36. Bundeswettbewerb Informatik 2017/2018, 1. Runde

Lösungshinweise und Bewertungskriterien

Bundeswettbewerb Informatik

Allgemeines

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

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

Dokumentation

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

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

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

Zu einer Einsendung gehören als zweiter Teil der Implementierung *Programme*, die möglichst eigenständig lauffähig sind. Für die gängigsten Skript-Sprachen stehen Interpreter zur Verfügung. Einige Entwicklungsumgebungen (z. B. BlueJ), bei denen die erstellten Programme ohne

Weiteres nur in der Umgebung selbst laufen, stehen bei der Bewertung zur Verfügung, aber sicher nicht alle. Kompilierung von Quellcode ist während der Bewertung nicht möglich. Deshalb können Programme nur dann mit unterschiedlichen Eingaben getestet werden, wenn diese vom Programm eingelesen oder über eine (nicht zwingend grafische) Schnittstelle eingegeben werden können.

Auch die folgenden eher inhaltlichen Dinge sind zu beachten:

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

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

Bewertung

Nun einige Erläuterungen zur Bewertung:

- Pro Aufgabe werden maximal fünf Punkte vergeben. Zu jeder Aufgabe gibt es eine Reihe von Bewertungskriterien. Sie sind in der Bewertung, die man im PMS einsehen kann, aufgelistet. In der ersten Runde gibt es in der Regel nur Punktabzüge; deshalb sind die Kriterien meist negativ formuliert. Hat man bei einem Kriterium 0 Punkte, ist die Einsendung in Bezug auf dieses Kriterium einwandfrei. Wenn das Kriterium nicht erfüllt ist, gibt es in der Regel einen Punkt Abzug (-1), manchmal auch zwei. Wurde die Aufgabe nur ansatzweise oder insgesamt zu schwach bearbeitet, wird ein spezielles Kriterium angewandt, bei dem es bis zu fünf Punkte Abzug geben kann. Im schlechtesten Fall wird die Aufgabenbearbeitung mit null Punkten bewertet.
- Für die Gesamtpunktzahl sind die drei am besten bewerteten Aufgabenbearbeitungen maßgeblich. Es lassen sich also maximal 15 Punkte erreichen. Einen 1. Preis erreicht man mit 14 oder 15 Punkten, einen 2. Preis mit 12 oder 13 Punkten und einen 3. Preis mit 9 bis 11 Punkten. Mit einem 1. oder 2. Preis ist man für die zweite Runde qualifiziert.
- Die Einsendung wird für die 2. Runde des Jugendwettbewerb Informatik (JwInf) gewertet, wenn darin mindestens eine Junioraufgabe bearbeitet wurde und wenn die gesamte Einsendung oder die in der Einsendung bearbeiteten Junioraufgaben von einem für diese Runde qualifizierten JwInf-Teilnehmer stammen. Wenn solche JwInf-Teilnehmer sich „nur“ als Team-Mitgied an einer BwInf-Einsendung beteiligt haben, ohne erkennbar mindestens eine Junioraufgabe eigenständig bearbeitet zu haben, erhalten sie zumindest in diesem Jahr eine Teilnahmeurkunde auch für die zweite JwInf-Runde, aber ohne Wertung. JwInf-Teilnehmer, die sich nicht für die 2. JwInf-Runde qualifiziert haben, können in dieser natürlich auch nicht gewertet werden.

- In der zweiten JwInf-Runde wird ein 1. Preis für 9 oder 10 Punkte, ein 2. Preis für 7 oder 8 Punkte und ein 3. Preis für 5 oder 6 Punkte vergeben.
- Leider wurde in einigen Fällen die Regelung zur Bearbeitung von Junioraufgaben als Teil einer BwInf-Einsendung nicht beachtet. Zitat aus dem Mantelbogen des Aufgabenblatts: „Junioraufgaben dürfen nur von Schülerinnen und Schülern bis einschließlich Jahrgangsstufe 10 (im G8: Einführungsphase) bearbeitet werden.“ Nur wenn ein Team mindestens ein Mitglied hat, das Junioraufgaben bearbeiten darf, können Bearbeitungen von Junioraufgaben im BwInf gewertet werden.
- Grundsätzlich kann die Dokumentation als „schlecht / nicht vollständig“ bewertet werden, wenn Teile wie Umsetzung, Beispiele oder Quellcode(auszüge) fehlen. Außerdem wird in der Regel gefordert, dass die gewählten Verfahren gut nachvollziehbar beschrieben sind und dass begründet wird, warum sie das gegebene Problem lösen.
- Leider ließ sich nicht verhindern, dass etliche Teilnehmende nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden. Das ist ziemlich riskant, da Fehler sich leicht einschleichen.
- Es ist verständlich, wenn jemand, der nicht weitergekommen ist, über eine Reklamation nachdenkt. Kritische Fälle, insbesondere die mit 11 Punkten, haben wir allerdings schon sehr gründlich und mit viel Wohlwollen geprüft.

Danksagung

An der Erstellung der Lösungsideen haben mitgewirkt: Melanie Schmidt (Junioraufgaben 1 und 2), Robert Czechowski (Junioraufgaben 1 und 2, Aufgabe 2), Felix Frei (Aufgabe 1), Karl Schrader (Aufgabe 3), Alexis Engelke (Aufgabe 4) und Nikolai Wyderka (Aufgabe 5).

Die Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerb Informatik entwickelt, und zwar aus Vorschlägen von Melanie Schmidt (Junioraufgaben 1 und 2, Aufgabe 4), Thomas Kesselheim (Aufgabe 1), Torben Hagerup (Aufgaben 1 und 5) und Holger Schlingloff (Aufgaben 2 und 3).

J1 Bücherregal

J1.1 Lösungsidee

Bei dieser Aufgabe soll eine Menge von Büchern in Abschnitte aufgeteilt werden, in denen die Bücher etwa gleich groß sind. „Etwa gleich groß“ bedeutet im Sinne der Aufgabe, dass keine zwei Bücher in einem Abschnitt einen größeren Höhenunterschied haben dürfen als 3 Zentimeter.

Da nur eine bestimmte Anzahl an Deko-Figuren zur Verfügung steht, um die Abschnitte voneinander abzutrennen, ist die Anzahl der möglichen Abschnitte begrenzt: Es darf höchstens einen Abschnitt mehr geben als Deko-Figuren vorhanden sind; wenn D die Zahl der Deko-Figuren ist, darf es also höchstens $D+1$ Abschnitte geben. Die Frage ist nun, ob eine Einteilung in Abschnitte unter diesen Bedingungen möglich ist und, falls ja, wie. Dabei wollen wir erlauben, dass Abschnitte leer bleiben. In der Aufgabenstellung heißt es nämlich: „Wie breit die Abschnitte sind, ist ihr egal.“ Also sollte auch eine Breite von 0 akzeptabel sein.

Diese Annahme erlaubt eine besonders einfache Lösungsstrategie: Man wählt zunächst das kleinste Buch aus, das noch nicht im Regal steht, und fügt es mit allen Büchern in einem Abschnitt zusammen, die nicht mehr als 3 Zentimeter größer sind als dieses Buch. Eine derartige Strategie heißt in der Informatik auch „greedy“ (engl. für: gierig), weil sie sich gierig alle Bücher nimmt, die in den Abschnitt passen, ohne zu beachten, ob sie auch in einen anderen Abschnitt passen würden. Dennoch entsteht auf diese Weise eine Einteilung in möglichst wenige Abschnitte; die Strategie ist also optimal in Bezug auf die Anzahl der verwendeten Abschnitte. Das ist aber hilfreich: Nur wenn die Lösung verlässlich die kleinste Zahl von Abschnitten finden kann, kann sie sicher feststellen, ob eine passende Einteilung der Buchmenge evtl. nicht möglich ist.

Die Strategie nimmt allerdings in Kauf, dass weniger als $D+1$ Abschnitte entstehen und einige Deko-Figuren nicht als Trenner verwendet werden. Wer das so nicht haben möchte, kann diese Situation „reparieren“ und übrige Deko-Figuren einfach irgendwo mitten in je einen gefundenen Abschnitt platzieren. Da jeder gefundene Abschnitt die Höhenbedingung erfüllt, wird die Bedingung auch von jedem seiner Teile erfüllt.

J1.2 Umsetzung

Die oben beschrieben Strategie lässt sich auf verschiedene Weise in einem Programm umsetzen. Werden die Bücherhöhen in eine Liste geschrieben, kann man die Liste sortieren (in vielen Programmiersprachen gibt es eine Sortierfunktion) und hat das gesuchte kleinste Buch vorne in der Liste stehen. Die sortierte Liste kann dann einfach der Reihe nach durchgegangen werden, bis ein Buch mehr als 3cm höher ist als das erste. Dieses Buch ist dann das erste Buch eines neuen Abschnitts, und alle Bücher bis dahin gehören noch zum vorigen Abschnitt. Insgesamt muss man die sortierte Liste einmal durchlaufen.

Sortieren lässt sich eine Menge aber auch, indem man ihre Elemente nach und nach in eine Datenstruktur einfügt, die am Ende ihre Elemente sortiert ausgeben kann. Auch solche Datenstrukturen gibt es in vielen Programmiersprachen; in Java z. B. SortedSet bzw. TreeSet.

Beispiel	# 1	# 2	# 3	# 4	# 5	# 6
Gegeben	4 Figuren	2 Figuren	2 Figuren	4 Figuren	3 Figuren	4 Figuren
Benötigt	4 Figuren	2 Figuren	3 Figuren	4 Figuren	3 Figuren	5 Figuren
Ergebnis	Möglich	Möglich	*Unmöglich*	Möglich	Möglich	*Unmöglich*
Abschnitte	168 170 Figur 202 211 229 Figur 233 254 260 Figur 272 Figur 306 307	169 175 Figur 203 209 210 229 Figur 235	170 174 Figur 202 229 Figur 235 Figur 279 305	160 ⋮ 190 Figur 196 ⋮ 225 Figur 233 ⋮ 261 Figur 264 ⋮ 293 Figur 295 ⋮ 304	160 ⋮ 180 Figur 201 ⋮ 231 Figur 232 ⋮ 261 Figur 263 ⋮ 270	125 Figur 160 ⋮ 180 Figur 200 ⋮ 228 Figur 247 ⋮ 264 Figur 281 ⋮ 310 Figur 340 341 342

Tabelle 1: Ergebnisse zu den vorgegebenen Beispielen

J1.3 Alternative Lösungsideen

Man kommt zur Not auch ohne Sortieren zurecht. Dazu muss man allerdings für jeden Abschnitt zweimal die gesamte Buchmenge durchsuchen: einmal, um das kleinste Buch zu finden und noch einmal, um alle Bücher herauszusuchen, die nicht mehr als 3cm höher sind. Ist N die Anzahl der Bücher und (wie schon gesagt) D die Anzahl der Deko-Figuren, muss man also $2(D+1)$ mal maximal N Bücher anschauen. Allerdings braucht bei ersten Strategie das Sortieren auch seine Zeit.

J1.4 Beispiele

Die Ergebnisse, die unsere „gierige“ Strategie für die vorgegebenen Beispiele liefert, sind in der Table 1 zusammengefasst. Anders als in der Aufgabenstellung gesagt wurden die Buchgrößen in den Beispieldateien in Millimetern angegeben, nicht in Zentimetern.

Diese Ergebnisse müssen nicht exakt so reproduziert werden. Z. B. ist auch ein Sortieren in absteigender Reihenfolge möglich und ergibt dann z. T. andere Verteilungen. Auch die Reihenfolge der Abschnitte oder die Reihenfolge der Bücher innerhalb der Abschnitte spielen keine Rolle.

J1.5 Bewertungskriterien

- Der Lösungsansatz (und das Programm) erfüllen die genannten Bedingungen:
  - Der Höhenunterschied beträgt pro Abschnitt höchstens 3 cm, und
  - es werden nicht mehr Abschnitte erzeugt als durch Figuren getrennt werden können.
- Die Lösung muss die minimale Anzahl an Abschnitten finden können, um auch unmögliche Fälle sicher zu bestimmen.
- Das Programm gibt die Bücherhöhen der Abschnitte aus. Es genügt auch, für jeden Abschnitt die Höhen des kleinsten und des größten Buchs eines Abschnitts anzugeben.
- Die Lösung sollte mit einer Laufzeit von höchstens $\mathcal{O}(n^2)$ auskommen.
- Die vorgegebenen Beispiele sollten bearbeitet und die zugehörigen Ergebnisse dokumentiert sein.

J2 Wintervorrat

Algorithmus 1: Sichere Felder ermitteln

```
1  berechne_sichere_felder begin
2      for alle Felder F do
3          F.sichereMinuten ← 0
4          Markiere F als unsicher
5      end
6      for alle Zeitschritte t ∈ {1,...,720} do
7          for alle Felder F do
8              F.sichereMinuten ← F.sichereMinuten + 1
9          end
10         for alle Vögel V do
11             if V.startZeit ≤ t then
12                 F ← V.aktuellesFeld
13                 if F.sichereMinuten > 30 und F unsicher then
14                     Markiere F als sicher
15                     F.beginnErsterSichererZeitraum ← t − F.sichereMinuten + 1
16                 end
17                 else if F unsicher then
18                     F.sichereMinuten ← 0
19                 end
20                 bewege_vogel(V)
21             end
22         end
23     end
24     for alle Felder f do
25         if F.sichereMinuten = 720 then
26             Markiere F als absolut sicher
27         end
28     end
29 end
```

J2.1 Lösungsidee

Diese Aufgabe lässt sich über eine Zeitschritt-Simulation lösen. Dabei werden die einzelnen Minuten nacheinander als Zeitschritt simuliert. In jedem Zeitschritt werden die Vögel entsprechend bewegt und es wird für alle Felder überprüft, wie lange sie bereits nicht von Vögeln überflogen wurden.

Dazu kann man z. B. für jedes Feld einen „Sicherheits“-Zähler jeden Zeitschritt um eins erhöhen. Wenn Vogel darüberfliegt, wird der Zähler auf 0 zurückgesetzt. Dann gibt dieser Zähler immer an, für wie viele Minuten das entsprechende Feld zu diesem Zeitpunkt bereits nicht von einem Vogel überflogen wurden.

Algorithmus 2: Vogel bewegen

```
1  bewege_vogel(V) begin
2      if V.Richtung = „N“ then
3          V.x ← V.x + 1
4          if V.x = x_max then
5              V.Richtung ← „S“
6          end
7      end
8      if V.Richtung = „O“ then
9          V.y ← V.y + 1
10         if V.y = y_max then
11             V.Richtung ← „W“
12         end
13     end
14     if V.Richtung = „S“ then
15         V.x ← V.x − 1
16         if V.x = 1 then
17             V.Richtung ← „N“
18         end
19     end
20     if V.Richtung = „W“ then
21         V.y ← V.y − 1
22         if V.y = 1 then
23             V.Richtung ← „O“
24         end
25     end
26 end
```

Falls der Zähler 30 erreicht, ist $t-29$ bis t ein sicherer Zeitraum für das Feld. Falls der Zähler am Ende der Simulation $12 \cdot 60 = 720$ erreicht, ist das Feld absolut sicher.

Um am Ende der Simulation auch Informationen über die sicheren, aber nicht absolut sicheren Felder zu haben, müssen die Felder auch während der Simulation schon ggf. als sicher markiert werden. Dafür gibt es zwei Möglichkeiten:

- Entweder: Wenn der „Sicherheits“-Zähler des Feldes den Wert 31 erreicht, wurde das Feld in den 30 vorherigen Minuten nicht von einem Vogel überflogen. Dann ist es also sicher.
- Oder: Wenn ein Vogel über ein Feld fliegt, dessen „Sicherheits“-Zähler einen Wert größer oder gleich 31 hat, wissen wir ebenfalls, dass das Feld in den 30 vorherigen Minuten nicht von einem Vogel überflogen wurde. Es ist also auch sicher.

Wenn wir die Felder also entsprechend einer der beiden Möglichkeiten als sicher markieren, und uns gleichzeitig den Startzeitpunkt des sicheren Zeitraumes für das entsprechende Feld merken, können wir am Ende auch für alle sicheren Felder einen sicheren Zeitraum angeben.

J2.2 Mögliche Fehler

Aus der illustration im Aufgabenblatt lässt sich außerdem interpretieren, dass die Vögel, bevor sie losfliegen, sich außerhalb des Gebietes befinden – also vor dem losfliegen kein Feld überblicken, auch nicht ihr Startfeld. Das lässt sich unter umständen aber auch anders interpretieren.

Es könnten Simulationsparameter gewählt werden, die der Aufgabenstellung widersprechen. Dazu gehören unter anderem:

- Vögel fliegen nur einmal über das Feld
- andere Verständnisfehler des Tagesablaufs (z. B. deutlich falsche Taglänge, Flugrichtung ignoriert, . . .)

