeit ersparen; die aktuelle Permutation wird bei jedem Aufruf an B übergeben. Interessante Ausgaben sind hier: die Spielzüge von B, die Entwicklung der Punktzahl von A, die Entwicklung des Stapels und die Häufigkeitsverteilung der Spielzüge. B kann auch noch Kommentare abgeben, die seine Entscheidungen begründen, was insbesondere fürs Debugging hilfreich ist.

Phase 3: A arbeitet die Liste ab Da A die Liste ohnehin kennt, wird die ganze Heimlichtuerei unnötig und man braucht nur einen Aufruf. Das Ergebnis ist eine Liste von Spielzügen, die vom Spielleiter dann ausgeführt werden, während er die Punkte zählt. Die interessanten Ausgaben sind hier wesensmäßig dieselben wie bei Phase 2.

Phase 4: Siegerehrung Interessante Ausgaben sind neben den Punktzahlen der Spieler vor allem deren Verhältnis und als Vergleichswert für die Güte von B's Spiel das Ergebnis, das bei gleichverteilt zufälliger Auswahl der Spielzüge herausgekommen wäre. Da einige Ergebnisse der vorigen Phasen sehr umfangreich werden können und man sie nicht immer sehen möchte, könnte hier auch dem Benutzer eine Gelegenheit gegeben werden, mittels einer interaktiven Schnittstelle (Prompt, Menü, Gedankenhelm, ...) diese Werte abzufragen.

2.3 Strategische Überlegungen

A erzeugt eine Liste

- Die einfachste Möglichkeit: A erzeugt die Liste komplett zufällig. Für B ist es unmöglich, dieser Strategie gezielt etwas entgegen zu setzen. Sie kann als Maßstab für die Bewertung anderer, ausgefuchsterer Strategien dienen oder bei der Bewertung von Strategien für B vorausgesetzt werden.
- A kann versuchen, Annahmen über B treffen; das ist (zumindest falls A und B Menschen sind) im Prinzip ein kompliziertes spieltheoretisches und psychologisches Problem. Eine Annahme kann A aber sicher treffen, ohne dass B sein Wissen darum besonders gut ausnutzen kann: B ist nicht in der Lage, nach wenigen Zahlen zu erkennen, welches demnächst die häufigste sein wird. Eine mögliche fiese Strategie wäre also: Wähle für einen Bereich der Liste eine Zahl, die mit erhöhter Häufigkeit vorkommen soll. Bei der zweiten Abarbeitung kann sie nach oben befördert werden und bringt nur einen Punkt für B, aber beim ersten Durchgang bemerkt B nicht gleich, welche Zahl das ist und kann sie darum auch nicht entsprechend behandeln, sodass sie einige Male mehr als einen Punkt bringt. B kann zwar versuchen, dieser Strategie zu begegnen, indem er die Häufigkeit der letzten paar Zahlen analysiert und dementsprechend die Häufigkeiten der nächsten Zahlen extrapoliert. Aber wenn die Unregelmäßigkeiten subtil genug versteckt sind, kommt er damit nicht weit; für ihn sieht die Liste zufällig aus, bringt aber weniger Punkte als eine solche, wenn A sie abarbeitet.

- Wenn B für den unbekannten Rest der Liste eine Gleichverteilung (oder irgendeine andere, von A vorhersehbare Verteilung) der Möglichkeiten annimmt, ist sein Verhalten vorhersehbar[3] und kann von A bei der Erstellung der Liste ausgenutzt werden, um für sich die maximale Punktzahl $n \cdot (\ell + 1)$ herauszuholen. Dabei ist zu bedenken, dass eine solche Liste vielleicht auch B einiges an Punkten einbringt. Andererseits ist dann die Annahme der Gleichverteilung falsch, und wenn B dahinterkommt, wird er *keine* Gleichverteilung mehr erwarten (ähnliches gilt für andere, von A vorhersehbare Verteilungen: A's eigene Entscheidungen, könnten sie von B vorhergesehen werden, würden A's Entscheidungen in gegensätzlicher Weise beeinflussen).
- Falls A die von B benutzte Strategie kennt, und falls diese deterministisch (vorhersehbar, ohne Zufallselemente) ist, kann sie die *grausame* Strategie benutzen. Diese simuliert laufend, welcher Block während der späteren Abarbeitung der Liste durch B ganz unten sein wird, und wählt immer ausgerechnet diesen als nächsten Block. A gewinnt bei dieser Strategie für jeden Listeneintrag genau ℓ Punkte.

B arbeitet die Liste ab

- Wenn B keinerlei Annahmen über den ihm unbekannten Rest der Liste machen kann, kann er auch keine Strategie anwenden, um die für A erreichbare Punktzahl zu verringern.
- Um Annahmen über die Liste zu machen, müsste B A's Strategie irgendwie durchschaut haben. A könnte ihn aber auch getäuscht haben und plötzlich eine unerwartete Strategie verwenden.
- A's Punktzahl zu minimieren stellt für B im allgemeinen ein schwieriges spieltheoretisches Problem mit psychologischer Komponente dar. Viele Anhaltspunkte gibt es nicht, weil A im Prinzip jede Strategie benutzt haben könnte und B nicht bekannt ist, bis zu welchem Grade seine Strategie durchschaut wurde.
- B hat also allgemein praktisch keinen zielgerichteten Einfluss auf A's Punktzahl.
- Was B tun kann, wenn der Stapel nach der zweiten Runde nicht neu geordnet wird: Da er den Anfang der Liste kennt, kann er versuchen, den Spielzustand nach Abarbeitung der Liste so aussehen zu lassen, das es zumindest am Anfang der zweiten Abarbeitung schwer für A ist, B's Punktzahl zu minimieren. Diese Idee verfolge ich nicht weiter, da, wie gesagt, die Auswirkungen gering wären und diese Variante der Spielregeln mir ohnehin nicht zusagt.

[3] Das stimmt nur, wenn B's Strategie deterministisch ist. Da B sich über den Rest der Liste sowieso nicht sicher ist, kann er auch seine Züge randomisieren, so dass im wesentlichen der gleiche Erwartungswert entsteht, aber A's Planung über den Haufen geworfen wird. A kann dann nur noch die Verteilung von B's Entscheidungen berücksichtigen.

- Man kann auch davon ausgehen, dass A's Repertoire an Strategien recht begrenzt ist, was bei einem programmierten Spieler A eine gerechtfertigte Annahme ist. Dann kann B das natürlich ausnutzen, indem er anhand der bisherigen Liste abschätzt, welche Strategie verfolgt wird, und entsprechende Erwartungen an den Rest der Liste stellt.
- Beim zweiten Durchgang hat es A relativ leicht, B's Punktzahl zu minimieren, da alles bekannt ist. Somit gibt es zu jeder Endkonfiguration des ersten Durchgangs eine minimale Punktzahl für B, die auch erreicht wird, wenn A optimal spielt. Der zweite Durchgang kann von B als deterministisch betrachtet werden, auch wenn er noch nicht die ganze Liste kennt. Wenn A von der optimalen Strategie abweicht, ist es ihr Pech. Wenn bereits i Zahlen der Liste im ersten Durchlauf abgearbeitet sind, kann B für jeden Endzustand der Liste berechnen, wie viele Punkte er im zweiten Durchgang, Schritt i, erwarten kann. Er kann also unter den möglichen Zügen den auswählen, der mit möglichst großer Sicherheit zu einem möglichst guten Endzustand führt. Natürlich muss er nach jeder neuen bekannten Zahl seinen Plan revidieren. Wenn B weiß, wann das Ende kommt, kann er besser berechnen, kann er seine Züge genauer bewerten, weil es s Züge vor Schluss sinnlos ist, einen Endzustand anzustreben, der mit mindestens $s+1$ Zügen erreicht werden muss.

Ein paar einfache Strategien für B sollen auch noch betrachtet werden – die im Rahmen einer Computersimulation des Spiels sinnvoll, weil einfach umzusetzen sind:

- Nichts tun: Es wird kein Block nach oben verschoben. Das ist offensichtlich keine besonders gute Strategie, da A dabei im asymptotischen Extremfall ℓ mal mehr Punkte als B bekommen kann.
- Nach vorne bringen (Move-To-Front, kurz: MTF): Der aktuelle Block wird immer ganz nach oben gebracht. Diese Strategie funktioniert recht gut; Listen, auf denen sie eine schlechte Punktzahl bringt, haben ebenfalls eine so hohe Minimalpunktzahl, dass das Punkteverhältnis $A/B = 2 - 2/(\ell + 1)$ nicht übersteigt.
- Zufällig nach vorne bringen (Random MTF): Der aktuelle Block wird nur mit einer gewissen Wahrscheinlichkeit ganz nach oben gebracht. Dadurch wird verhindert, dass B's Züge vorhersehbar werden und von A ausgenutzt werden.
- Vertauschen (Transpose): Der aktuelle Block wird immer mit dem darüber liegenden vertauscht, wenn möglich. Die häufigen Blöcke kommen dabei langsam nach oben.
- Zufällig (random): Der aktuelle Block wird um eine zufällige Distanz nach oben bewegt; die Verteilung der Möglichkeiten sei z. B. eine Gleichverteilung. Diese Methode funktioniert so ähnlich wie „Zufällig nach vorne bringen“. Der oben genauer beschriebene Algorithmus fällt in diese Kategorie, benutzt aber eine kompliziertere (und hoffentlich besser angepasste) Wahrscheinlichkeitsverteilung.
- BIT-Verfahren: der aktuelle Block kommt jedes zweite Mal ganz nach oben und bleibt jedes andere Mal, wo er ist. Beim ersten Zug wird eine dieser Varianten zufällig gewählt.

Natürlich wurden noch viele weitere Strategien ausgedacht und untersucht.

A arbeitet die Liste ab

Für die zweite Abarbeitung durch A ist die minimale Punktzahl für B, wie bereits erwähnt, exakt berechenbar und die Zugfolge, um das zu erreichen, müsste eigentlich gar nicht von A ausgeführt werden, sondern kann in einem separaten Programmteil berechnet werden.

Der Algorithmus dafür ist aber ein Problem für sich. Eine exakte Lösung mittels dynamischer Programmierung berechnet für jede Position in der Liste (von hinten angefangen) und jede Permutation der Blöcke die minimal erreichbare Punktzahl, wobei jeweils die Summe aus der Stapelposition des aktuellen Listenelements und dem Minimum der bereits berechneten Ergebnisse über alle erreichbaren Permutationen bei der nachfolgenden Listenposition das Ergebnis ist. Die Gesamtlösung ist dann das Ergebnis für die erste Listenposition und die Startpermutation. Diese Lösung hat leider eine Zeitkomplexität von $O(n \cdot \ell^{\ell+1})$ und lässt sich nicht auf unsicheres Wissen verallgemeinern, um den Algorithmus für B zu erhalten.