Diese Aufgabe ist besonders anfällig für *off by one*-Fehler. Mögliche Fehlerquellen sind: Zählen der Zeit ab Minute 0 oder 1, initialisieren der Felder mit 0 oder 1, verfrühtes / verspätetes losfliegen der Vögel.

Das Fiese ist, dass sich mehrere Fehlerquellen gegenseitig wieder aufheben, oder – noch schlimmer – teilweise wieder aufheben können. Daher können solche Fehler beim Beispiel auf dem Aufgabenblatt das richtige Ergebnis liefern, aber bei anderen Beispiele falsche Ergebnisse liefern. Die Reihenfolge der verschiedenen Update-Schritte spielt hierbei eine große Rolle!

Weil sich diese Fehler aber auch durch eine andere Interpretation der Aufgabe ergeben können, gibt es dafür in der Regel keine Abzüge – es sei denn, das Beispiel auf dem Aufgabenblatt kann nicht reproduziert werden.

Vertauschen der Nord-Süd-Richtung ist ein weiteres mögliches Problem. Da die zur Verfügung gestellte Programmierumgebung den Koordinaten-Ursprung nach links oben legt, in den Beispielen jedoch im Südwesten (links unten) des Waldes liegt, müssen entweder die Koordinaten umgerechnet werden, oder es muss beachtet werden, dass Vögel, die nach Norden oder Süden fliegen, sich unintuitiv nach unten bzw. oben bewegen. Eine solche unintuitive Interpretation (quasi ein gespiegelter Kompass) sollte in der Dokumentation zumindest erwähnt, besser noch, begründet werden (z. B. mit der Koordinatenvorgabe der Programmierumgebung). Es muss aber berücksichtigt werden, dass dann auch „Flugrichtung Norden“ nach unten bedeutet und „Flugrichtung Süden“ nach oben.

Andere Fehler könnten sein:

- Nur absolut sichere Felder berechnet.
- Nur sichere Felder berechnet.
- Keine Angabe der sicheren Zeitintervalle.

J2.3 Ausgabe

Die Ausgaben kann – muss aber nicht – graphisch erfolgen. In der in den Materialien angebotenen Blockly-Umgebung war eine graphische Ausgabe zum Beispiel die einzige Möglichkeit der Ausgabe.

In den Materialien war ein Ausgabebeispiel im Textformat angegeben (siehe *Diskstation*). Andere textuelle Ausgabeformate sind aber auch denkbar. Zum Beispiel eine Datei mit den Angaben `<x> <y> <anfang> <ende>`, die für jedes sichere Feld einen sicheren Zeitraum (also min. 30 Minuten) beschreibt. Dabei genügt es auch, wenn absolut sichere Felder implizit über `<x> <y> 1 720` angegeben sind.

J2.4 Beispiele

In den folgenden Beispielen ist die Nord-Süd-Richtung vertauscht – im Gegensatz zu dem Beispiel auf dem Aufgabenblatt, da diese Beispiele auch mit der Blockly-Umgebung erzeugt wurden.

Grün markiert absolut sichere Felder.

Gelb markiert sichere Felder. Die Zahl auf den Felder gibt den negativen Startzeitpunkt des ersten sicheren Zeitraumes an. D. h. eine -7 bedeutet z. B., dass das Feld von Minute 7 bis Minute 36 sicher ist, und dass es vor Minute 7 nie sicher ist. (Danach kann es aber durchaus noch weitere sichere Zeiträume geben.)

Weiß markierte Felder haben keine sicheren Zeiträume. (Die Zahlen geben an, wie viele Minuten vor Sonnenuntergang noch ein Vogel über das Feld geflogen ist.)

1	0	3
-1	1	720
-1	2	720
-1	3	720
-1	4	720
-1	5	720
-1	6	720
-1	7	720
-1	8	720
-1	9	720
-1	10	720
-1	-1	720
-1	-1	720
-1	-1	720
-1	-1	720

Beispiel 1

7	0	1	2	3
3	2	1	0	7
-1	0	-1	-1	-1
-1	0	3	4	5
1	0	3	4	5

Beispiel 2

3	-1	720	720	720	720	720	-1	720	720	0	720	-1	720	-1	720	720	-1	720	720
0	-3	-4	-5	1	2	3	1	5	6	1	8	0	10	0	12	-24	1	-22	-21
1	-1	-1	-1	-1	-1	-1	-1	-1	4	2	2	1	0	3	3	-24	0	-22	-21

Beispiel 3

1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
15	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	0	5	6	7	3	9	10	11	1	3	1	-1	-1
9	2	4	6	5	0	2	2	1	0	0	4	2	0	20
15	3	3	11	9	1	1	9	8	7	6	5	0	1	2
14	0	1	2	3	2	0	0	1	2	3	4	4	2	0
0	1	1	3	4	3	6	7	8	0	1	2	3	3	1
12	6	0	8	6	0	1	2	0	4	5	0	1	2	2
3	2	1	0	5	5	9	5	1	6	5	4	3	0	1
0	8	11	6	4	6	0	1	2	0	3	2	1	0	4
1	-1	-1	-1	-1	-1	-1	-1	-1	0	7	8	9	7	5
2	-1	-1	-1	-1	-1	-1	-1	-1	-1	5	6	7	8	6
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	4	5	6	7
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Beispiel 4

2	1	0	5	6
-30	720	-27	720	-17
-79	720	-28	720	-18
-80	720	-64	720	-19
-81	720	-65	720	-20
-82	720	-66	720	-21
-83	720	-67	720	-22
3	720	-68	720	-23
2	720	-69	720	3
1	720	-70	720	2
0	720	-71	720	1
21	720	0	720	0
22	720	7	720	9
23	720	0	720	10
24	720	1	720	11
25	720	2	720	12
0	720	-45	720	13
1	720	-44	720	14
2	720	-44	720	15
3	720	-45	720	16
4	720	-46	720	17
5	720	-47	720	18
6	720	-48	720	19
-54	720	-49	720	-52
2	1	0	5	6
-54	720	-51	720	-50
-55	720	-52	720	-49
-56	720	-53	720	-48
-57	720	15	720	-47
-1	720	16	720	-46
-1	720	17	720	-46
1	0	3	4	5
-115	720	-112	720	-48
-114	720	-111	720	-49
-113	720	-110	720	-50
-112	720	-109	720	-51
-113	720	-108	720	-52
17	720	-107	720	19
16	720	-106	720	18
15	720	-105	720	17
14	720	-104	720	16
13	720	-103	720	15
12	720	-103	720	14
11	720	-104	720	13
2	1	0	5	6
9	720	-16	720	11
8	720	-15	720	10
7	720	-14	720	9
6	720	-13	720	0
5	720	-12	720	1
0	720	-11	720	2
1	720	-10	720	3
1	0	3	4	4
-24	720	-8	720	-22
-23	720	-7	720	-21
-22	720	-6	720	-20
-21	720	-5	720	-19
-20	720	-4	720	-18
-19	720	-3	720	-17
1	0	3	4	5

Beispiel 5

-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	4	5	6	7	-41	-40	-39	11	-37	-36	-35	15	-33	-32	-31	19
-3	720	720	720	-3	720	720	720	-3	720	720	720	-3	720	720	720	-3	720	720	720	-3	720	720	720	-3	720	720	720	-3
-4	720	720	720	-4	720	720	720	-4	720	720	720	-4	720	720	720	-4	720	720	720	-4	720	720	720	-4	720	720	720	-4
-5	720	720	720	-5	720	720	720	-5	720	720	720	-5	720	720	720	-5	720	720	720	-5	720	720	720	-5	720	720	720	-5
-6	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	4	5	6	7	-41	-40	-39	11	-37	-36	-35	15	-33	-32	-31	19
-7	720	720	720	-7	720	720	720	-7	720	720	720	-7	720	720	720	-7	720	720	720	-7	720	720	720	-7	720	720	720	-7
-8	720	720	720	-8	720	720	720	-8	720	720	720	-8	720	720	720	-8	720	720	720	-8	720	720	720	-8	720	720	720	-8
-9	720	720	720	-9	720	720	720	-9	720	720	720	-9	720	720	720	-9	720	720	720	-9	720	720	720	-9	720	720	720	-9
-10	-3	-4	-5	-10	-7	-8	-9	-10	-11	-12	-13	-14	4	5	6	7	-41	-40	-39	11	-37	-36	-35	15	-33	-32	-31	19
-11	720	720	720	-11	720	720	720	-11	720	720	720	-11	720	720	720	-11	720	720	720	-11	720	720	720	-11	720	720	720	-11
-12	720	720	720	-12	720	720	720	-12	720	720	720	-12	720	720	720	-12	720	720	720	-12	720	720	720	-12	720	720	720	-12
-13	720	720	720	-13	720	720	720	-13	720	720	720	-13	720	720	720	-13	720	720	720	-13	720	720	720	-13	720	720	720	-13
3	18	17	16	3	14	13	12	3	0	1	2	3	4	5	4	3	2	1	0	3	12	13	14	3	16	17	18	3
4	720	720	720	4	720	720	720	4	720	720	720	4	720	720	720	4	720	720	720	4	720	720	720	4	720	720	720	4
5	720	720	720	5	720	720	720	5	720	720	720	5	720	720	720	5	720	720	720	5	720	720	720	5	720	720	720	5
6	720	720	720	6	720	720	720	6	720	720	720	6	720	720	720	6	720	720	720	6	720	720	720	6	720	720	720	6
7	18	17	16	7	14	13	12	7	0	1	2	3	4	5	4	3	2	1	0	7	12	13	14	7	16	17	18	7
-41	720	720	720	-41	720	720	720	-41	720	720	720	-41	720	720	720	-41	720	720	720	-41	720	720	720	-41	720	720	720	-41
-40	720	720	720	-40	720	720	720	-40	720	720	720	-40	720	720	720	-40	720	720	720	-40	720	720	720	-40	720	720	720	-40
-39	720	720	720	-39	720	720	720	-39	720	720	720	-39	720	720	720	-39	720	720	720	-39	720	720	720	-39	720	720	720	-39
11	-3	-4	-5	11	-7	-8	-9	11	-11	-12	-13	3	4	5	6	-42	-41	-40	-39	-38	-37	-36	-35	-38	-33	-32	-31	-38
-37	720	720	720	-37	720	720	720	-37	720	720	720	-37	720	720	720	-37	720	720	720	-37	720	720	720	-37	720	720	720	-37
-36	720	720	720	-36	720	720	720	-36	720	720	720	-36	720	720	720	-36	720	720	720	-36	720	720	720	-36	720	720	720	-36
-35	720	720	720	-35	720	720	720	-35	720	720	720	-35	720	720	720	-35	720	720	720	-35	720	720	720	-35	720	720	720	-35
15	-3	-4	-5	15	-7	-8	-9	15	-11	-12	-13	3	4	5	6	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33	-32	-31	-34
-33	720	720	720	-33	720	720	720	-33	720	720	720	-33	720	720	720	-33	720	720	720	-33	720	720	720	-33	720	720	720	-33
-32	720	720	720	-32	720	720	720	-32	720	720	720	-32	720	720	720	-32	720	720	720	-32	720	720	720	-32	720	720	720	-32
-31	720	720	720	-31	720	720	720	-31	720	720	720	-31	720	720	720	-31	720	720	720	-31	720	720	720	-31	720	720	720	-31
19	-3	-4	-5	19	-7	-8	-9	19	-11	-12	-13	3	4	5	6	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33	-32	-31	-30

Beispiel 6

J2.5 Bewertungskriterien

- Alle in der Aufgabenstellung geforderten Daten müssen von der Lösung berechnet werden, also insbesondere
  - die absolut sicheren Felder,
  - die sicheren Felder und
  - ein sicheres Zeitintervall für jedes sichere Feld.
- Diese Daten sollten in der Ausgabe, ob graphisch oder textuell, ausdrücklich enthalten sein.

- Der Simulationsablauf muss korrekt geregelt sein. Z. B. muss im Zeitschritt die Bewegung der Vögel nach dem Aktualisieren des Feldes erfolgen, damit auch die Vögel, die in diesem Zeitschritt starten, das richtige Feld markieren.
- Der Vogelflug muss korrekt ablaufen. Insbesondere müssen Startzeit und Flugrichtung in Übereinstimmung mit der Eingabe umgesetzt sein; auch andere Aspekte wie etwa der Richtungswechsel müssen stimmen. Wenn Nord- und Süd-Richtung in den Beispielen vertauscht sind, muss die Flugrichtung natürlich entsprechend angepasst sein.
- Die Simulation muss nicht zwingend in Einzelschritten durchlaufbar sein.

Aufgabe 1: Zimmerbelegung

1.1 Vorbereitung: Modellierung

Um diese Aufgabe zu lösen, wollen wir zunächst eine einfache mathematische Modellierung für das Problem finden. Beim Einlesen der Eingabedaten ordnen wir zunächst jeder Schülerin eine natürliche Zahl zu, praktischerweise der Reihe nach von 1 bis N, wenn es genau N Schülerinnen gibt.

Weiterhin modellieren wir die gegebenen Präferenzen zwischen den Schülerinnen als Graph. Wer mit den Grundbegriffen der Graphentheorie (Knoten, (gerichtete) Kanten, Pfade, Adjazenzmatrizen etc.) vertraut ist, kann den folgenden einleitenden Abschnitt überspringen. Für alle anderen ist der folgende Abschnitt hoffentlich hilfreich, um zumindest die Konstruktion der Adjazenzmatrix nachzuvollziehen, die für die Repräsentation des Graphen verwendet wird.

Von Tabellen zu Graphen

Betrachten wir das erste Beispiel, zur Vereinfachung ohne Cleo und Jo. Die Schülerinnen bekommen dann folgende Zahlen zugeordnet:

Anna	1
Paula	2
Dani	3
Lotta	4
Steffi	5

Mit dem folgendem Zeilenvektor (man könnte auch einfach „Liste“ sagen) v können die Präferenzen von Anna gespeichert werden[1]:

$$\begin{bmatrix} ? & 1 & D & D & D \end{bmatrix}$$

Die letzten 4 Einträge ergeben sich direkt aus den Angaben in der Aufgabenstellung. Dabei bezeichnet v_j (der j-te Eintrag von v) die Präferenz von Anna bezüglich der Schülerin mit der Nummer j. Wenn Anna mit Schülerin j auf einem Zimmer sein will, setzen wir $v_j = 1$. Wenn Anna mit Schülerin j nicht auf einem Zimmer sein will, setzen wir $v_j = 0$. Wenn es Anna egal ist, ob sie mit Schülerin j auf einem Zimmer landet, dann setzen wir $v_j = D$. Aber was ist mit dem ersten Eintrag von v, an dessen Stelle wir ein Fragezeichen hingeschrieben haben? Wenn wir davon ausgehen, dass Anna mit sich selbst auf einem Zimmer sein will, dann können wir auch $v_1 = 1$ setzen.

Um die Präferenzen von Paula zu repräsentieren, können wir entsprechend den folgenden Zeilenvektor w verwenden:

$$\begin{bmatrix} 0 & 1 & D & D & D \end{bmatrix}$$

Hier gehen wir davon aus, dass Paula mit sich selbst auf einem Zimmer sein will, und setzen deshalb $w_2 = 1$.

[1] D als Abkürzung für “Don’t Care”.

Für die 5 Schülerinnen aus dem gegebenen Beispiel enthalten wir also 5 Zeilenvektoren. Wenn wir diese einfach untereinander schreiben, erhalten wir die folgende Matrix (oder: Tabelle) G.

$$\begin{bmatrix} 1 & 1 & D & D & D \\ 0 & 1 & D & D & D \\ D & D & 1 & 0 & 1 \\ D & D & 0 & 1 & 1 \\ D & D & D & D & 1 \end{bmatrix}$$

In dieser beschreibt der Eintrag $G_{i,j}$ (d.h. der Eintrag in Zeile i und Spalte j) die Präferenz der Schülerin mit der Nummer i bezüglich der Schülerin mit der Nummer j. Dabei beschreiben die Diagonaleinträge $G_{i,i}$ die Präferenzen der Schülerinnen gegenüber sich selbst. Diese Matrix können wir in unserem Programm unkompliziert mit einem zweidimensionalen Array abspeichern.

Anders interpretiert entspricht diese Darstellung einem Graphen, und die angegebene Matrix ist tatsächlich eine Adjazenzmatrix, die einen Graphen definiert. Die Graphentheorie ist ein Teilbereich der Mathematik, der bei vielen praktischen Problemen Anwendung findet; die Informatik wiederum kennt viele effiziente Algorithmen für Graphen. Zum Verständnis der restlichen Musterlösung für diese Aufgabe ist es hilfreich, sich zunächst mit den wichtigsten Grundbegriffen der Graphentheorie vertraut zu machen. Ab dem folgenden Absatz wird ein grundsätzliches Verständnis der Begriffe von Knoten und (gerichteten) Kanten, sowie Pfaden[2] vorausgesetzt.[3]

Graphen

Als anschauliche Datenstruktur eignet sich ein (vollständiger[4]) Graph $G = (V, E)$, in dem wir die Menge der Schülerinnen mit der Menge der Knoten identifizieren. Die i-te eingelesene Schülerin entspricht dann v_i. Wie in Beispiel 1 erkennbar kann es vorkommen, dass Anna mit Paula auf ein Zimmer will, aber Paula nicht mit Anna. Daher ist der Graph gerichtet: es gibt genau dann eine Kante von v_i nach v_j mit Beschriftung 1, wenn Schülerin i ein Zimmer mit Schülerin j teilen möchte. Wenn i mit j kein Zimmer teilen möchte, fügen wir eine Kante (i, j) mit Beschriftung 0 hinzu. Wenn i keine Meinung zu j hat, fügen wir eine Kante (i, j) mit Beschriftung D hinzu. Um den Graphen möglichst einfach zu modellieren, verwenden wir eine Adjazenzmatrix: Für n Schülerinnen benötigen wir eine Matrix $G \in \{0, 1, D\}^{N \times N}$. Wenn es eine Kante von i nach j gibt mit Beschriftung b, so ist $G_{i,j} = b$. Das erste Beispiel (zur Vereinfachung ohne Cleo und Jo) wird entsprechend dieser Vorschrift durch den folgenden Graphen bzw. durch die folgende Matrix modelliert:

$$\begin{bmatrix} 1 & 1 & D & D & D \\ 0 & 1 & D & D & D \\ D & D & 1 & 0 & 1 \\ D & D & 0 & 1 & 1 \\ D & D & D & D & 1 \end{bmatrix}$$

Dabei gehen wir vereinfachend davon aus, dass alle Schülerinnen glücklich und ohne Selbsthass leben und daher gern mit sich selbst ein Zimmer teilen, d.h. $G_{i,i} = 1$ für alle i. Im später

[2] `https://de.wikipedia.org/wiki/Weg_(Graphentheorie)`
[3] `https://de.wikipedia.org/wiki/Graph_(Graphentheorie)`
[4] In einem vollständigen Graphen gibt es zwischen jeden zwei Knoten eine Kante.

vorgestellten Verfahren müssen diese Einträge allerdings gar nicht getestet werden, daher kann die Diagonale auch andere beliebige Einträge enthalten.

Eine Belegung, welche die N Schülerinnen in Z Zimmer einteilt, kann man mathematisch als Abbildung B modellieren: $B : \{1 \dots N\} \to \{1 \dots Z\}$ für ein $Z \in \{1 \dots N\}$. Eine Belegung die alle in der Tabelle G dargestellten Wünsche aller Schülerinnen erfüllt, nennen wir eine *passende Belegung*; sie erfüllt diese Bedingungen: $G_{i,j} = 0 \Rightarrow B(i) \neq B(j)$ und $G_{i,j} = 1 \Rightarrow B(i) = B(j)$ jeweils für alle $i, j \in \{1 \dots N\}$.

1.2 Lösungsidee Spezialfall

Um eine Lösung für diese Aufgabe zu entwickeln, betrachten wir zunächst den Spezialfall, in dem jede Schülerin jede ihrer Mitschülerinnen auf eine der beiden Listen gesetzt hat. Das bedeutet, es gibt keine Don't-Care-Fälle; wir können uns demnach auf Matrizen $G \in \{0,1\}^{N \times N}$ beschränken.

Teil 1: Symmetrietest

Die Aufgabenstellung enthält bereits einen Hinweis auf eine notwendige Bedingung für die Existenz einer Belegung, die alle Wünsche erfüllt. Wenn Schülerin i ein Zimmer mit Schülerin j teilen will, j aber nicht mit i, so gibt es keine passende Belegung. Daher prüfen wir zunächst, ob die gegebene Matrix symmetrisch ist, d.h. ob $G_{i,j} = G_{j,i}$ für alle i, j erfüllt ist.[5] Die Funktion PRÜFE_SYMMETRIE_SPEZIALFALL nutzt zwei verschachtelte For-Schleifen, um die Symmetrie zu prüfen.

Algorithmus 3: Symmetrie-Spezialfall prüfen

```
1  prüfe_symmetrie_spezialfall(G) begin
2      for i = 1 ... N do
3          for j = i ... N do
4              if G_{i,j} ≠ G_{j,i} then
5                  return false
6              end
7          end
8      end
9      return true
10 end
```

Ist die gegebene Matrix symmetrisch, bedeutet das aber noch nicht, dass es tatsächlich eine passende Belegung gibt, wie das erste Beispiel (eingeschränkt auf Dani, Lotta und Steffi) ebenfalls zeigt.Daher sind wir noch nicht fertig.

[5] In anderen Worten: Wir prüfen, ob man das Problem auch mit einem ungerichteten Graphen modellieren kann.

Teil 2: Konstruktion einer Belegung

Um weiterhin die Existenz einer passenden Zimmerbelegung zu prüfen, versuchen wir einfach, eine solche Zimmerbelegung zu konstruieren. Dafür orientieren wir uns an den Kanten mit Beschriftung 1 und erzwingen so, dass 2 Schülerinnen i, j mit $G_{i,j} = G_{j,i} = 1$ auf dem gleichen Zimmer sind. Am Ende überprüfen wir, ob sich in irgendeinem Zimmer zwei Schülerinnen befinden, die nicht zusammen auf einem Zimmer sein wollen. In diesem Fall verwerfen wir unsere Belegung und wissen, dass es keine passende Belegung gibt.

Zunächst beginnen wir mit einer beliebigen Schülerin i und notieren alle ihre Mitschülerinnen, mit denen sie sich auf jeden Fall ein Zimmer teilen muss. Das sind auf jeden Fall alle Schülerinnen j mit $G_{i,j} = 1$, ferner aber auch alle anderen Schülerinnen j, die von i aus im Graph über einen Pfad erreichbar sind, in dem jede Kante mit 1 beschriftet ist, d.h. alle j für die es $k_1, k_2, \ldots k_m$ gibt mit $G_{i,k_1} = G_{k_1,k_2} = \cdots = G_{k_{m-1},k_m} = G_{k_m,j} = 1$. Alle diese Schülerinnen finden wir z.B. mit der Funktion FÜLLE_ZIMMER[6] (hier beginnen wir mit der Schülerin s).

Algorithmus 4: Zimmer füllen

```
1  fülle_zimmer(s, G) begin
2      Z ← {}
3      Q ← {s}
4      while Q ≠ {} do
5          Wähle i ∈ Q
6          Q ← Q \ {i}
7          Z ← Z ∪ {i}
8          forall the j mit G_{i,j} = 1 und j ∉ Q ∪ Z do
9              Q ← Q ∪ {j}
10         end
11     end
12     return Z
13 end
```

Betrachten wir als nächstes folgende Situation, in der eine Schulklasse nur aus Ada, Grace und Sheila besteht: Ada möchte mit Grace ein Zimmer teilen, und Grace auch mit Sheila, allerdings möchten Ada und Sheila auf keinen Fall ein Zimmer teilen. Wenn wir die Funktion FÜLLE_ZIMMER mit dem Argument 'Ada' aufrufen, erhalten wir ein Zimmer mit Ada, Grace und Sheila. Dieses Beispiel zeigt, dass FÜLLE_ZIMMER manchmal ungültige Zimmer zusammenstellt. In diesem Fall wissen wir direkt, dass es keine passende Belegung geben kann, da nach Konstruktion zwei Schülerinnen nur dann zusammen auf einem Zimmer sind, wenn dies absolut notwendig ist, um alle Wünsche des Zusammenwohnens zu erfüllen. Wir müssen also nun prüfen, ob die erstellte Zimmerbelegung für alle beteiligten Schülerinnen akzeptabel ist. Dafür vergleichen wir einfach mit der Funktion PRÜFE_ZIMMER die vorher erhaltene Belegung Z mit den im Graphen festgehaltenen Wünschen.[7]

[6] Genau genommen würde es im vorliegenden Spezialfall ausreichen, nur die direkten Nachbarn (über Kanten mit Beschriftung '1') von s zu betrachten: Wenn es weiter entfernte Nachbarn gibt, liefert diese Funktion immer eine ungültige Belegung. Aber so wie hier vorgestellt können wir die Funktion FÜLLE_ZIMMER für den allgemeinen Fall wiederverwenden.

[7] Wenn es in der Aufgabenstellung verlangt wäre, würden wir diese Funktion auch nutzen, um zu zeigen, warum keine passende Belegung existiert.

Algorithmus 5: Zimmer prüfen

```
1 prüfe_zimmer(Z, G) begin
2     forall the i, j ∈ Z mit i ≠ j do
3         if G_{i,j} = 0 then
4             return false
5         end
6     end
7     return true
8 end
```

Bis jetzt können wir also ein Zimmer zusammenstellen und dessen Gültigkeit überprüfen. Als nächstes müssen wir noch sicherstellen, dass tatsächlich alle Schülerinnen einem Zimmer zugeordnet werden. Dafür erstellen wir zunächst eine Liste L mit allen Schülerinnen, die wir anschließend der Reihe nach abarbeiten können unter Verwendung der bereits definierten Hilfsfunktionen.

Algorithmus 6: Zimmerbelegung erstellen

```
1  erstelle_belegung(G) begin
2      B ← []
3      L ← {1...N}
4      while L ≠ {} do
5          Wähle s ∈ L
6          Z ← FÜLLE_ZIMMER(s, G)
7          if PRÜFE_ZIMMER(Z, G) = false then
8              return false
9          end
10         L ← L \ {s}
11         append(B, L)
12     end
13     return B
14 end
```

Als Rückgabewert der Funktion ERSTELLE_BELEGUNG erhalten wir also entweder 'false', falls keine passende Belegung existiert, oder ein Array, das die passende Belegung enthält.[8]

1.3 Lösungsidee allgemeiner Fall

Wie die Beispiele zeigen, kommt der bisher besprochene Spezialfall in der Realität quasi nie vor. Zunächst scheint es, als wäre der allgemeine Fall deutlich komplizierter, da unsere Matrix jetzt drei statt wie bislang nur zwei verschiedene Einträge haben kann. Allerdings zeigen wir jetzt, wie man den bisherigen Ansatz nur leicht abändern muss, um eine Lösung für den Spezialfall zu erhalten. Dafür teilen wir die Lösungsidee wieder in zwei Teile auf.

[8] Puristen, die 'false' nicht für ein Array halten und konsistente Datentypen fordern, können hier alternativ ein leeres Array zurückgeben.

Teil 1a: Symmetrietest

Unsere erste Beobachtung ist, dass das Auftreten von Don't Cares nichts an der uns bereits bekannten notwendigen Bedingung ändert: Wenn i mit j auf ein Zimmer möchte, j aber auf keinen Fall mit i, so ist es nie möglich, eine passende Belegung zu finden. Um diese Bedingung zu prüfen, müssen wir die Funktion PRÜFE_SYMMETRIE_SPEZIALFALL nur leicht abändern und erhalten die Funktion PRÜFE_SYMMETRIE_ALLGEMEIN.

Algorithmus 7: Allgemeine Symmetrie prüfen

```
1  prüfe_symmetrie_allgemein(G) begin
2      for i = 1 ... N do
3          for j = i ... N do
4              if G_{i,j} ≠ G_{j,i} und G_{i,j} ≠ D und G_{j,i} ≠ D then
5                  return false
6              end
7          end
8      end
9      return true
10 end
```

Teil 1b: Eliminierung von Don't Cares

Als nächsten Schritt beobachten wir, dass es möglich ist, manche (aber in der Regel nicht alle) Don't Cares zu eliminieren. Wenn Schülerin i mit j auf einem Zimmer sein will, so ist in jeder passenden Belegung zwingend i mit j zusammen auf einem Zimmer. Ob j auch mit i auf einem Zimmer sein will oder ihr dies egal ist, ist für die Existenz einer passenden Belegung unerheblich. Daher können wir in einer Matrix mit $G_{i,j} = 1$ und $G_{j,i} = D$ auch $G_{j,i} = 1$ setzen. Analog können wir für $G_{i,j} = 0$ und $G_{j,i} = D$ auch $G_{j,i} = 0$ setzen. Dies wird durch die Funktion ELIMINIERE_DON'T_CARES erledigt.

Algorithmus 8: *Don't cares* elimimieren

```
1  eliminiere_don't_cares(G) begin
2      for i = 1 ... N do
3          for j = 1 ... N do
4              if G_{i,j} = 1 und G_{j,i} = D then
5                  G_{j,i} ← 1
6              end
7              else if G_{i,j} = 0 und G_{j,i} = D then
8                  G_{j,i} ← 0
9              end
10         end
11     end
12     return G
13 end
```

Danach gilt für alle i, j: $G_{i,j} = D$ genau dann, wenn auch $G_{j,i} = D$. Somit fahren wir analog zum vorhin besprochenen Spezialfall mit einer symmetrischen Matrix fort.

Teil 2: Konstruktion einer Belegung

Wie können wir jetzt im allgemeinen Fall eine passende Belegung konstruieren? Tatsächlich müssen wir gar nichts besonderes tun – das für den Spezialfall vorgestellte Verfahren funktioniert für symmetrische Matrizen auch im allgemeinen Fall. Der komplette Algorithmus zur Lösung der Aufgabe ist in der Funktion ZIMMERBELEGUNG zusammengefasst.

Algorithmus 9: Zimmerbelegung erzeugen

```
1 zimmerbelegung(input) begin
2     G ← KONSTRUIERE_ADJAZENZMATRIX(input)
3     if PRÜFE_SYMMETRIE_ALLGEMEIN(G) = false then
4         return false
5     end
6     G' ← ELIMINIERE_DON'T_CARES(G)
7     B ← ERSTELLE_BELEGUNG(G') return B
8 end
```

Dabei haben wir die neu eingeführten Don't Cares komplett ignoriert. Beim Füllen der Zimmer werden sie praktisch wie 0-en behandelt. Dies ist auch sinnvoll: Wenn es zwei Schülerinnen wechselseitig egal ist, ob sie zusammen auf einem Zimmer sind, gibt es keinen guten Grund, sie unbedingt zusammen auf ein Zimmer zu packen, da dies zu Konflikten über Ecken führen könnte.

Bei der Prüfung auf Gültigkeit werden Don't Cares dagegen wie 1-en behandelt: Wenn es zwei Schülerinnen egal ist, ob sie auf einem Zimmer sind, dann war es kein Fehler, sie demselben Zimmer zuzuordnen.

Das vorgestellte Verfahren erfüllt die Aufgabe bereits vollständig, das Ergebnis hat jedoch noch einen kleinen Haken...

Eindeutigkeit der Zimmerbelegung

Beim Spezialfall berechnet unser Verfahren die einzig mögliche Zimmerbelegung, falls es eine gibt. Im allgemeinen Fall gibt es jedoch verschiedene mögliche Zimmerbelegungen. Um sich dies klar zu machen, betrachte man ein Beispiel, in dem alle Schülerinnen nach einer gemeinsamen langen Phase der Meditation jegliches Verlangen sowie alle Abneigungen abgelegt haben,[9] d.h. die resultierende Matrix enthält (abgesehen von der Diagonale) nur D-Einträge. In diesem Fall liefert unser Algorithmus immer eine Belegung, in der jede Schülerin ihr eigenes Zimmer bekommt. Das ist zwar im Sinne der Aufgabenstellung eine gültige Lösung, aber vielleicht ist es ja gewünscht, dass es auf einer Klassenfahrt keine Einzelzimmer gibt.

[9] https://de.wikipedia.org/wiki/Drei_Geistesgifte

Optional: Minimale/maximale Zimmeranzahl

Wenn man eine Belegung mit minimaler Zimmeranzahl wünscht,[10] könnte man sich einen neuen Graphen konstruieren, für den wir die Knoten mit den bisher erstellten Zimmern identifizieren. Zwei Zimmer werden genau dann mit einer Kante verbunden, wenn die Zusammenlegung beider Zimmer kein ungültiges Zimmer hervorbringen würde. Dann könnte man jede Knotenteilmenge des Graphen zu einem neuen Zimmer zusammenlegen, insofern es sich bei dieser Teilmenge um eine Clique[11] handelt.

Eine passende Belegung mit der maximalen Anzahl von Zimmern, wie von unserem Algorithmus konstruiert, ist eindeutig. Dagegen ist die passende Belegung mit der minimalen Anzahl von Zimmern nicht eindeutig. Um dies nachzuvollziehen, betrachte man wiederum das Beispiel von Ada, Grace und Sheila, aber diesmal mit Don't Cares: Wenn Ada nicht mit Sheila auf ein Zimmer möchte, aber sonst nur Don't Cares vorhanden sind, liefert unser Algorithmus eine Belegung mit lauter Einzelzimmern. Es gibt jetzt zwei verschiedene mögliche Zusammenlegungen: Ada und Grace könnten zusammengelegt werden oder auch Grace und Sheila. Da aber nicht alle drei zusammengelegt werden können, ist die Belegung mit der minimalen Zimmeranzahl also nicht eindeutig.