Eine recht gute Heuristik basiert auf folgenden Überlegungen (sei x die Nummer des Blockes, der nach oben wandert, und y die Nummer des Blockes, der gerade zwischen den Fingern des Spielers durchrutscht):

- Wenn x das letzte Mal vorkommt, bringt es nichts, den Block nach oben zu bewegen; es könnte sogar schaden, da dabei andere Blöcke nach unten bewegt werden.
- Wenn y nicht mehr in der Liste vorkommt, sollte man x immer überholen lassen, da y ja keine Punkte mehr bringt. Dafür bringt x beim nächsten Mal einen Punkt weniger.
- Wenn zwischen der aktuellen Position und dem nächsten Vorkommen von x kein y in der Liste vorkommt, kann man x überholen lassen; dann bringt er beim nächsten Mal einen Punkt weniger, allerdings bringt y später einen Punkt mehr.
- Wenn zwischen der aktuellen Position und dem nächsten Vorkommen von x genau ein y in der Liste vorkommt, kann man x überholen lassen; dann bringt er beim nächsten mal einen Punkt weniger und y bringt später einen Punkt mehr – allerdings nur unter der Annahme, das y x nicht wieder überholt. Es muss also eine „Vorfahrtregelung“ existieren, um bei alternierenden Ketten der Form $x \ldots y \ldots x \ldots y \ldots x \ldots y$ zu entscheiden, welches das obere sein darf. Wenn das Muster der Kette von $y \ldots y$ oder $y \ldots$ [Ende] unterbrochen wird, hat y Vorfahrt; bei $x \ldots x$ oder $x \ldots$ [Ende] ist es x.
- Wenn zwischen der aktuellen Position und und dem nächsten x-Vorkommen y in der Liste $m > 1$ mal vorkommt, sollte man x nicht überholen lassen; da er von y ohnehin wieder überholt werden wird, bringt es beim nächsten x in der Liste keine Punktersparnis, und y bringt später einen Punkt mehr.
- Die Entscheidungen, die einen Punkt mehr bringen, müssen abgewogen werden gegen die, die einen Punkt weniger bringen. Das kann allerdings die Vorfahrtregelung durcheinanderbringen, was vermutlich dafür verantwortlich ist, dass die Punktzahl bei $\ell = 5$ und $n = 10.000$ für Zufallslisten um ca. ein Prozent größer ist als minimal möglich.

Dieses Verfahren hat eine Zeitkomplexität in $O(n \cdot \ell)$ und benutzt nur einfache Informationen über die Folge in seinen Entscheidungen. Diese Informationen können auch als Wahrscheinlichkeiten gegeben sein, was in jedem Schritt zu einer Wahrscheinlichkeitsverteilung über den möglichen Aktionen führt, die dann zur endgültigen Entscheidungsfindung benutzt werden kann. Auf diese Weise lässt sich also auch B's Strategie realisieren.

2.4 Ein fairer Wert für c

Allgemeine Überlegungen

Ein fairer Wert für c wäre einer, bei dem zwei gleich starke Spieler im Mittel gleich viele Punkte bekommen. Da die Aufgaben der Spieler hier so unterschiedlich sind, lassen sich ihre Spielstärken leider nicht wirklich vergleichen. Auf jeden Fall kann man nach vernünftigen Grenzen für c (in Abhängigkeit von ℓ) suchen.

Bei der Verwendung von Listen der Länge n mit gleich verteilten Zufallszahlen zwischen 1 und ℓ (das entspricht der einfachsten vernünftigen A-Implementierung) ist A's erwartungsgemäße Punktzahl (a) maximal und liegt bei $n \cdot (\ell+1)/2$. B's Punktzahl (b) ist dann auch maximal und kann ermittelt werden, indem man den optimalen Algorithmus auf mehrere Listen anwendet und den Durchschnitt bildet.

Wenn man jetzt Listen betrachtet, die nur ein kleines bisschen unzufällig sind, sollte sich das auf a nicht sehr stark auswirken, da B dann die Regelmäßigkeiten nicht bewusst ausnutzen kann, im Gegensatz zu A. Daher sollte das Verhältnis a/b in Abhängigkeit von der Zufälligkeit am Anfang ansteigen, weil b schneller kleiner wird als a. a/b für zufällige Listen ist somit eine Untergrenze für c, da bei kleineren Werten für c auch bei nicht ganz zufälligen Listen a immer größer wäre als bc (auf Erwartungswerte bezogen, natürlich; B könnte ja auch mal einen Glückstreffer haben). Für ganz regelmäßige Listen (z.B. $(3,3,3,3,\ldots)$) strebt a/b offensichtlich gegen eins (optimales Verhalten von B vorausgesetzt). Solche Listen zu erstellen gehört bestraft, darum ist es gerechtfertigt, diese Untergrenze anzunehmen, auch wenn kleinere Werte für a/b durchaus auftreten können.

Experimente

Die Kurve „a/b in Abhängigkeit von der (geeignet quantifizierten) Zufälligkeit der Liste“ startet also am zufälligen Ende bei $c_{min} > 1$, steigt an und hat irgendwo ein Maximum c_{max}, um sich dann an 1 anzunähern. (Siehe Abbildung 2.1. Abbildung 2.2 zeigt, dass dieses Verhalten nicht für jede Kombination von Listenerzeugungsalgorithmus und Online-Algorithmus auftritt.[4] Das heißt wohl, dass mein Listenerzeuger zu einfach zu durchschauen ist.) Eine geeignete Quantifizierung für Zufälligkeit ist z.B. die erwartete minimal mögliche Punktzahl, da

[4] Sowohl BIT als auch RMTF bewegen jeden Block im Mittel jedes zweite mal nach oben. Trotzdem so ein Unterschied!

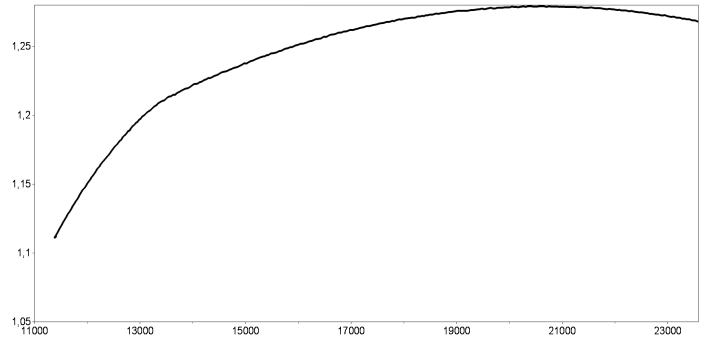

Abbildung 2.1: BIT-Strategie
X-Achse: Punktzahl von A; Y-Achse: Verhältnis der Punktzahlen a/b.
$n = 10.000$, $\ell = 5$, Datenpunkte gemittelt über jeweils 1000 Versuche.
Listenerzeugungsalgorithmus: Lokale Abweichung unterschiedlicher Stärke von der Gleichverteilung für eine Dauer von 5 bis 15 Listenelementen.
Online-Algorithmus: BIT; Offline-Algorithmus: Heuristik.