In der 2. Runde des Wettbewerbs wäre es sicher eine gute Idee, jetzt noch eine Erweiterung einzubauen. Aber hier kümmern wir uns nicht weiter darum.

1.4 Laufzeitanalyse

Auf eine detaillierte Laufzeitanalyse des vorgestellten Algorithmus wird hier verzichtet. Offensichtlich kann man das vorgestellte Verfahren leicht in Polynomialzeit implementieren.[12] Da alle Klassen nur 43 Schülerinnen (oder weniger) haben, ist es nicht so wichtig, hier das effizienteste Verfahren zu wählen und hochoptimiert zu implementieren.[13] Das heißt aber nicht, dass man zur Lösung dieser Aufgabe ein beliebig ineffizientes Verfahren verwenden kann. Wenn man beispielsweise einfach alle Partitionen (d.h. alle Möglichkeiten, eine Menge in Teilmengen aufzuteilen) durchprobiert, so müsste das Programm bereits für eine Klasse von 12 Schülerinnen stolze 4213597 verschiedene Belegungen prüfen.[14]

Zur Berechnung der Ergebnisse für die Beispiele wurde entsprechend dem präsentierten Pseudocode ein Programm in Python geschrieben, das als Datenstrukturen Sets und Listen verwendet, und das für alle Beispiele inklusive Einlesen der Datei und Ausgabe der Ergebnisse deutlich weniger als 100ms benötigt.

[10] Vielleicht ist das ja am kostengünstigsten.

[11] `https://de.wikipedia.org/wiki/Clique_(Graphentheorie)`

[12] `https://en.wikipedia.org/wiki/P_(complexity)`

[13] Ferner könnte man bei dieser geringen Zahl von Schülerinnen darüber nachdenken, die Laufzeit mit einem alternativen Kostenmaß zu bestimmen: da bei einer 64-Bit-Architektur das bitweise Und/Oder/Xor von zwei Zahlen in einem Schritt berechnet werden kann, sind die verwendeten Mengenoperationen in den vorgeschlagenen Funktionen jeweils in einem Schritt berechenbar.

[14] `https://de.wikipedia.org/wiki/Bellsche_Zahl`

1.5 Alternative Lösungen

Zur Lösung dieser Aufgabe sind verschiedene Verfahren denkbar, die nicht unbedingt auf Graphen basieren müssen. Im Folgenden sind ein paar Ideen skizziert.

Ein Graph von Listen

Im oben vorgestellten Verfahren wurden sowohl die $(+)$-Listen als auch die $(-)$-Listen in einen einzigen Graphen eingearbeitet. Wir können aber auch nur die $(+)$-Listen verwenden, um einen Graphen mit unbeschrifteten Kanten zu erstellen: Für eine Schülerin s mit $(+)$-Liste $s_2, s_3, \ldots s_m$ konstruiere den Knoten v_s mit der Menge $L(v_s) := \{s, s_2, s_3, \ldots s_m\}$. Wir fügen genau dann eine Kante $\{v_s, v_r\}$ hinzu, wenn $L(v_s) \cap L(v_r) \neq \emptyset$. Dann entsprechen die Zusammenhangskomponenten dieses Graphen genau den belegten Zimmern, die wiederum noch auf Gültigkeit getestet werden müssten.

Adjazenzlisten

Wir haben zwecks Anschaulichkeit eine Adjazenzmatrix zur Repräsentation des Graphen verwendet. Dafür brauchen wir allerdings N^2 Speicherplatz. Wenn man sich die Beispiele ansieht, wird aber schnell klar, dass fast alle Einträge der Matrix Don't Cares sind. Dies legt nahe, stattdessen Adjazenzlisten zu verwenden und die Don't Cares gar nicht als Kanten im Graphen zu repräsentieren, wie es das Eingabeformat der vorgegebenen Beispiele auch nahelegt. Das erlaubt eine etwas effizientere Implementierung des hier vorgestellten Verfahrens. Zum Beispiel hat unsere Funktion PRÜFE_SYMMETRIE_SPEZIALFALL offensichtlich eine Laufzeit von $\Theta(N^2)$. Mit Adjazenzlisten ließe sich die gleiche Überprüfung in $\Theta(|E|)$ implementieren.

Relationen

Die Modellierung des Problems muss nicht mit Begriffen der Graphentheorie erfolgen. Alternativ können die Schülerinnen z.B. als Menge aufgefasst werden mit den zweistelligen Relationen $R_+ \subseteq \{1 \ldots N\} \times \{1 \ldots N\}$: *i will mit j ein Zimmer teilen* sowie $R_- \subseteq \{1 \ldots N\} \times \{1 \ldots N\}$: *i will mit j auf keinen Fall ein Zimmer teilen*. Eine gültige Belegung ist eine Partition der Ausgangsmenge. Die belegten Zimmer entsprechen dann den Äquivalenzklassen der symmetrisch-reflexiv-transitiven Hülle von R_+ und für die Überprüfung dieser Zimmer muss getestet werden, ob der Schnitt von R_+ und R_- leer ist.

Trennung statt Vereinigung

Wir haben mit einzelnen Schülerinnen angefangen und diese zu Zimmern zusammengefügt. Alternativ kann man natürlich auch mit einem einzigen großen Zimmer beginnen, das alle Schülerinnen enthält, und dies solange in verschiedene Zimmer aufteilen, bis jedes Zimmer keine Schülerinnen mehr enthält, die sich nicht ausstehen können.

Logik

Eine ganz andere Möglichkeit ergibt sich, indem man das Problem nicht selbst löst, sondern stattdessen einen SAT-Solver[15] damit beauftragt. Dafür muss man lediglich vorher ein gegebenes Beispiel in eine logische Formel transformieren. Da die N Schülerinnen auf maximal N Zimmer aufgeteilt werden, kann man N Variablen ϕ_i^k einführen, die genau dann mit *true* belegt sind, wenn Schülerin i zu Zimmer k eingeteilt wird. Die Formel $\bigwedge_{i=1\ldots N} \bigvee_{k=1\ldots N} \phi_i^k$ ist dann nur erfüllt, wenn jede Schülerin auf mindestens einem Zimmer eingeteilt ist. Zusätzlich erzwingt die Formel $\bigwedge_{k\in\{1\ldots N\}} \bigwedge_{j\in\{1\ldots N\}\setminus\{k\}} \neg(\phi_i^k \wedge \phi_i^j)$, dass Schülerin i nicht auf zwei verschiedenen Zimmern gleichzeitig eingeteilt ist. Fügt man dann noch Formeln hinzu, um sicherzustellen, dass die Belegung die Wünsche der Mädchen erfüllt (das wird hier der Leserin überlassen), so erhält man als Konjunktion aller Teilformeln eine einzige große Formel, die genau dann erfüllbar ist, wenn eine passende Belegung existiert. An einer erfüllenden Belegung für die Formel kann man dann auch die Zimmerbelegung an den ϕ_i^k direkt ablesen. Indem man die Anzahl der erlaubten Zimmer reduziert,[16] kann man auch die minimal benötigte Zimmeranzahl berechnen.

Behandlung unklarer Eingabedaten

In dieser Bearbeitung haben wir die Einträge auf der Diagonale praktisch ignoriert. Wenn man sich die Beispiele genau anschaut, stellt man aber fest, dass manche Schülerinnen explizit angeben, (nicht) mit sich selbst auf einem Zimmer sein zu wollen. Dies könnte bedeuten, dass die Schülerinnen nur Witze machen oder einfach nicht mitfahren wollen. Oder es könnte bedeuten, dass 2 Schülerinnen mit gleichem Namen mitfahren. Hier sind verschiedene Interpretationen zulässig, die zu verschiedenen Ergebnissen führen.

1.6 Ergebnisse

In diesem Abschnitt werden die Ergebnisse für die vorgegebenen 6 Beispiele angegeben. Einzig für `zimmerbelegung1.txt` existiert keine passende Belegung. Was im Abschnitt über die Eindeutigkeit der Zimmerbelegung schon angesprochen wurde, wird vor allem in Beispiel 5 sichtbar: Das Programm berechnet wann immer möglich eine passende Belegung mit vielen Einzelzimmern. Die Zahl in Klammern vor jedem Zimmer beschreibt die Anzahl der Schülerinnen in diesem Zimmer.

```
zimmerbelegung2.txt
(2) Alina Lilli
(3) Emma Mia Zoe
(1) Lara


zimmerbelegung3.txt
(14) Anna Larissa Lisa Marie Julia Johanna Emma Nele Antonia Emily Jana
   Michelle Sofia Carolin
(1) Hannah
(5) Clara Celine Lea Lina Lena
(13) Josephine Melina Sarah Leonie Kim Pauline Alina Katharina Sophie
   Annika Lilli Pia Vanessa
```

[15] `https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`
[16] Z.B. iterativ durch binäre Suche

```
(10) Jessika Merle Nina Charlotte Laura Lara Luisa Jasmin Celina Miriam


zimmerbelegung4.txt
(6) Anna Jessika Julia Jasmin Pauline Miriam
(19) Hannah Merle Vanessa Nina Charlotte Clara Lara Lina Emma Luisa Nele
   Annika Kim Celine Michelle Pia Carolin Celina Lisa
(17) Sophie Lea Sarah Larissa Sofia Marie Lena Laura Alina Johanna Melina
   Emily Katharina Josephine Jana Lilli Leonie
(1) Antonia


zimmerbelegung5.txt
(6) Hannah Sarah Lisa Lena Johanna Emily
(3) Kim Leonie Luisa
(2) Lara Pauline
... sonst nur Einzelzimmer


zimmerbelegung6.txt
(25) Anna Leonie Lisa Marie Lena Laura Lara Julia Alina Emma Luisa Nele
   Antonia Katharina Lina Pia Carolin Miriam Jessika Josephine Nina
   Charlotte Clara Kim Pauline
(11) Hannah Melina Sarah Larissa Johanna Emily Sophie Jana Lilli Celine
   Vanessa
(3) Merle Sofia Jasmin
(2) Lea Michelle
(2) Annika Celina
```

Die Angabe in Beispiel 5, dass Marie nicht mit sich selbst in einem Zimmer sein will, wird hier als unsinnig ignoriert. Es ist aber auch akzeptabel, diesen Wunsch ernst zu nehmen; dann existiert für dieses Beispiel keine passende Belegung.

Weitere Beispiele

Nicht lösbar:

```
Alexa
+ Bettina
-

Bettina
+ Chiara
-

Chiara
+ Dajana
-

Dajana
+ Emma
-

Emma
+ Fiona
-

Fiona
+
- Alexa
```

Lösbar:

```
Alexa
+ Bettina Fiona
-

Bettina
+
- Chiara

Chiara
+
- Dajana

Dajana
+ Alexa
- Hanna

Emma
+ Alexa Bettina
-

Fiona
+
- Gisa

Gisa
+ Hanna
-

Hanna
+ Gisa
- Chiara
```

Das linke Beispiel ist nicht lösbar, da Alexa mit Bettina, Bettina mit Chiara, Chiara mit Emma und Emma mit Fiona auf ein Zimmer möchten, Fiona aber nicht mit Alexa. Das rechte Beispiel ist (nur) lösbar über die Einteilung (Alexa, Bettina, Dajana, Emma), (Chiara), (Gisa, Hanna).

1.7 Bewertungskriterien

- Eine Lösung für den allgemeinen Fall ist auch eine Lösung für den spezielleren Fall ohne Don't Cares. Daher ist es ausreichend, eine korrekte Lösung für den allgemeinen Fall anzugeben. Wenn nur eine Lösung für den speziellen Fall angegeben wird, so ist dies nicht ausreichend.

- Das Verfahren sollte grundsätzlich korrekt und vollständig sein, d.h. für alle Eingaben (nicht nur für die Beispiele) genau dann eine gültige Belegung liefern, wenn eine solche existiert.
- Es muss *nicht* die minimale Anzahl an Zimmern gefunden werden. Im Extremfall ganz ohne Präferenzen kann auch jede ihr eigenes Zimmer bekommen – oder alle in ein Zimmer gesteckt werden.
- Die vorgeschlagene Lösung muss nicht besonders effizient sein. Brute-Force (z.B. Ausprobieren aller Einteilungen) ist allerdings nicht akzeptabel.
- Es ist schön, aber nicht erforderlich, die Eindeutigkeit zu diskutieren.
- Es ist schön, aber nicht erforderlich, eine Belegung mit der minimalen Anzahl an Zimmern zu berechnen.
- Das Programm muss auf alle 6 Beispiele angewendet werden, und die Dokumentation sollte die (korrekten) Ergebnisse enthalten. Zusätzliche Beispiele sind schön, aber nicht erforderlich.
- Die Ausgabe muss nicht im vorgeschlagenen Format erfolgen, aber sie sollte eindeutig und übersichtlich sein.
- Falls die Diagonaleinträge anders behandelt werden als hier vorgeschlagen, so sind andere Ergebnisse möglich. Diese sollten dann aber explizit erklärt werden.

Aufgabe 2: Schwimmbad

2.1 Lösungsidee

Als erstes kann man bei dieser Aufgabe feststellen, dass das tatsächliche Alter der Personen gar keine Rolle spielt. Die Preise hängen lediglich von drei Altersstufen ab:

Kinder: 0–3 Jahre alt

Jugendliche: 4–15 Jahre alt

Erwachsene: 16–∞ Jahre alt

Es genügt also auch, einfach nur die Altersstufen einzulesen statt des tatsächlichen Alters.

Kinder können in der Preisberechnung ignoriert werden, sobald mindestens ein Erwachsener zur Gruppe gehört, da Kinder (unter 4) in Begleitung eines Erwachsenen kostenlos ins Schwimmbad dürfen. Ansonsten zählen sie zu den Jugendlichen, da sie ja auch unter 16 Jahre alt sind.

Ohne Gutscheine Zunächst ohne Betrachtung der Gutscheine, lässt sich der Preis rekursiv definieren, indem sich Preise für viele Personen durch den Preis eines Tickets und den Preis für die verbleibenden Personen ergibt. Dabei muss das Minimum über alle Möglichkeiten, ein Ticket zu kaufen, gebildet werden (Sechserkarte, Familienkarte, Normalpreis). Der Basisfall, der als Rekursionsanker dient, ist der Preis für 0 Jugendliche und 0 Erwachsene – die kein Geld zahlen müssen, um ins Schwimmbad zu gehen.

Wir definieren den Preis (ohne Gutscheine) also folgendermaßen (mit j – die Anzahl der Jugendlichen und e – die Anzahl der Erwachsenen):

$$
\begin{aligned}
&\text{Preis}(0,0) := 0\,¢ \\
&\text{Preis}(j,e) := \min[&&\text{Jugendpreis} + \text{Preis}(j-1,e), \\
&&&\text{Normalpreis} + \text{Preis}(j,e-1), \\
&&&\text{Familienpreis} + \text{Preis}(j-2,e-2), \\
&&&\text{Familienpreis} + \text{Preis}(j-3,e-1), \\
&&&\text{Sechserpreis} + \text{Preis}(j-6,e), \\
&&&\text{Sechserpreis} + \text{Preis}(j-5,e-1), \\
&&&\quad\vdots \\
&&&\text{Sechserpreis} + \text{Preis}(j,e-6)]
\end{aligned}
$$

Hierbei können die Anzahlen j und e auch negativ werden. Damit auch dieser Fall abgedeckt ist, kann man einfach die folgenden Preisdefinitionen hinzufügen:

$$
\begin{aligned}
&\text{Preis}(j,e) := \text{Preis}(0,e) && \text{falls } j < 0 \\
&\text{Preis}(j,e) := \text{Preis}(j,0) && \text{falls } e < 0
\end{aligned}
$$

Die Werte von den Preisen (Jugendpreis, Normalpreis, Familienpreis, Sechserpreis) hängen von dem jeweiligem Tag (ist Wochenende? Ist Ferien?) ab. Der Umstand, dass der Sechserpreis nicht am Wochenende verfügbar ist, lässt sich zum Beispiel entweder durch Abänderung der obigen Formel berücksichtigen. Alternativ kann man z. B. auch den Sechserpreis am Wochenende teurer machen als sechs Einzeltickets ($\text{Sechserpreis}_{\text{WE}} := 6 \cdot \text{Normalpreis} + 1¢$) – dadurch wird er dann automatisch außen vor gelassen.

Mit Gutscheinen Für den Fall *mit* Gutscheinen können wir den Preis auch analog rekursiv definieren – diesmal über die Anzahl der Gutscheine. Der Basisfall ist hier der Fall mit 0 Gutscheinen und ergibt sich dann als der oben definierte Preis.

$$\text{Preis}_{\text{G}}(j,e,0) := \text{Preis}(j,e)$$

Allerdings ist hier zu beachten, dass nur *höchstens ein* Gutschein für den 10%-Rabatt verwendet werden darf. Dies kann man auf verschiedene Weisen berücksichtigen. Hier unterteilen wir das Problem in die zwei Teile $g > 1$ und $g = 1$ und erlauben nur dass der letzte Gutschein (also der Fall $g = 1$) für den 10%-Rabatt verwendet werden darf.