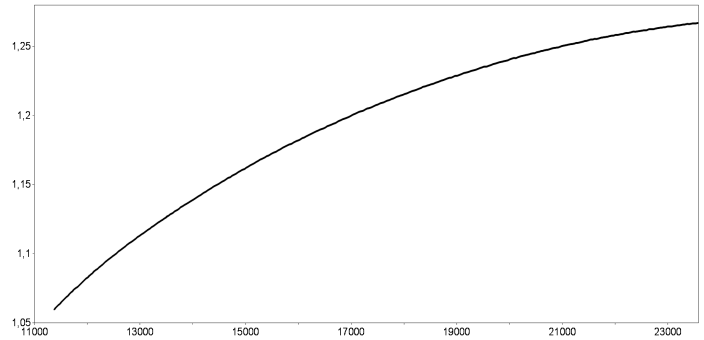

Abbildung 2.2: RMTF-Strategie
Online-Algorithmus: Random Move-to-front mit $p = 0,5$.
Rest wie oben.

diese für wachsende Zufälligkeit monoton steigt. c_{max} ist experimentell durch Variation der Zufälligkeitsparameter im Listenerzeugungsalgorithmus[5] zu ermitteln. Es ist die Obergrenze für c (im Kontext der verwendeten Algorithmen), weil bei größeren Werten A keine Liste erstellen könnte, die die Wahrscheinlichkeit eines Sieges von B auf einen fairen Wert bringt. Unter der Annahme, dass A nichts über B's Strategie weiß und dass $\ell = 5$, ergeben meine Simulationen in etwa $c_{min} = 1,2675$ und $c_{max} = 1,271$.

Wo aber liegt c in diesem Intervall? Zu nah an einem der Endpunkte wäre fast genau so unfair wie genau auf einem Endpunkt. Also mehr zur Mitte hin. Welche Mitte? Neben $\frac{c_{max}+c_{min}}{2}$ käme auch noch der a/b-Wert mit demjenigen b (also derjenigen Listenzufälligkeit) in Frage, das auf halbem Weg zwischen den bs im Nenner von c_{min} und c_{max} liegt.

Eine andere Möglichkeit basiert auf der Annahme, dass ein experimentell ermitteltes c_{max} deutlich unter dem tatsächlich möglichen c_{max} liegt und hauptsächlich von den Anstrengungen des Programmierers abhängt. Unter der Annahme, dass man sich bei den Algorithmen für A und B gleichmäßig angestrengt hat, sind sie als gleich stark (zumindest aufeinander bezogen), aber nicht perfekt anzusehen. Wenn man sie oft genug gegeneinander antreten lässt, sollte $\sum_{i=0}^{n} \frac{a_{\text{Runde i}}}{b_{\text{Runde i}}}$ einen fairen c-Wert ergeben. Da ich in den Algorithmus zur Erzeugung der Listen durch A ungleich viel weniger Aufwand investiert habe, lässt sich das Verfahren auf meine Algorithmen eigentlich nicht anwenden, ergibt aber $c \approx 1,2704$.

Berechnungen

Zur Ermittlung der Grenzen für c kann man auch Berechnungen anstellen. Betrachten wir den Fall, das A B's Strategie kennt. A wählt am Anfang ihrer Abarbeitung der Liste eine zufällige unter den möglichen Blockreihenfolgen, jede davon mit Wahrscheinlichkeit $1/\ell!$. Während ihrer Bearbeitung hält sie die bereits wenigstens einmal ausgerufenen Blöcke immer oberhalb der anderen Blöcke und in ihrer gewählten Reihenfolge, eingeschränkt auf die ausgerufenen Blöcke. Nehmen wir zuerst vereinfachend an, dass die Blockreihenfolge von Anfang an mit der von ihr gewählten Reihenfolge übereinstimmt. Dann ist die Reihenfolge also vollkommen zufällig. Daraus folgt, dass B bei jeder Abarbeitung eines Listeneintrags eine erwartete Punkteanzahl von genau $(\ell+1)/2$ gewinnt („im Mittel ist der Block in der Mitte"). Man sieht, dass A (auf jeden Fall) genau $2\ell/(\ell+1) = 2-2/(\ell+1)$ mal so viele Punkte wie B's erwartete Punkteanzahl gewinnt, was mit $c = 2-2/(\ell+1)$ in Verbindung gebracht werden kann.

A muss in Wirklichkeit keine zufällige Reihenfolge wählen. Sie berechnet und wählt eine für sie günstigste Reihenfolge, die jedenfalls nicht schlechter als eine zufällige Reihenfolge ist. Und die Zusatzpunkte, die sie verlieren kann, weil die Blockreihenfolge am Anfang noch nicht mit der von ihr gewählten Reihenfolge übereinstimmt, sind höchstens $\binom{\ell}{2}$ – das hängt damit zusammen, dass eine Permutation von $1,\ldots,\ell$ nicht mehr als $\binom{\ell}{2}$ sogenannte Inversionen hat. $\binom{\ell}{2}$ Punkte würden wir typisch als vernachlässigbar betrachten, weil es im Spiel bei einer hinreichend langen Liste um viel mehr Punkte geht.

[5] Das echte c_{max} wird man damit nicht bekommen, da wohl kaum jemand den ultimativen Listenerzeugungsalgorithmus und den optimalen Online-List-Update-Algorithmus gefunden haben wird.

Die Konklusion aus diesen ganzen Betrachtungen ist: Falls A B's Strategie kennt und diese deterministisch ist, dann kann sie immer gewinnen, falls $c < 2 - 2/(\ell+1)$, zumindest falls sie wählen darf, die Liste beliebig lang zu machen. Analog kann man zeigen, dass A immer gewinnen kann, falls $c < 3/2 - 2/(\ell+3)$ und A B's Strategie kennt, selbst wenn B's Strategie Zufallselemente enthält. Für $\ell = 5$ liegt die Grenze also bei $5/4 = 1,25$, der experimentell ermittelte Wert $1,27$ ist also zumindest nicht unfair.

2.5 Bewertungskriterien

Teilaufgabe 1 Die ersten drei Lücken sollten erkannt werden; zu überlegen, ob das Spiel mehrere Runden hat, ist meines Erachtens nicht nötig. Wie die Lücken gefüllt werden, ist relativ unmaßgeblich, solange die Festlegungen vernünftig und gut begründet sind und ihrerseits keine Lücken mehr lassen, also vollständig sind. Dass das Spiel uninteressant wird, wenn die Blockanzahl in der Größenordnung der Listenlänge ist, sollte bemerkt werden.

Teilaufgabe 2 Die Programmteile müssen implementiert und auf abstraktem Niveau beschrieben sein, entsprechend funktionieren und dokumentiert sein. Die Beschreibung muss deutlich machen, welche Programmkomponente welche Funktionalität implementiert. Die Aufteilung der Funktionalität auf verschiedene Komponenten muss sinnvoll sein. Es muss nicht unbedingt eine selbstständige Spielleiter-Komponente geben, aber Blockstapel, Liste und Punktzahlen der Spieler sollten zumindest unabhängig von den Spieler-Komponenten angelegt sein. Ein modularer Aufbau, der beliebige Zusammenstellungen von Listen(erzeugern), Online-Listenabarbeitern und Offline-Listenabarbeitern erlaubt, ist gut und bringt Pluspunkte. Zusätzliche interessante Ausgaben bringen ebenfalls Pluspunkte, insbesondere, wenn sie übersichtlich präsentiert werden. Eine gute Heuristik für den Offline-Algorithmus ist in Ordnung, da kein in n und ℓ zeitlinearer optimaler Algorithmus bekannt ist und die Verwendung eines optimalen Algorithmus entweder unzumutbar schwierig oder zu langsam ist. Die Teilnehmer sollten sich aber dessen bewusst sein und eine Heuristik nicht für optimal halten. Ausprobieren verschiedener Strategien, insbesondere für B, ist gut. Randomisierte Strategien sollten dabei sein, da sonst A die grausame Strategie anwenden könnte.

Teilaufgabe 3 Es sollte erkannt werden, dass die Grenzen für c von ℓ abhängen. Es sollten eigenständig Kriterien für einen fairen c-Wert formuliert werden. Wie die Aufgabenstellung schon sagt, sollen die Experimente zur Ermittlung von c motiviert, beschrieben, durchgeführt und dokumentiert werden. Die Experimente sollten geeignet sein, eine untere und obere Schranke für c zu finden. Es sollte erkannt werden, dass eine experimentbasierte Wahl von c von den Spieler(n/algorithmen) abhängt, deren Intelligenz sich nicht vergleichen lässt, weil sie unterschiedliche Aufgaben ausführen. Auch werden sich Menschen wahrscheinlich anders verhalten als die Algorithmen. Unter diesen Gesichtspunkten sollte eine konkrete Wahl für c getroffen werden.

Aufgabe 3: Geheimnisvolle Töne

SETI@home-Freunde werden sich freuen: Kürzlich wurde ein Tonsignal aufgezeichnet, welches zu viele Regelmäßigkeiten besitzt, als dass es natürlichen Ursprungs sein könnte. Dummerweise wird vermutet, dass es sich doch nicht um eine außerirdische Quelle handelt. Dies soll anhand der zur Verfügung gestellten WAVE-Datei nachgewiesen werden, indem die enthaltene Nachricht entschlüsselt wird.

3.1 Detektivische Untersuchungen

Also ran an die Arbeit, hören wir uns die Datei mal an: Man hört kurze, schnell aufeinander folgende Pieptöne, die durch immer stärker werdendes Rauschen überdeckt werden, welches anfangs fast gar nicht, am Ende ziemlich stark vorhanden ist und zu einer Art metallischen Kratzens wird, auch wenn die Töne immer hörbar bleiben. Schnelle Zähler können vielleicht sogar die Anordnung der Töne in 8er-Blöcke heraushören. Allzu viel Information kann man durch bloßes Hören aber nicht gewinnen, da alle Töne gleich klingen.

Schauen wir ein wenig genauer hin: Mittels eines Programms wie des freien Sound-Editors Audacity kann man sich die Datei näher betrachten, nämlich zu jedem der diskreten Zeitschritte die Auslenkung des Tones anschauen. Zoomt man ein wenig hinein, so offenbart sich klar die Anordnung von Tönen zu 8er-Gruppen (Abbildung 3.1). Auch sieht man gut, wie das Grundrauschen zum Ende hin zunimmt (Abbildung 3.2). Schaut man sich nun die einzelnen Töne genauer an, so stellt man fest, dass sich alle recht ähneln, bei noch genauerer Betrachtung aber, dass es zwei Sorten Töne gibt: Jeder Ton ist eine Art Sinusschwingung, bei der zwischen Maximum und Minimum noch ein weiterer Schwung vorkommt. Dadurch rückt das (globale) Minimum der Kurve näher zu seinem linken bzw. rechten rahmenden Maximum für die zwei Arten von Tönen. Abbildung 3.3 zeigt diese beiden Töne ohne viel Rauschen und Abbildung 3.4 die zwei wahrscheinlich zur Erzeugung genutzten Funktionen. Erstaunlich ist, dass sich beide Töne aber gleich anhören.