Für $g > 1$ gilt also:

$$\begin{aligned}\text{Preis}_{\text{G}}(j,e,g) := \min[&\text{Preis}_{\text{G}}(j-1,e,g-1),\\ &\text{Preis}_{\text{G}}(j,e-1,g-1),\\ &\text{Preis}_{\text{G}}(j,e,1)] \qquad \text{Gutscheine nicht nutzen}\\ &\qquad\qquad\qquad\quad\ \ \text{(außer ggf. 10\% Rabatt)}\end{aligned}$$

Für $g = 1$ gilt dann:

$$\begin{aligned}\text{Preis}_{\text{G}}(j,e,1) := \min[&\frac{9}{10} \cdot \text{Preis}(j,e), \qquad \text{Einmalig 10\% Rabatt}\\ &\text{Preis}(j-1,e),\\ &\text{Preis}(j,e-1),\\ &\text{Preis}(j,e)] \qquad \text{Gutschein nicht nutzen}\end{aligned}$$

(Genau genommen kann man in dieser Definition die letzte Zeile weglassen, da $\frac{9}{10}\text{Preis}(j,e)$ ja auf jeden Fall kleiner als $\text{Preis}(j,e)$ und daher $\text{Preis}(j,e)$ nicht das Minimum sein kann.)

Dynamische Programmierung

Wenn man die Funktionen zur Preisberechnung so wie oben beschrieben implementiert, stellt man schon bei der Funktion Preis_{G} fest, dass diese sich mindestens 2^g mal selber aufruft, da in jedem Aufruf immer zwei rekursive Aufrufe mit um 1 vermindertem g stattfinden. Die Berechnung ist also exponentiell im Paramater g. Analog ist die Preisberechnung auch in den Parametern j und e exponentiell (hier aber je nach Implementierung mit leicht unterschiedlichen Koeffizienten). Formal kann man die Laufzeitberechnung mit $k \cdot 2^{j+e+g}$ Schritten nach

oben abschätzen (k beliebig aber fest; man schreibt $\mathcal{O}(2^{j+e+g})$) und bei naiver Implementierung mit $k \cdot 2^{j+e+g}$ Schritten nach unten abschätzen (man schreibt $\Omega(2^j + 2^e + 2^g)$). Dies ist natürlich sehr ineffizient.

Die rekursive Herangehensweise lässt sich aber zum Glück mit Dynamischer Programmierung noch verbessern. Es lässt sich nämlich feststellen, dass die Laufzeit deswegen so schlecht ist, weil viele Preise mehrfach berechnet werden müssen. Man kann also eine Menge Berechnung sparen, wenn man sich einfach schon berechnete Preise merkt, um dann bei Bedarf wieder darauf zurückgreifen zu können.

Dafür erstellt man sich eine dreidimensionale Tabelle, in der man bereits berechnete Werte nachschlagen kann (je eine Dimension für: Anzahl Jugendliche, Anzahl Erwachsene und Anzahl Gutscheine). Diese Tabelle wird üblicherweise „Lookup-Tabelle“ genannt (von engl. *to look up – nachschlagen*). Wir nennen sie hier „PreisLookup$[J,E,G]$“. Werte, die man bereits ausgerechnet hat, speichert man darin ab und greift bei erneutem Bedarf darauf zurück. Dabei sind J, E und G die Maximalwerte der Anzahl Jugendlicher, Erwachsener und Gutscheine (also die Eingabewerte).

Das Ganze könnte algorithmisch für die Funktion Preis_G mit $g > 1$ so aussehen:

Algorithmus 10: Preis berechnen mit Lookup-Tabelle für $g > 1$

```
1 Preis_G(j,e,g) begin
2     if PreisLookup[j,e,g] gesetzt then
3         return PreisLookup[j,e,g]
4     end
5     else
6         PreisLookup[j,e,g]
          ← min[Preis_G(j−1,e,g−1), Preis_G(j,e−1,g−1), Preis_G(j,e,1)]
7         return PreisLookup[j,e,g]
8     end
9 end
```

Analog müssen natürlich noch die Fälle $g = 1$ und $g = 0$ implementiert werden.

Das verbessert die Laufzeit der Preisberechnung nun erheblich, da die Funktion Preis jetzt höchstens $k \cdot j \cdot e \cdot g$ mal aufgerufen wird (mit implementierungsabhängigem Faktor k). Die Laufzeit der Preisberechnung ist damit $\mathcal{O}(j \cdot e \cdot g)$ (weil die Funktion selbst nur eine endliche Anzahl von Schritten hat).

Mögliche Fehler

Teilweise tauchte die Frage auf, ob Jugendliche auch mit Karten für Erwachsene in das Schwimmbad dürfen. Dies lohnt sich eigentlich bei der Familienkarte. Die Aufgabenstellung erlaubt aber explizit Familienkarten nur für 4 Personen, von denen 1 oder 2 erwachsen sind. Daher sollte davon ausgegangen sein, dass Familienkarten nur verwendet werden dürfen, wenn auch mindestens ein Erwachsener dafür zur Verfügung steht.

Aus diesem Grund darf auch nicht davon ausgegangen werden, dass Gutscheine immer zuerst an Erwachsene ausgegeben werden müssen. Bei 4 Jugendlichen und 1 Erwachsenen ist es während

der Schulzeit von Vorteil, einen Gutschein dem Jugendlichen zu geben und für die anderen eine Familienkarte zu kaufen. (Leider wird dieser Fall in den Beispielen nicht getestet.)

Sind mehrere Gutscheine vorhanden, müssen zuerst alle bis auf einen für freie Eintritte verwendet werden. Erst dann kann der letzte Gutschein für eine Ermäßigung um 10% angesetzt werden. Es ist offensichtlich nicht korrekt, zuerst die Ermäßigung anzusetzen: auf Eintrittskarten, die man anschließend gar nicht kauft.

2.2 Beispiele

(Die Angaben in Klammern wurden nicht gefordert.)

Beispiel 0 (Aufgabenblatt) 4 Jugendliche, 1 Erwachsener, 1 Gutschein, Wochenende, Ferien. Was kostet der Eintritt?

- 1·Jugendlich (250 ¢) + 1·Familie (800 ¢) = **1050 ¢**

Beispiel 1 0 Jugendliche, 27 Erwachsene, 3 Gutscheine. Kostet der Eintritt am Wochenende in der Schulzeit mindestens 500 ¢ weniger als irgendwann in den Ferien?

- Ferien (nicht WE): 3·Normal (je 280 ¢) + 4·Sechser (je 1100 ¢) = **5240 ¢**
  (Gutscheine sind nicht gültig)
- nicht-Ferien, WE: (2·freier Eintritt,) 25·Normal (je 350 ¢) (− 10% durch Gutschein): **7875 ¢**

Beispiel 2 4 Jugendliche, 5 Erwachsene, 0 Gutscheine, Ferien. Was kostet der Eintritt am Wochenende und in der Woche?

- Ferien Wochenende: 1·Normal (350 ¢) + 2·Familie (je 800 ¢): **1950 ¢**
- Ferien nicht-Wochenende: 3·Jugendlich (je 200 ¢) + 1·Sechser (1100 ¢): **1700 ¢**

Beispiel 3 14 Erwachsene, 1 Gutschein und 18 Erwachsene, 1 Gutschein, Wochende, Schulzeit. Was sparen die Gruppen, wenn sie gemeinsam gehen?

- Alleine: 32·Normal (je 350 ¢) (− 10% durch Gutschein): **10080 ¢**
  (14·Normal je 350 ¢ − 10% durch Gutschein: 4410 ¢ +
  18·Normal je 350 ¢ − 10% durch Gutschein ¢: 5670 ¢)
- Zusammen: (1·freier Eintritt,) 31·Normal (je 350 ¢) (− 10% durch Gutschein): **9765 ¢**

2.3 Bewertungskriterien

- Alle Regeln müssen korrekt umgesetzt sein.
- Das Programm verwendet eine Abstraktion über die Beispiele. Da in der Aufgabenstellung ein Einlesen von Daten nicht gefordert ist, die Beispiele nur informell beschrieben sind und es sich um wenige Beispiele handelt, ist es ausnahmsweise akzeptabel, wenn die Angaben zur Gruppenkonstellation als leicht veränderliche Variablen bzw. Konstanten im Programm realisiert sind. Nicht akzeptabel ist, wenn die Unterschiede zwischen den Beispielen im prozeduralen Teil des Programmcodes abgehandelt werden, etwa durch if-Bedingungen oder gar unterschiedliche Codestücke.

- Alle Daten über Gruppen können definiert werden: Alter(sstufen) der Personen, Wochenende, Ferien, Gutscheine.
- Die vorgeschlagene Lösung muss nicht besonders effizient sein. Uneingeschränktes Brute-Force (z.B. wenn Kartenkombinationen aufgezählt werden, die nicht zu den Eingaben passen) ist allerdings nicht akzeptabel.
- Es reicht nicht, nur die Preise auszugeben; die Aufgabenstellung verlangt explizit die Ausgabe, welche Karten gekauft werden müssen (Sechserkarte, Familienkarte, Normalpreis). Die Karten müssen aber nicht in irgendeiner Art den Personen zugeordnet werden. Eine Angabe über die Nutzung der Gutscheine ist wünschenswert, aber von der Aufgabenstellung nicht explizit gefordert.
- Die Beispiele von der Webseite werden hinreichend besprochen (immer zwei Preise!).
- Dynamisches Einlesen der Preisregeln über eine selbstdefinierte Regelsprache ist *nicht* nötig (aber eine schöne Idee).

Aufgabe 3: Dreiecke zählen

3.1 Lösungsidee

Wir wählen einen Ansatz, der aus zwei Schritten besteht: Zuerst werden alle Schnittpunkte ermittelt, danach werden die Dreiecke gezählt.

Einschränkungen in der Aufgabenstellung

Der nette Aufgabensteller bewahrt uns vor zwei Sonderfällen:

Keine zwei Strecken auf der gleichen Geraden. In diesem Fall könnten sich die Strecken überlappen, und mehr als einen Schnittpunkt besitzen. Für unser Programm wissen wir: zwei Geraden haben entweder einen oder keinen Schnittpunkt.

Höchstens zwei Strecken treffen sich in einem Punkt. Durch diese Aussage wissen wir: Wenn sich drei Strecken paarweise schneiden, formen sie ein Dreieck. Andernfalls hätte sie sich auch alle in einem Punkt schneiden können, und würden kein Dreieck formen.

Allerdings hat der nette Beispieldatenersteller die Einschränkungen auf eigene Weise interpretiert:

1. Die erste Einschränkung ist im Beispiel 5 genau genommen verletzt: hier liegen zwei Strecken auf einer Geraden. Allerdings überlappen sie sich nicht und machen deshalb keine Probleme.
2. Die zweite Einschränkung gilt in den zur Verfügung gestellten Beispieldaten allgemein nur für berechnete Punkte. In den Beispielen 5 und 6 kommen drei Strecken mit gemeinsamem Endpunkt vor. Wenn man davon ausgeht, dass die Einschränkung für alle Punkte gilt, kann man bei diesen Beispielen auch „Dreiecke mit Flächeninhalt 0“ (alle drei Eckpunkte des Dreiecks sind gleich) erhalten.

Ermitteln der Schnittpunkte

In diesem Schritt wollen wir alle Schnittpunkte ermitteln und diese in einer Matrix speichern. Dazu müssen wir zuerst in der Lage sein, den Schnittpunkt zweier Strecken zu ermitteln. Dies geht am besten mit ein wenig Mathematik. Bei uns sind die Strecken durch beide Endpunkte gegeben. Für die Berechnung ist es jedoch einfacher, diese durch einen Fußpunkt und einen Vektor darzustellen. Wir speichern also für jede Gerade gegeben durch die Punkte a, b:

$$\text{Fußpunkt } p = a$$
$$\text{Richtungsvektor } r = b - a$$

Wenn sich zwei Strecken $A(p,r)$ und $B(q,s)$ schneiden, dann gilt am Schnittpunkt

$$p + tr = q + us \qquad t, u \in \mathbb{R}$$

Diese Gleichung können wir jetzt nach t und u lösen:

$$\begin{aligned} p + tr &= q + us \\ \Leftrightarrow (p + tr) \times s &= (q + us) \times s \\ \Leftrightarrow p \times s + t(r \times s) &= q \times s + u(s \times s) \\ \Leftrightarrow t(r \times s) &= (q - p) \times s + 0 \\ \Leftrightarrow t &= \frac{(q - p) \times s}{r \times s} \end{aligned}$$

Und entsprechend für u:

$$u = \frac{(p - q) \times r}{s \times r}$$

Um den Rechenaufwand zu reduzieren und möglichst viele Zwischenergebnisse von der Berechnung von t wiederverwenden zu können, stellen wir noch um zu:

$$u = \frac{(q - p) \times r}{r \times s}$$

Um zu prüfen, ob sich die beiden Linien schneiden, müssen t, u in $[0, 1]$ liegen, sonst liegt der Schnittpunkt der durch A, B repräsentierten Geraden außerhalb der Strecken.

Am Ende kommen wir auf folgenden kompakten Algorithmus:

Algorithmus 11: Prüfen, ob zwei Strecken sich schneiden

```
1 function Schnittpunkt (A, B)
  Input  : Strecken A(p, r), B(q, s)
  Output: Schnittpunkt, falls sich die Strecken schneiden
2 ΔFußpunkt := p − q
3 Richtungsprodukt := r × s
4 t := (ΔFußpunkt × s)/Richtungsprodukt
5 u := (ΔFußpunkt × r)/Richtungsprodukt
6 if t ∈ [0, 1] ∧ u ∈ [0, 1] then
7 |   return p + tr
8 end
```

Einfacher Algorithmus

Wir erhalten alle Schnittpunkte, indem wir einfach für jedes Paar von Geraden die oben beschriebene Funktion aufrufen und in einer Matrix die Schnittpunkte speichern (Algorithmus 12). Es würde sogar genügen, nur zu speichern, ob es einen Schnittpunkt gibt; dazu müsste Algorithmus 11 nur *wahr* oder *falsch* zurückgeben.

Als nächstes müssen wir die Dreiecke zählen. Dazu können wir über alle Tripel von Geraden iterieren. Dabei müssen wir noch sicherstellen, dass die Schnittpunkte nicht übereinstimmen. In diesem Fall wäre entsprechend kein Dreieck zu zählen Algorithmus 13).

Algorithmus 12: Ermitteln aller Schnittpunkte

```
1  function AlleSchnittpunkte (L)
   Input  : Menge an Strecken L
   Output: Schnittpunktmatrix der Strecken
2  A := Matrix der Größe |L|^2
3  for (i = 0; i < |L|; i++) do
4  |  for (j = i+1; j < |L|; j++) do
5  |  |  A[i,j] := A[j,i] := Schnittpunkt(L[i], L[j])
6  |  end
7  end
8  return A
```

Algorithmus 13: Dreiecke im Graphen zählen

```
1  function DreieckeZählen (A)
   Input  : Schnittpunktmatrix A
   Output: Anzahl der Dreiecke
2  AnzDreiecke := 0
3  for (i = 0; i < |L|; i++) do
4  |  for (j = i+1; j < |L|; j++) do
5  |  |  for (k = j+1; k < |L|; k++) do
6  |  |  |  if (A[i,j] ∧ A[j,k] ∧ A[k,i] ∧ A[i,j] ≠ A[j,k]) then
7  |  |  |  |  AnzDreiecke++
8  |  |  |  end
9  |  |  end
10 |  end
11 end
12 return AnzDreiecke
```

Laufzeit Der erste Teil hängt quadratisch von der Anzahl n der Geraden ab, der hintere kubisch. Das führt zu

$$\mathcal{O}(n^2 + n^3) = \mathcal{O}(n^3)$$

Geht das schneller? Aber deutlich!

Fortgeschrittener Algorithmus

Das Finden aller Schnittpunkte können wir durch den von Balaban 1995 beschriebenen[17] Algorithmus auf $\mathcal{O}(n \log n + k)$ drücken, mit n als Anzahl der Geraden und k als Anzahl der Schnittpunkte. Dieser nutzt die gleiche Idee wie der Bentley-Ottmann-Algorithmus, ist jedoch weniger anschaulich, weshalb ich Letzteren hier vorstellen möchte.

Wir nutzen folgenden Ansatz (unter Vernachlässigung aller Spezialfälle): Wenn man eine zur y-Achse parallele Linie (Sweepline) über die Strecken führt, ändert sich die Menge und y-Reihenfolge der geschnittenen Strecken nur an endlich vielen Stellen. Auch klar ist: Zwei Strecken können sich nur dann schneiden, wenn sie davor nebeneinander Schnittpunkte der Sweepline waren. Durch intelligentes Updaten der Menge an Strecken, die die Sweepline schneiden, kommt man auf Algorithmus 14.