Schaut man an das Ende des Signals, so sind die Tonsorten dort noch erkennbar, aber stark durch Rauschen gestört. Es sieht sehr danach aus, dass dieses Rauschen additiv durch eine im Intervall $[-R,R]$ gleich verteilte unabhängige Zufallsvariable erzeugt wird, wobei R innerhalb des Tonsignals linear ansteigt (Abbildung 3.2). Dies ist der Grund, warum es schwer ist, die letzten Töne automatisch nach ihrer Tonsorte zu unterscheiden.

3.2 Rauschen beheben

Eine Möglichkeit, dieses Problem zu umgehen, ist, das Rauschen so gut wie möglich zu entfernen. In der Signalverarbeitung sind hierzu so genannte lowpass-Filter üblich, die hochfrequente Signale wie unser Rauschen entfernen und nur die niedriger frequenten Teile, unsere Töne also, übrig lassen. Ohne zu sehr auf die Details eingehen zu wollen, will ich den einfachsten lowpass-Filter beschreiben:

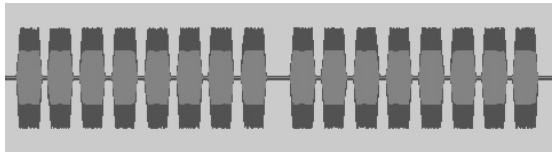

Abbildung 3.1: Ausschnitt aus der Original-WAVE-Datei, in dem die Einteilung in 8er-Blöcke zu sehen ist.

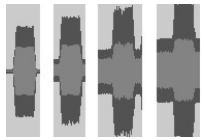

Abbildung 3.2: Vier Ausschnitte aus der Original-WAVE-Datei, die jeweils einen Ton etwa am Anfang, bei 1/3, bei 2/3 und am Ende des Signals zeigen. Man sieht deutlich die Zunahme des Grundrauschens.

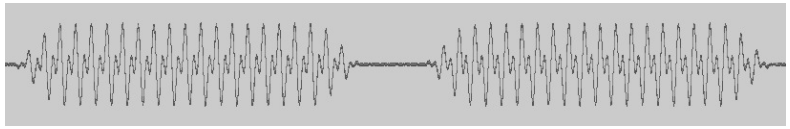

Abbildung 3.3: Ausschnitt aus der Original-WAVE-Datei, der die zwei Sorten gleich klingender Töne zeigt. Einmal sind die Minima links-geneigt, einmal rechts-geneigt.

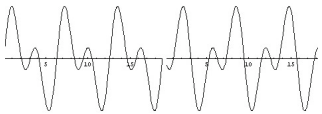

Abbildung 3.4: Die Funktionen $\sin(x)+\sin(2x)$ und $\sin(x)-\sin(2x)$ im Bereich 0 bis 20.

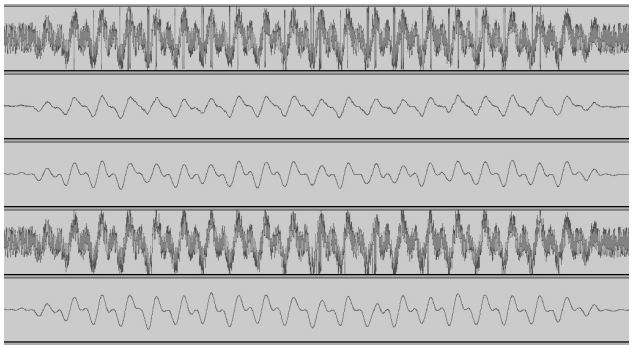

Abbildung 3.5: Ausschnitt nahe am Ende des Signals, im Original und nach Überarbeitungen. Die Wellenlänge des Tones kann man als etwa 200 ablesen. Die einzelnen Zeilen: (1) Original-Signal. (2) einfaches Box-Filtering, $m = 35$. (3) 3-fach iteriertes Box-Filtering mit $m = 15$. (4) Original-Signal mit entfernten Sprüngen, die meisten der vertikalen Linien im Original (Sprünge) sind verschwunden. (5) 3-fach iteriertes Box-Filtering nach Entfernen der Sprünge, $m = 15$.

Der schon fast naive Ansatz des Box-Filters bedeutet, dass wir die Auslenkung an der Stelle i ersetzen durch die mittlere Auslenkung im Intervall $[i-m, i+m]$ für eine Konstante m. Damit mitteln wir das Rauschen; also entfernen wir es, falls m groß genug ist, da das Rauschen ja aus einer Zufallsvariablen mit Mittel 0 entsprungen ist. Allerdings dürfen wir m nicht zu groß wählen, da wir sonst auch die Töne zerstören: Haben wir zwei ganze Wellenlängen in unserem Intervall $[i-m, i+m]$, so wird jede Auslenkung zu 0 gemittelt, abgesehen von etwas zurückbleibendem Rauschen. Auch für kleine m sind diese Auswirkungen schon anhand der verringerten maximalen Amplitude zu sehen (Abbildung 3.5.2).

Bessere Ergebnisse als mit dem bloßen Box-Filter erhält man, indem man diesen Filter mehrmals iteriert. Auf diese Weise approximiert man nämlich einen so genannten Gaußschen Filter, der als einer der besten angesehen wird. Abbildung 3.5.3 zeigt einen der letzten Töne des Signals nach 3-maligem Box-Filtering.

Man kann ein noch schöneres Signal herausholen, wenn man das im ersten Schritt gehörte metallische Kratzen zum Ende des Signals ernst nimmt. Es resultiert wahrscheinlich daraus, dass mit Zahlen im Bereich -32767 bis 32768 gerechnet wurde bei der Erstellung des Signals: Hat man eine hohe Auslenkung des Tones und addiert eine große Auslenkung aufgrund des Rauschens dazu, so kann man schnell größer als 32768 werden und so im stark negativen Bereich landen. Diesen Effekt hört man tatsächlich.