Das Finden aller Dreiecke in einem durch eine Schnittpunktmatrix gegebenen Graphen G (die Schnittpunktmatrix ist dann dessen Adjazenzmatrix) lässt sich auf $\mathcal{O}(a(G)k)$ reduzieren, wobei $a(G)$ die so genannte „Arboricity“[18] des Graphen ist und k wieder die Anzahl der Schnittpunkte. An dieser Stelle gibt es viele interessante Algorithmen, aber der in “Arboricity and subgraph listing algorithms”[19] von Chiba und Nishizeki ist noch gut zu implementieren.

Laufzeit Durch die Verwendung der beschriebenen Algorithmen kommen wir auf

$$\mathcal{O}(n \log n + k + a(G)k) = \mathcal{O}(n \log n + a(G)k)$$

mit n als Anzahl der Strecken, k als Anzahl der Schnittpunkte, und $a(G) \leq \lceil k/n \rceil$ als Arboricity des Schnittpunktgraphen.

Umsetzung

Die Umsetzung der naiven Variante ist im Wesentlichen eine Reihe von geschachtelten `For`-Schleifen. Als Referenz sei hier nur die Funktion zum Ermitteln des Schnittpunktes zweier Geraden angegeben. So oder so ähnlich lässt sich diese in jeder Sprache implementieren, bei Bedarf mit mehr oder weniger Objektorientierung. Dieses Beispiel ist für C#.

[17] Balaban, I. J. (1995), An optimal algorithm for finding segments intersections, Proc. 11th ACM Symp. Computational Geometry, pp. 211–219, doi:10.1145/220279.220302

[18] Das ist die kleinste Anzahl von Baum-Mengen, in die man einen Graph zerlegen kann, und dient u.a. dazu, die Dichte eines Graphen zu charakterisieren.

[19] Chiba, Nishizeki (1985), “Arboricity and subgraph listing algorithms”, SIAM Journal on Computing, Volume 14 Issue 1, Feb. 1985, pp. 210-223, doi:10.1137/0214017

Algorithmus 14: Ermitteln aller Schnittpunkte

```
1  function BentleyOttmann (L)
   Input  : Menge an Geraden L
   Output: Doubly Linked Adjacency Lists A
2  A := Matrix der Größe |L|^2
3  Q := Priority-Queue von Punkten sortiert nach x-Koordinate
4  Q.insert(alle Start und Endpunkten in L)
5  T := Binärbaum von Strecken (die die Sweepline kreuzen) sortiert nach y-Koordinate
6  while Q ≠ ∅ do
7      p := Q.pop()
8      if p ist linker Endpunkt einer Strecke l then
9          T.insert(l)
10         k := Strecke unter l in T
11         if s := Schnittpunkt(k,l) then
12             Q.insert(s)
13         end
14         m := Strecke über l in T
15         if s := Schnittpunkt(l,m) then
16             Q.insert(s)
17         end
18     else if p ist rechter Endpunkt einer Strecke l then
19         k := Strecke unter l in T
20         m := Strecke über l in T
21         T.remove(l)
22         if s := Schnittpunkt(k,m) then
23             Q.insert(s)
24         end
25     else if p ist ein Schnittpunkt von (k,l) then
26         A[k].add(l)
27         A[l].add(k)
28         j := Strecke unter j in T
29         m := Strecke über l in T
30         T.swap(k,l)
31         if s := Schnittpunkt(j,l) then
32             Q.insert(s)
33         end
34         if s := Schnittpunkt(k,m) then
35             Q.insert(s)
36         end
37 end
38 return A
```

Algorithmus 15: Dreiecke im Graphen zählen

```
1  function DreieckeZählen (A)
   Input  : Graph G
   Output: Anzahl der Dreiecke
2  AnzDreiecke := 0
3  (v_1, ..., v_n) := Knoten von G absteigend sortiert nach Grad
4  for i = 1, ..., n-2 do
5  |  Alle Nachbarn von v_i markieren
6  |  foreach Markierter Knoten u do
7  |  |  foreach Nachbar w von u do
8  |  |  |  if w markiert und (u, w, v_i) echtes Dreieck then
9  |  |  |  |  AnzDreiecke++
10 |  |  |  end
11 |  |  end
12 |  end
13 |  G.delete(v_i)
14 end
15 return AnzDreiecke
```

```
private static PointF? GetLineIntersection(Line l1, Line l2)
{
  var s = (-l1.D.Y * (l1.P.X - l2.P.X) + l1.D.X * (l1.P.Y - l2.P.Y))
          / (-l2.D.X * l1.D.Y + l1.D.X * l2.D.Y);
  var t = (l2.D.X * (l1.P.Y - l2.P.Y) - l2.D.Y * (l1.P.X - l2.P.X))
          / (-l2.D.X * l1.D.Y + l1.D.X * l2.D.Y);
  if (s >= 0 && s <= 1 && t >= 0 && t <= 1)
    return new PointF(l1.P.X + t * l1.D.X, l1.P.Y + t * l1.D.Y);
  return null;
}
```

Beispiele

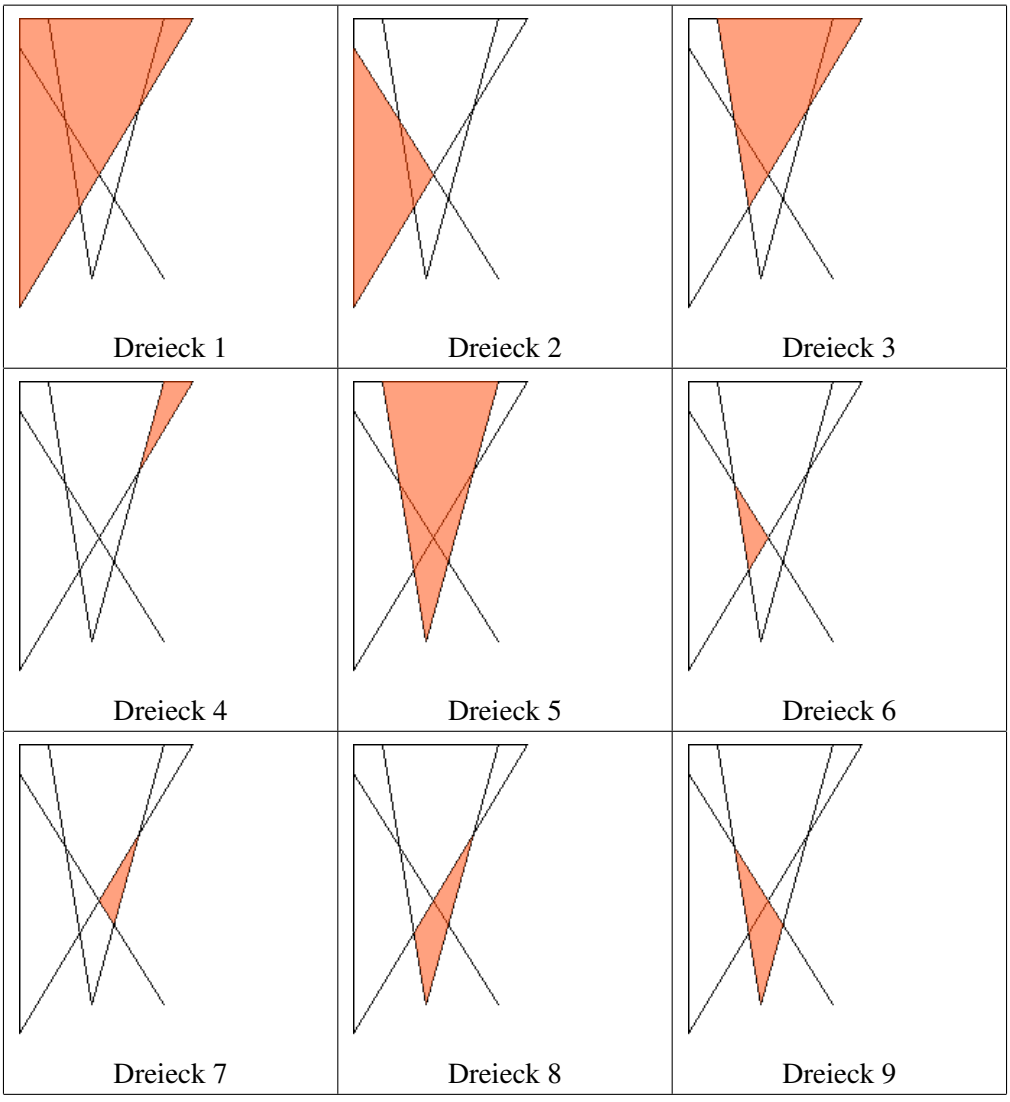
Als Übersicht die Anzahl der zu findenden Dreiecke in jedem der Beispiele:

Beispiel	# 1	# 2	# 3	# 4	# 5	# 6
Anzahl der Dreiecke	9	0	3*	5	**1**	**20**
Anzahl, falls Dreiecke mit Fläche 0 zählen	9	0	3*	5	**4**	**25**
Ursprüngliche Beispieldatei 3**			**7 / 9**			

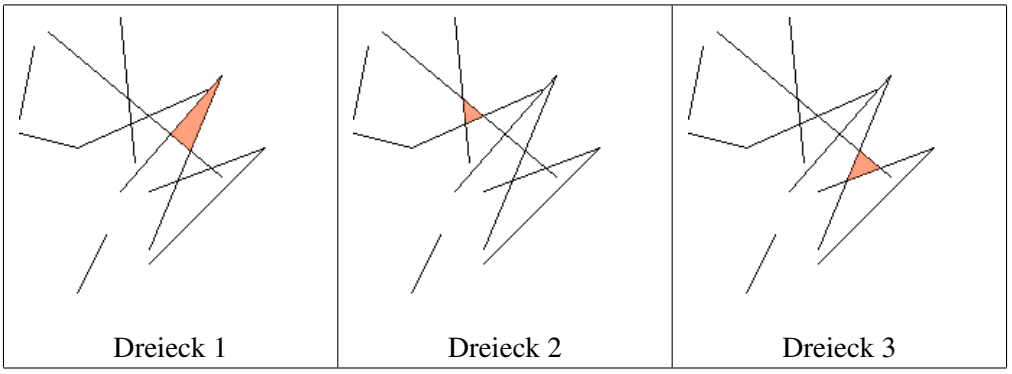
*Mit genügend Toleranz bei der Schnittpunktberechnung sind hier auch 4 Dreiecke findbar. Noch ungenauer sollte die Schnittpunktberechnung aber nicht werden.
**Die Beispieldatei 3 wurde einen Monat nach Veröffentlichung durch eine neue Beispieldatei ersetzt. Wer noch die alte Datei benutzt, konnte 7 Dreiecke finden bzw. 9, falls Dreiecke mit Fläche 0 zählen.

Hier sind die Dreiecke im Einzelnen:

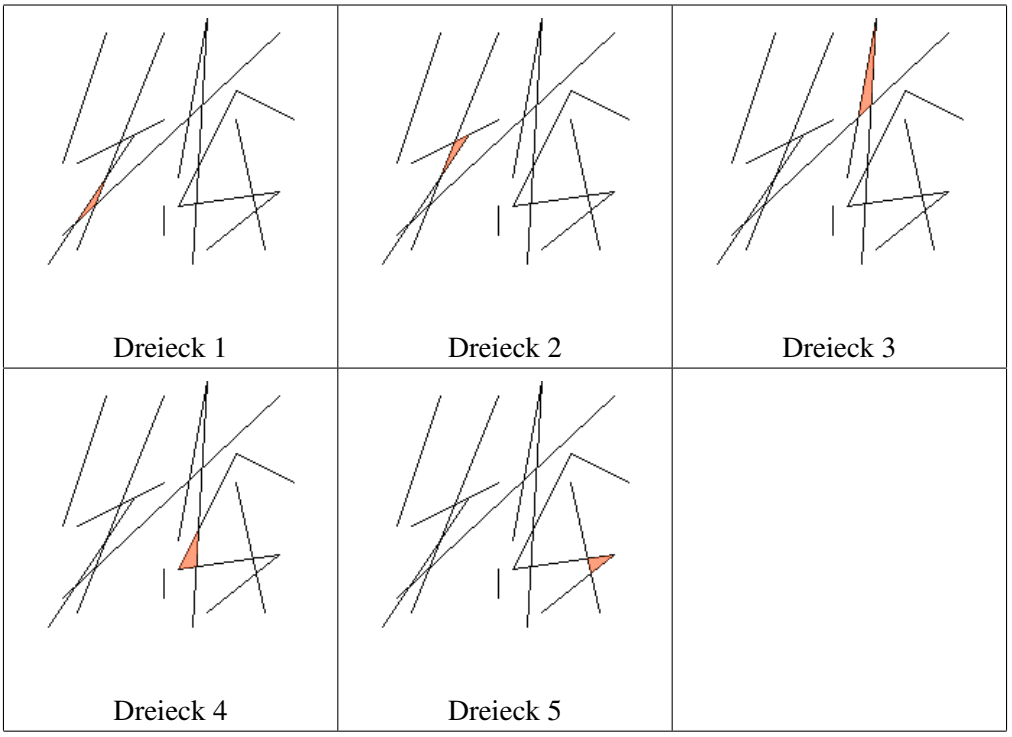
Beispiel 1 (Wegen der einfacheren Behandlung der Koordinaten ist die Darstellung horizontal gespiegelt.)



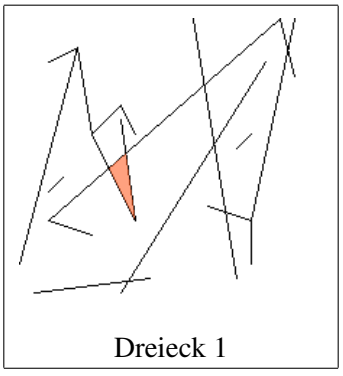
Beispiel 3



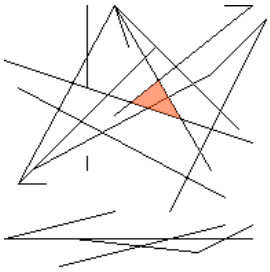
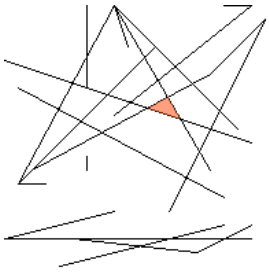
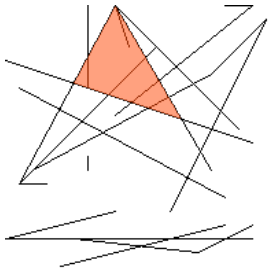
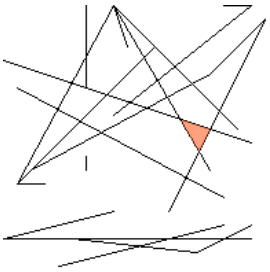
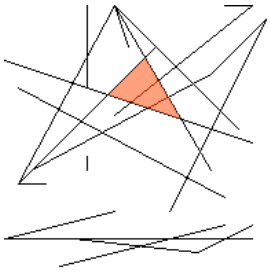
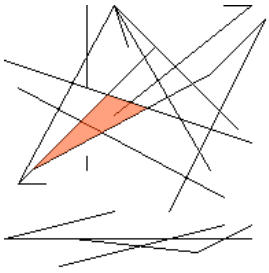
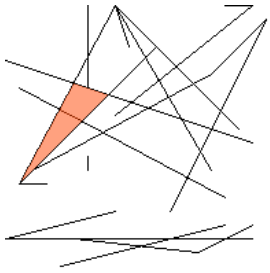
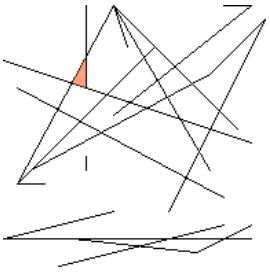
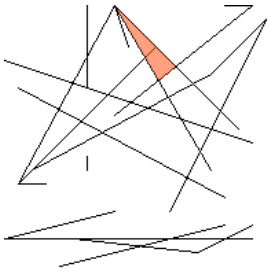
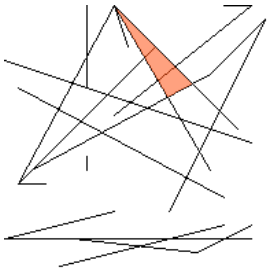
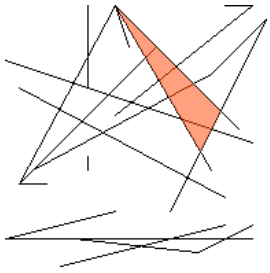
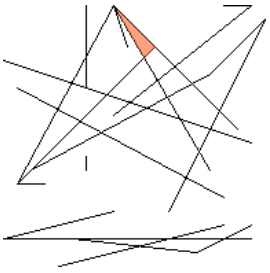
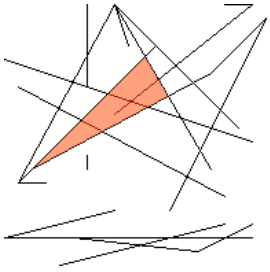
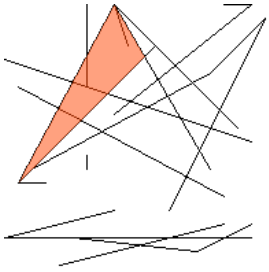
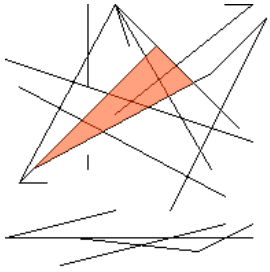
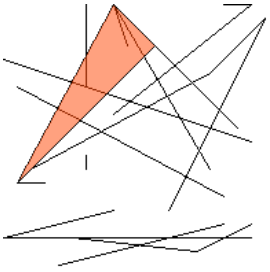
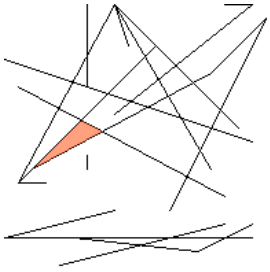
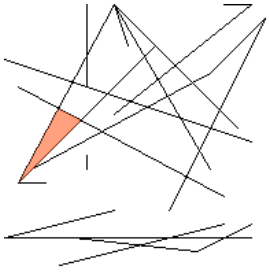
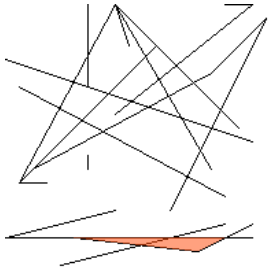
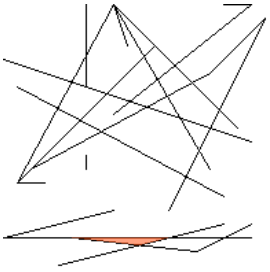
Beispiel 4



Beispiel 5



Beispiel 6

 Dreieck 1	 Dreieck 2	 Dreieck 3	 Dreieck 4
 Dreieck 5	 Dreieck 6	 Dreieck 7	 Dreieck 8
 Dreieck 9	 Dreieck 10	 Dreieck 11	 Dreieck 12
 Dreieck 13	 Dreieck 14	 Dreieck 15	 Dreieck 16
 Dreieck 17	 Dreieck 18	 Dreieck 19	 Dreieck 20

Bewertungskriterien

- Das Verfahren sollte korrekte Ergebnisse liefern. Werden Dreiecke mit Fläche 0 mitgezählt, ist das akzeptabel; aus der Aufgabenstellung konnte man ableiten, dass solche Fälle ausgeschossen sind.
- Eine Laufzeit von $\mathcal{O}(n^3)$ ist problemlos erreichbar. Akzeptiert wurden aber auch größere polynomielle Laufzeiten, mit denen – auf Grund der kleinen Eingabedateien – die Berechnung der Ergebnisse in akzeptabler Zeit möglich ist.
- Eine graphische Ausgabe ist nett, aber nicht notwendig. Genau genommen genügt es, nur die Anzahl der Dreiecke auszugeben. Dann wird aber erwartet, dass das Funktionieren insbesondere der Berechnung der Schnittpunkte in der Dokumentation deutlich wird.
- Die vorgegebenen Beispiele sollten bearbeitet sein. Die Anzahl der gefundenen Dreiecke sollte in der Dokumentation ersichtlich werden. Am besten ist eine konkrete Angabe der Zahl; akzeptabel ist, wenn die Anzahl der Dreiecke aus einer graphischen Ausgabe klar problemlos abgelesen werden kann.
- Weitere Beispiele, die bestimmte Sonderfälle illustrieren, sind wünschenswert, aber nicht gefordert.

Aufgabe 4: Auto-Scrabble

4.1 Vorbemerkung

Wir werden zunächst eine etwas verallgemeinerte Variante des Problems betrachten: Wie kann ein Wort auf maximal k Kennzeichen dargestellt werden, wenn überhaupt? Antworten auf die Fragen aus der Aufgabe ergeben sich unmittelbar aus der Lösung dieses Problems.

4.2 Lösungsidee

Bei der Betrachtung des Problems fällt auf, dass es sich bei den Worten, die auf (mehreren) Kennzeichen stehen können, um eine *reguläre Sprache* handelt. So mag es naheliegend sein, einen regulären Ausdruck für die Kennzeichen-Sprache zu formulieren und mithilfe einer Regex-Bibliothek zu prüfen, ob ein gegebenes Wort dem regulären Ausdruck entspricht:

$$\texttt{\^{}((A|AA|AB|ABG|ABI|AC|...|ZZ)[A-Z]\{1,2\})\{1,}k\texttt{\}\$}$$

Abgesehen davon, dass die meisten Regex-Bibliotheken bei Gruppen nur die letzte Zuordnung speichern, sind reguläre Ausdrücke dieser Form gerade bei längeren Wörtern sehr ineffizient, da die Anzahl der Aufteilungsmöglichkeiten exponentiell wächst. Insbesondere ist mehr Information über das Problem bekannt als in dem Ausdruck codiert ist. So hat es beispielsweise keine Relevanz, ob die Aufteilung auf einem Kennzeichen `AB-C` oder `A-BC` lautet – beide Kennzeichen haben eine Länge von drei Zeichen und das folgende Kennzeichen beginnt in beiden Fällen beim vierten Zeichen.

Für ein effizienteres Lösungsverfahren betrachten wir zunächst ein Teilproblem, welches die Frage beantwortet, wie viele Zeichen am Anfang eines Wortes auf einem Kennzeichen dargestellt werden können. Für das Wort `ABCÖ` wäre dies beispielsweise $\{2, 3\}$. Wir werden dieses Teilproblem im Folgenden als *Präfix-Analyse* bezeichnen. Durch wiederholtes Lösen dieses Teilproblems und der Verwendung der Methode der *dynamischen Programmierung* erhalten wir ein Lösungsverfahren mit einer polynomiellen Laufzeit.

Präfix-Analyse

Zunächst ist zu prüfen, welche Kürzel aus der gegebenen Kürzelliste mit dem Anfang des Wortes übereinstimmen. Im Folgenden sei l die maximale Länge eines Kürzels. Zu Beginn werden alle Kürzel in eine Hashtabelle eingefügt. Das Finden von Kürzel-Präfixen besteht dann lediglich aus l Lookup-Operationen in der Hashtabelle, je ein Lookup für jede mögliche Kürzellänge. Bei einer hinreichend großen Hashtabelle und einer guten Hashfunktion kann ein Lookup in konstanter Zeit erfolgen. Wenn man sämtliche Kürzel-Präfixe gefunden hat, so erhält man die möglichen Kennzeichen, in dem man 1–2 weitere Zeichen des Namens an die Kürzel anhängt. Zu beachten ist, dass diese Zeichen keine Umlaute sein dürfen.

Als Optimierung kann man, wie bereits oben beschrieben, von mehreren Kennzeichen gleicher Länge eines auswählen und die anderen verwerfen.

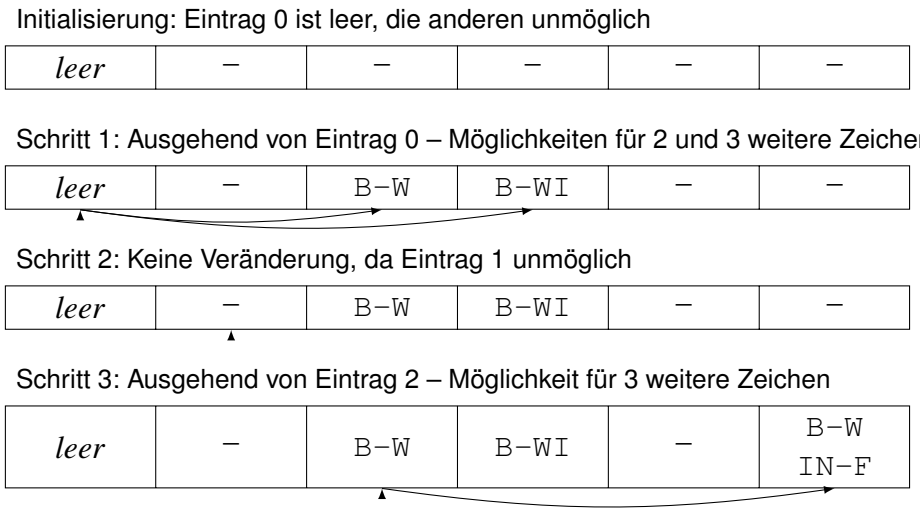


Abbildung 1: Beispiel für Funktionsweise des iterativen Verfahrens für das Wort `BWINF`. Spalte i (beginnend bei 0) enthält eine mögliche Aufteilung der ersten i Buchstaben des Wortes. Nach der Initialisierung werden in jedem Schritt von einem Eintrag mögliche Verlängerungen um ein weiteres Kennzeichen in die Tabelle eingetragen.

Dynamische Programmierung

Sei im Folgenden n die Länge des Wortes, das auf den Kennzeichen geschrieben sein soll. Zur Lösung des gesamten Kennzeichen-Problems verwenden wir ein iteratives Verfahren, welches eine Tabelle der Länge $n+1$ ausfüllt. Eintrag i in der Tabelle beschreibt, wie die ersten i Buchstaben des Wortes auf Kennzeichen dargestellt werden können, oder dass dies *nicht möglich* ist.

Der Eintrag 0 wird anfangs auf eine leere Liste gesetzt (ein leeres Wort kann mithilfe von null Kennzeichen dargestellt werden), alle anderen Einträge werden mit *nicht möglich* initialisiert.

Für einen gültigen (*möglichen*) Eintrag i in der Tabelle kann man mit der oben beschriebenen Methode die Kennzeichen(-längen) bestimmen, die im Wort ab Buchstabe i beginnen. Diese Kennzeichen werden jeweils an die Liste von Eintrag i angehängt und an den entsprechenden Positionen in die Tabelle eingetragen – solange die Anzahl der Kennzeichen nicht größer ist als die maximale Kennzeichen-Anzahl k.

Am Ende des Verfahrens enthält Eintrag n entweder eine mögliche Unterteilung des Wortes in maximal k Kennzeichen oder den Vermerk *nicht möglich*.

4.3 Beispiele

Teilaufgaben 1 und 2

Beispiele für Wörter, die nicht auf Kennzeichen stehen können:

`TIMO` `None`
Anmerkung: Es gibt kein Kürzel `TI` oder `TIM`

TIER None
Anmerkung: Es gibt kein Kürzel TI oder TIE

SÜD None
Anmerkung: Es gibt kein Kürzel SÜ

TAT None
Anmerkung: Es gibt kein Kürzel T oder TA

IN None
Anmerkung: Es gibt kein Kürzel I

UM None
Anmerkung: Es gibt kein Kürzel U

Teilaufgaben 3 und 4

Mögliche Lösungen für die gegebenen Beispiele:

HEIKE ('HEI-KE')

BIBER ('B-I', 'BE-R')

BUNDESWETTBEWERB ('B-U', 'ND-ES', 'WE-TT', 'B-E', 'WER-B')

CLINTON ('C-LI', 'NT-ON')

DONAUDAMPFSCHIFFFAHRTSKAPITÄNSMÜTZE None
Anmerkung: Es gibt kein Kürzel mit dem Buchstaben Ä

ETHERNET ('E-T', 'H-E', 'RN-ET')

INFORMATIK ('IN-FO', 'R-M', 'AT-IK')

LLANFAIRPWLLGWYNGYLLGOGERYCHWYRNDROBWLLLLANTYSILIOGOGOGOCH
('LL-A', 'NF-AI', 'RP-WL', 'LG-WY', 'N-GY', 'LL-GO',
'GER-YC', 'H-WY', 'RN-D', 'RO-B', 'WL-LL', 'LAN-TY',
'S-I', 'LI-OG', 'OG-OG', 'OC-H')

RINDFLEISCHETIKETTIERUNGSÜBERWACHUNGSAUFGABENÜBERTRAGUNGSGESETZ
None

SOFTWARE ('SO-FT', 'WA-RE')

TRUMP ('TR-U', 'M-P')

TSCHÜSS None

VERKEHRSWEGEPLANUNGSBESCHLEUNIGUNGSGESETZ ('VER-KE', 'HR-SW',
'EG-EP', 'LAN-UN', 'GS-BE', 'SC-HL', 'EU-NI', 'G-UN',
'GS-GE', 'SE-TZ')

Falls in den Wörtern vorher die Umlaute umgewandelt wurden, ergeben sich für die Wörter mit Umlauten:

TSCHUESS ('TS-CH', 'UE-SS')

RINDFLEISCHETIKETTIERUNGSUEBERWACHUNGSAUFGABENUEBERTRAGUNGSGESETZ
None

DONAUDAMPFSCHIFFFAHRTSKAPITAENSMUETZE ('D-O', 'NAU-D', 'AM-PF', 'SC-HI', 'FF-F', 'AH-R', 'TS-KA', 'PI-TA', 'EN-SM', 'UE-T', 'Z-E')

4.4 Bewertungskriterien

- Alle Teilaufgaben sollen bearbeitet worden sein; die Bearbeitung soll also enthalten:
  - eine Begründung, dass TIMO nicht auf einem Kennzeichen stehen kann,
  - andere Wörter mit 2, 3 und 4 Buchstaben, die nicht auf Kennzeichen stehen können, sowie
  - eine Implementierung des geforderten Programms (es genügt ein Programm, das die Teilaufgaben 3 und 4 gemeinsam abdeckt).
- Die Ergebnisse sollten korrekt sein. Wenn es keine Möglichkeit zur Darstellung gibt, sollte das Programm dies ausgeben (es sollte also nicht etwa zu Laufzeitfehlern kommen). Insbesondere ist zu beachten, dass der Mittelteil eines Kennzeichens keine Umlaute enthalten darf.
- Das gewählte Verfahren muss nicht besonders effizient sein. Einigermaßen clevere (rekursive) Suchbaum-Verfahren sind akzeptabel, eine vollständige Aufzählung möglicher Kennzeichen bis zum Match mit dem Suchbegriff aber z. B. nicht.
- Eine lange Laufzeit kann auch daran liegen, dass versucht wird, nicht nur eine, sondern alle Aufteilungen zu bestimmen und auszugeben. Solange dadurch die Bearbeitung der Beispiele nicht beeinträchtigt wurde, ist das akzeptabel.
- Mindestens eine mögliche Aufteilung oder die Feststellung, dass keine Aufteilung möglich ist, sollten für jedes vorgegebene Beispiel in der Dokumentation enthalten sein.

Aufgabe 5: Bauernopfer

5.1 Einleitung

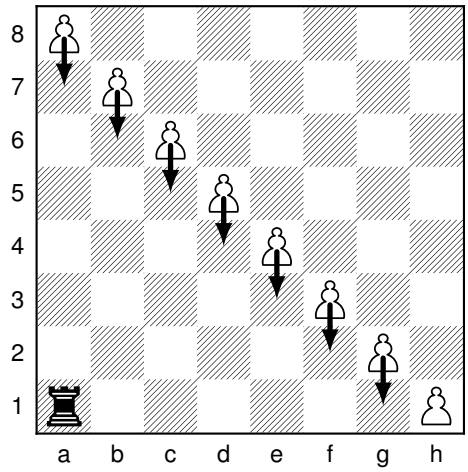
Dieses Schachrätsel verlangt sowohl eine analytische Vorgehensweise als auch Geschick beim Programmieren. Wir wollen herausfinden, ob es einer bestimmten Anzahl (im Weiteren k genannt) von weißen Bauern auf einem Schachfeld möglich ist, einen schwarzen Turm zu fangen. Dabei dürfen die Bauern aber, anders als beim üblichen Schach, in eine beliebige Richtung ziehen.

Anschließend untersuchen wir allgemeinere Varianten des Rätsels: zunächst eine Variante, bei der Weiß jeweils mehrmals (l-mal) hintereinander ziehen darf, und anschließend, was sich ändert, wenn statt eines schwarzen Turms eine schwarze Dame auf dem Spielfeld steht. Zunächst analysieren wir den Fall $k = 8$.