Entfernen lässt er sich recht gut, indem man zu jeder Auslenkung die nachfolgende Auslenkung um +65536 bzw. -65536 adjustiert, falls die absolute Differenz beider Auslenkungen größer als 32767 ist. Damit erhält man Amplituden, die betragsmäßig größer als 32768 sind, also die der Erzeugung nach wahren Auslenkungen. Wendet man hiernach das iterierte Box-Filtering an, so erhält man das schönere Signal in Abbildung 3.5.4.

Natürlich sind viele andere Arten denkbar, das Rauschen zu umgehen, die hier beschriebene ist aber eine der einfachsten.

3.3 Bits auslesen

Nun haben wir schon ein sehr schönes Signal, aus dem wir fast sofort die enthaltenen „Bits“ auslesen können, also zu jedem Ton feststellen können, ob er rechts oder links geneigt ist. Dazu müssen wir die Wellenberge und -täler entdecken (und zwar diejenigen, die zu den großen „globalen“ Maxima, nicht zu den kleinen lokalen gehören), sie zu Bits zuordnen und deren Geneigtheit auslesen. Dies kann mit relativ einfachen Mitteln geschehen, wobei wir allerdings noch immer mit ein wenig Restrauschen zu kämpfen haben:

Wir legen fest, dass ein in einer kleinen Nachbarschaft (etwa 10-60 Zeitpunkte) lokales Maximum zu einem globalen Maximum gehört, wenn seine Auslenkung wenigstens die Hälfte der maximalen Auslenkung im gesamten Signal beträgt. Auf die gleiche Art finden wir die interessanten Minima. Nun können wir zwei aufeinanderfolgende Maxima als zum gleichen Ton gehörend klassifizieren z.B. indem ihre Distanz höchstens doppelt so groß ist wie die durchschnittliche Distanz zwischen zwei aufeinanderfolgenden Maxima im Signal. Ein Ton ist hiernach links geneigt, wenn zwischen mehr als der Hälfte seiner zugehörigen Maxima das dazwischenliegende Minimum näher beim linken rahmenden Maximum liegt als beim rechten. Puh!

Die dabei auftretenden Konstanten sind natürlich frei gewählt, auch sind viel ästhetischere Möglichkeiten des Bit-Auslesens denkbar. Aber es funktioniert auch so. Hier hat man sogar ein wenig Fehlerkorrektur enthalten: Ein Ton besteht aus etwa 15-20 Wellenbergen. Nutzen wir das Maximumsprinzip, so können wir bis zu 7-9 der Neigungen zwischen zwei Wellenbergen falsch auslesen, ohne dass unser Resultat verfälscht wird.

3.4 Zeichenfolge gewinnen

Hat man nun die Geneigtheit der Töne ausgelesen, ergeben sich die in der Nachricht enthaltenen Bits – allerdings weiß man nicht, welche Neigung der 1 entspricht und welche der 0. Man liest also eine der folgenden 150 Byte langen Bit-Folgen:

```
10111011 10010110 10011010 10001100 11011111 ...
```

oder

```
01000100 01101001 01100101 01110011 00100000 ...
```


Da wir einen menschlichen Ursprung nachweisen wollen, sind für uns natürlich vor allem in ASCII kodierte Textnachrichten interessant. Also wählen wir diejenige der beiden Alternativen, bei der die Nachricht möglichst viele Buchstaben enthält; die Tatsache, dass ASCII-Codes von Buchstaben im ersten Bit immer eine 0 haben, ist da sehr hilfreich. Letztlich wird uns folgende Nachricht enthüllt, die sich als Limerick präsentiert und deren Inhalt auf wundersame Weise zur Aufgabe passt:

```
Dies ist ein geheimes Gedicht,
bekannt werden will's niemals nicht.
Doch bist du ganz schlau
und auch noch genau,
bringst du seinen Inhalt ans Licht/
```

Wichtig sind hierbei die letzten beiden Zeichen, die korrekt entschlüsselt **Schrägstrich, Zeilenumbruch** sein sollten.

3.5 Andere Nachrichten dieser Form

Die Aufgabenstellung verlangt, dass auch andere Nachrichten „dieser Form“ entschlüsselt werden können sollen. Das heißt, dass man erst einmal dazu in der Lage sein muss, solche Nachrichten zu generieren. Damit liegt es nahe, ein eigenes Kodierungsprogramm zu entwickeln, das Textnachrichten nach dem erkannten Prinzip verschlüsselt. Die dafür genutzte Methode kann auf die gemachten Entdeckungen zum Ursprung von Rauschen und die beiden schon beiläufig erwähnten Formeln $\sin(x)+\sin(2x)$, $\sin(x)-\sin(2x)$ für die Töne zurückgreifen, aber natürlich auch geringfügig davon abweichende nutzen. Idealerweise wird ein Kodierer mit vielen Einstellmöglichkeiten realisiert, um das Entschlüsselungsprogramm mit vielen Varianten herausfordern zu können. Da nur recht wenige Lösungen auch einen solchen Kodierer enthalten, wird dieser als Erweiterung belohnt. Ohne Kodierer kann man (zu Testzwecken) an andere Nachrichten kommen, indem man z.B. die Beispieldatei mit Hilfe eines Audio-Editors manipuliert. Auf jeden Fall muss die Leistungsfähigkeit der Lösung in Bezug auf andere Nachrichten mit genügend Beispielen deutlich gemacht werden.

Auch ist es erstrebenswert, das Programm so „stabil“ wie möglich zu machen: Es sollte auch Nachrichten entschlüsseln können, bei denen sich die Abstände zwischen den einzelnen Tönen bzw. zwischen den 8er-Blöcken oder auch die Amplituden und Frequenzen der Töne von der ursprünglichen Nachricht unterscheiden. Hier sollte man vor allem von zu vielen Konstanten im Quelltext unabhängig sein und vielleicht möglichst viele dieser aus dem gegebenen Signal ableiten, so dass unterschiedliche Nachrichten kein Eingreifen des Nutzers erfordern.

Denkbar ist, dass bei der Kodierung andere Funktionen als $\sin(x)\pm\sin(2x)$ für die Töne verwendet wurden: Ausgesprochen gut wäre es also, wenn das Programm ohne Kenntnis der Funktionen trotzdem zwei verschiedene Tonsorten unterscheiden könnte, auch wenn diese stark von den ursprünglichen abweichen. Hier sind, wie man sieht, vielerlei Erweiterungen denkbar.

3.6 Bewertungskriterien

Beschreibung des Vorgehens Das detektivische Vorgehen bei der Analyse des Signals muss beschrieben sein. Am Ende sollte das Kodierungsprinzip erkannt worden sein.

Erklärung aller Schritte Die Schritte, die zur Entschlüsselung der Nachricht führen, müssen ausreichend erklärt und begründet werden.

Entschlüsselung Die zur Entschlüsselung angewandten Techniken sollten angemessen sein. Wie die Aufgabenstellung schon andeutet, sind Verfahren der Signalverarbeitung nicht erforderlich. Gegen ihre angemessene Verwendung, insbesondere über Büchereifunktionen, ist aber auch nichts einzuwenden.

Korrektes Ergebnis Die Nachricht muss (möglichst inklusive der letzten beiden Zeichen) vollständig und korrekt entschlüsselt werden. Ist dies nicht der Fall, weist dies auf Schwächen des Entschlüsselungsverfahrens hin.

Automatischer Programmablauf Das Programm muss die gegebene Nachricht eigenständig entschlüsseln, ohne dass der Nutzer helfen oder eingreifen muss, indem er beispielsweise die Tongrenzen von Hand festlegen muss, die entschlüsselte Nachricht aus mehreren Varianten auswählen muss, etc.

Andere Nachrichten dieser Form Das Programm muss grundsätzlich in der Lage sein, auch andere Nachrichten zu entschlüsseln. Die Entschlüsselungsfähigkeiten des Programms müssen an Beispielen deutlich gemacht werden. Ein eigenes Kodierungsprogramm zu deren Erstellung wird als Erweiterung anerkannt.

Überlegungen zu Korrektheit und Leistungsfähigkeit Die letzten Zeichen der entschlüsselten Beispielnachricht sind überraschend. Hier helfen Überlegungen zur Korrektheit der angewandten Verfahren. Sowohl Korrektheit als auch die Leistungsfähigkeit in Bezug auf die Entschlüsselung anderer Nachrichten sollten erläutert und begründet sein.

Stabilität des Programms Es sollten möglichst viele, auch von der gegebenen Form abweichende Nachrichten entschlüsselt werden können, ohne dass Konstanten geändert werden müssen oder der Benutzer eingreifen muss. Bei hoher, begründeter Stabilität kann es Zusatzpunkte geben.

Perlen der Informatik – aus den Einsendungen

Allgemeines Welcher Kuchen schmeckt Informatikern am besten? Der Googlehupf!

Da die Aufgabe sehr problematisch umgesetzt wurde, sollte man prinzipiell auch davon ausgehen können, dass ein Programm, welches für kleine Eingabedaten korrekte Ergebnisse liefert, dies auch für große tun wird.

Da das Programm relativ umfangreich ist, wollte ich mich nicht unnötig damit aufhalten, Spezifikationen … herauszufinden.

Es ist logisch, dass der Wert der Konstanten immer größer wird …

Rekrutiert die Stringpräsentation einer Referenten. *Kommentar zur Methode toString()*

Da ich Java mittlerweile satt habe, habe ich dieses Programm in Visual C++ geschrieben (war wohl nicht so DIE Idee).

Aufgabe 1 Kein normaler Mensch will dieselbe Veranstaltung mehrmals besuchen.

Keiner soll gezwungen sein eine Veranstaltung zu besuchen, wenn diese Veranstaltung nicht auf seiner Wunschliste steht (um z.B. die Mindestanzahl der Teilnehmer zu erlangen, damit sich andere freuen können).

Wer will denn schon seine Zeit für etwas verschwenden, womit er negativ zufrieden ist?

Aufgabe 2 Ein letzter unwichtiger Vorteil ist, dass gerade die Zahlen 7 & 13 für ein Spiel, das Blocksberg heißt und von Leuten, die auf Namen wie Bibi hören, gespielt wird, auf Grund der Symbolik geeignet sind.

Denn ein *hohes C* deutet auf einen Vorteil für Bibi hin, …

Die Klasseninstanzen Bibi und Ben werden durch prozedurale Variablen handlungsfähig.

Bibis hübsche Nase

Um solche Ideen zu sammeln, kann das Spiel aus Holz nachgebastelt und durch das Zwingen der Familienmitglieder durchgeführt werden.

Wenn ein Stein zum wiederholten Male aufgerufen wird, so wird er über alle Steine direkt über ihm gelegt, welche vor dem Stein selbst zum ersten Mal aufgerufen wurden, und unter den am weitesten unten liegenden Stein, der noch nicht aufgerufen wurde oder nach dem Stein selbst zum ersten Mal aufgerufen wurde.

Aufgabe 3 Diese Aufgabe finde ich jetzt im Nachhinein zu vage formuliert. Ich habe nämlich erst sehr lange nach der Art der Verschlüsselung suchen müssen.

Eine Frequenzanalyse der Datei mit dem Windows Media Player (Visualisierungen „Meeresdunst“ und „Feuersturm“) hat ergeben …

Slow-Fourier-Transformation, weil die Fast-Fourier-Brainfucking-Transformation mir zu komplex ist.

… machen nur Dualzahlen als Informationen echten Sinn (zumindest für Menschen).