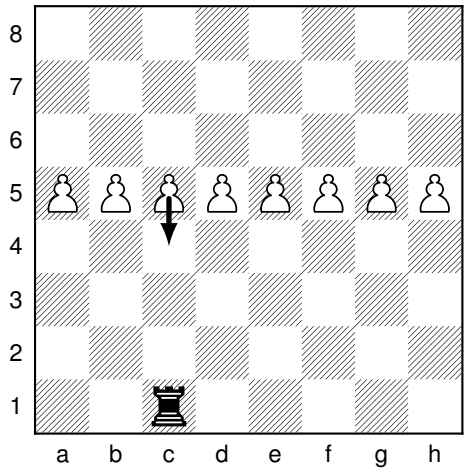
5.2 Aufgabenteil 1: Acht Bauern

Gibt es acht Bauern, kann Weiß mit der richtigen Strategie immer gewinnen. Wir stellen zwei verschiedene gewinnbringende Strategien vor, deren Grundaufstellungen für Weiß wie folgt aussehen (die Position des schwarzen Turms ist nur beispielhaft):

Strategie 1:



Strategie 2:



Der Witz bei beiden Strategien ist, dass das Feld in zwei Teile geteilt wird. Egal welche Startposition Schwarz für seinen Turm wählt: Weiß kann anschließend das Gebiet, in dem der Turm sich aufhält, immer kleiner werden lassen.

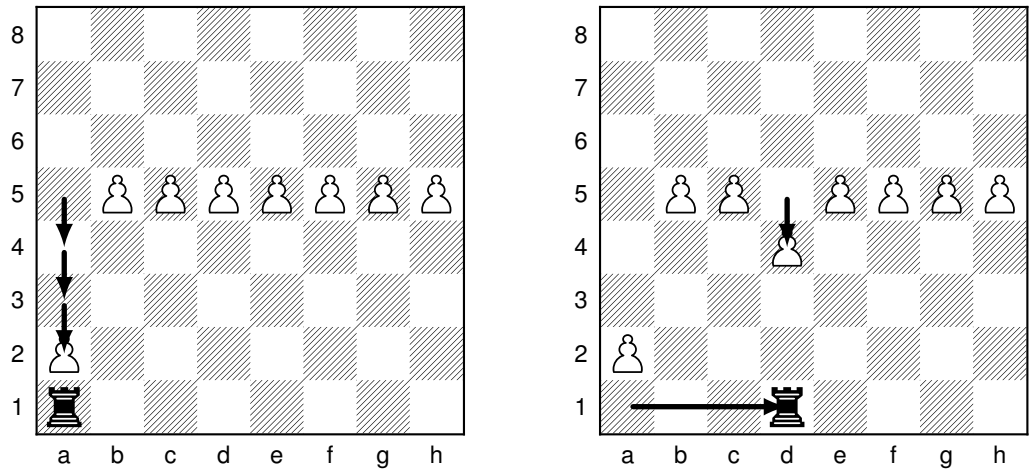
Bei Strategie 1 läuft das wie folgt ab: Angenommen, Schwarz entscheidet sich für eine Position in der unteren linken Hälfte. Dann kann Weiß zunächst den Bauern auf a8 ein Feld nach unten auf a7 ziehen lassen. Im nächsten Zug zieht der Bauer auf b7 ein Feld nach unten, und so weiter, bis g2 nach g1 gezogen wird. Zu keinem Zeitpunkt klaffte in der Mauer eine Lücke, durch die Schwarz hätte entkommen können, und das erreichbare Gebiet von Schwarz ist jetzt deutlich kleiner. Jetzt beginnt Weiß wieder von links mit dem Bauern auf a7 und zieht diesen ein Feld

nach unten, und so weiter, bis der dritte Bauer von rechts von f2 nach f1 zieht. Anschließend fängt Weiß wieder links an und so weiter und so fort.

Irgendwann stehen alle weißen Bauern außer dem auf der a-Linie auf der Grundlinie, und die einzige Position, die dem schwarzen Turm bleibt, ist a1. Nach dem folgenden weißen Zug von a2 auf a1 ist der Turm also spätestens gefangen.

Diese Strategie ist völlig unabhängig vom Bewegungsmuster des schwarzen Turms; zu keinem Zeitpunkt kann er seinem immer kleiner werdenden Gefängnis entkommen. Nach spätestens $7+6+5+4+3+2+1 = 28$ Zügen ist Schwarz „matt“.

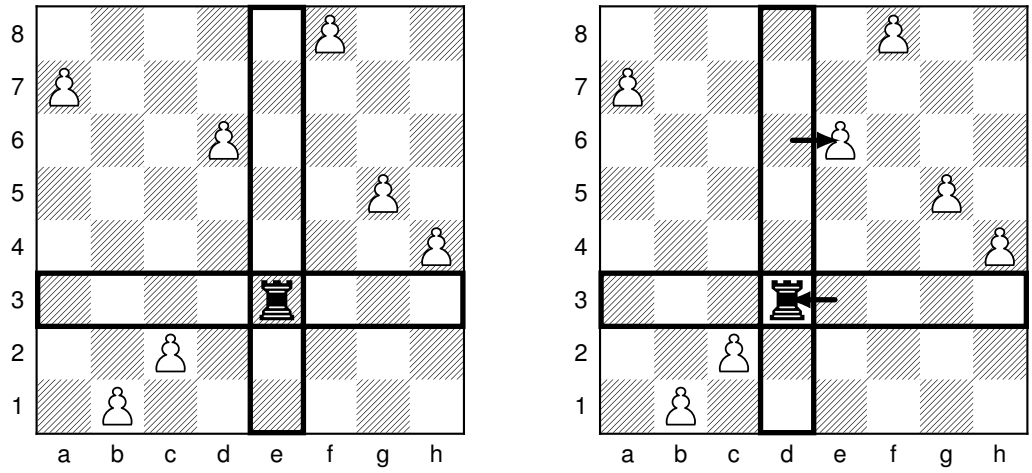
Mit der Aufstellung von Strategie 2 könnte man nach dem gleichen Verfahren vorgehen, also von links nach rechts die weißen Bauern jeweils ein Feld nach unten setzen (falls Schwarz eine Startposition in der unteren Hälfte gewählt hat) bzw. oben (falls der Turm oben steht), und hätte nach spätestens $4 \cdot 8 = 32$ Zügen den Turm gefangen. Allerdings funktioniert hier auch eine „adaptive“ Strategie, also eine, die auf die Züge von Schwarz reagiert: Weiß zieht jeweils den Bauern, der in der gleichen Spalte steht wie der Turm, ein Feld in dessen Richtung. Bei dieser Strategie ist Vorsicht geboten, denn es entstehen Löcher in der Wand aus Bauern, durch die Schwarz scheinbar seinem Gefängnis entfliehen kann. Doch das ist nicht der Fall. Möchte nämlich Schwarz diese Strategie ausnutzen, um sich einen Durchschlupf zu basteln, würde er auf der Grundlinie in einer Spalte (Beispielsweise auf a1) stehen bleiben und den Bauern so nahe wie möglich an sich heran lassen:



Anschließend müsste er um diesen Bauern herumlaufen, z.B. durch einen Zug auf d1. Danach rückt aber der Bauer auf d5 nach d4 und versperrt damit dem Turm den nun nötigen Zug auf d3, da er dort anschließend von dem d-Bauern gefangen würde. Auch in diesem Fall kann Schwarz also nicht entkommen. Die klügste Gegenstrategie bestünde darin, in jeder Spalte genau drei Züge lang zu verharren und den jeweiligen Bauern bis auf ein Feld an ihn herankommen zu lassen. Spätestens wenn alle Bauern in der zweiten Reihe stehen, ist Schwarz aber verloren und somit nach spätestens $3 \cdot 8 + 1 = 25$ Zügen gefangen.

5.3 Aufgabenteil 2: Sieben Bauern

Interessanterweise können sieben Bauern den Turm nicht fangen, wenn dieser sich geschickt verhält. Das geht wie folgt: Zunächst platziert Weiß seine Bauern irgendwie. Da nur sieben Bauern vorhanden sind, gibt es mindestens ein Feld, in dessen Reihe und Spalte kein weißer Bauer zu finden ist. Auf dieses Feld stellt sich der schwarze Turm, wie im folgenden Bild links zu sehen:



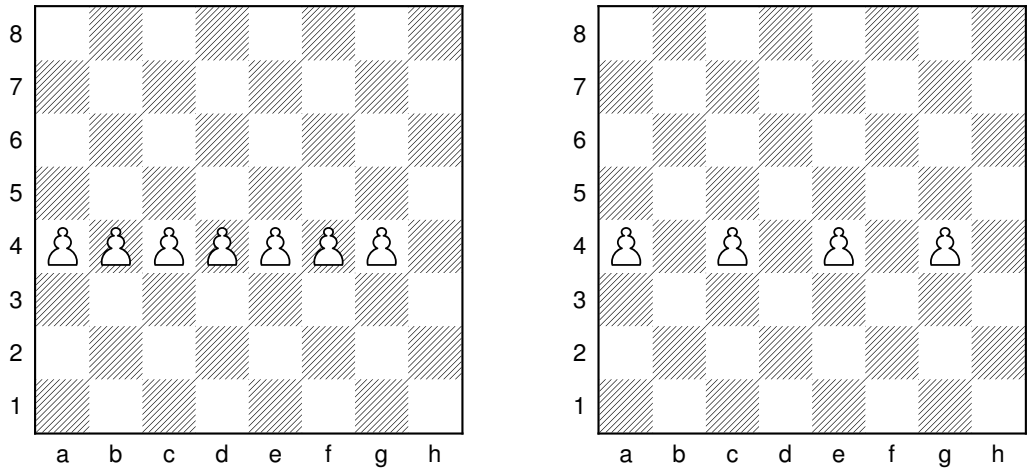
Wenn Weiß den Turm fangen will, muss es früher oder später mit einem Bauern entweder die Reihe oder die Spalte des Turms betreten, kann aber niemals gleichzeitig beides blockieren. Nehmen wir an, ein Bauer betritt die Spalte, in der der Turm steht (wie im Bild rechts). Dann kann der Turm entlang der Reihe jede beliebige andere Spalte erreichen. Da es nur sieben Bauern gibt, gibt es mindestens eine Reihe, in der kein Bauer steht. Der Turm bewegt sich also entlang der leeren Spalte zu dieser Reihe und hat wieder ein Feld wie am Anfang erreicht. Daher kann er niemals gefangen werden.

5.4 Aufgabenteil 3: k Bauern mit l Zügen

Dieser Aufgabenteil ist nicht ganz eindeutig. Laut Aufgabenstellung gibt es „[...] k weiße Bauern, von denen sich in jedem Zug l jeweils einen Schritt bewegen.“ Die Frage ist, ob man in einem Zug denselben Bauern zwei mal bewegen darf, oder ob es l unterschiedliche Bauern sein müssen. Die Formulierung deutet eher auf Letzteres hin, dennoch behandeln wir hier beide Fälle. Wir beginnen mit dem für Weiß günstigeren Fall, dass der gleiche Bauer mehrmals ziehen darf.

Ein Bauer darf mehrmals ziehen

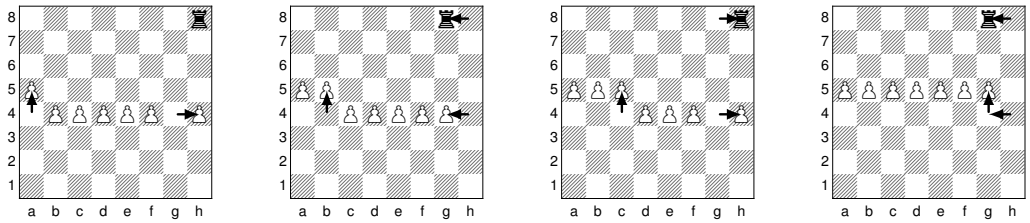
Zwei Züge Wie vorgeschlagen, betrachten wir zunächst den Fall $k = 7$. Den Unterfall $l = 1$ haben wir bereits behandelt: Der Turm gewinnt. Ist jedoch $l = 2$, gewinnt Weiß. Dazu stellt sich Weiß zunächst wie im folgenden Bild links auf:



Damit teilt Weiß das Brett in zwei Teile. Das Ziel ist es, den Teil, in den sich der Turm stellt, systematisch zu verkleinern und dabei zu verhindern, dass der Turm die Seite wechselt.

In der Mauer aus Bauern gibt es eine Lücke. Möchte Schwarz also auf die andere Seite gelangen, muss er zunächst auf die Spalte mit der Lücke ziehen.

Nun ist Weiß an der Reihe und hat zwei Züge. Steht der Turm auf der Spalte ohne Bauern, zieht Weiß mit einem Bauern in den Weg des Turms, um den Seitenwechsel zu verhindern. Anschließend wählt Weiß einen der am weitesten hinten stehenden Bauern aus, um mit diesem einen Schritt entlang der Spalte in Richtung des Turms zu machen. Dadurch verkleinert sich der Bereich des Turms, indem die weiße Mauer unaufhaltsam auf den Turm zu rückt. Das Verfahren wird durch die folgenden Bilder verdeutlicht (Schwarz zieht jeweils zuerst und beginnt mit dem Setzen des Turms):



Der letzte gezeigte Zug verlangt ein mehrfaches Ziehen des gleichen Bauern. Das gleiche Verfahren funktioniert natürlich auch für $l > 2$, also mehr Züge.

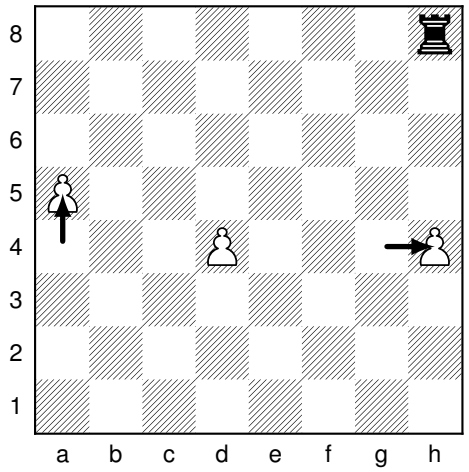
Das gleiche Verfahren funktioniert auch für eine kleinere Anzahl von Bauern. Wichtig ist, dass die Mauer aus Bauern keine Lücke enthält, die nicht von einem der Bauern geschlossen werden kann und anschließend Weiß noch einen Zug übrig hat, um den Bereich des schwarzen Turms zu verkleinern. Die Frage ist, wie viele Bauern mindestens nötig sind, um das zu gewährleisten.

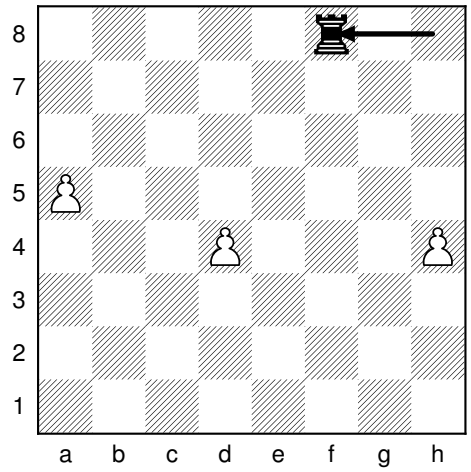
Sind $k = 4$ Bauern vorhanden, können wir die Startaufstellung wie im oberen Bild rechts wählen.

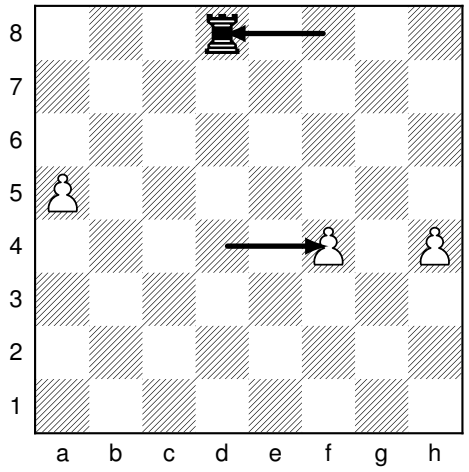
Auch hier gilt: möchte der schwarze Turm auf die andere Seite der Barriere wechseln, muss er in eine Spalte ohne Bauern ziehen. Weiß kann auch mit nur vier Bauern stets mit einem Schritt diese Spalte blockieren und anschließend noch mit einem Bauern vorziehen. Deshalb gewinnt Weiß auch hier.

Was ist mit $k = 3$? Diesmal kann Schwarz gewinnen, indem Weiß gezwungen wird, die zwei zur Verfügung stehenden Züge für die Spaltenblockade zu nutzen.

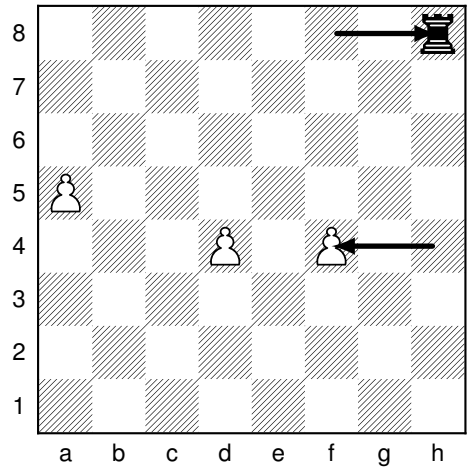
Anschließend hat Weiß zwei Möglichkeiten, die jedoch beide wieder zu einer Situation führen, dass es Spalten gibt, die zwei Züge von den weißen Bauern entfernt sind (Weiß zieht zuerst):







oder



Das heißt, dass Weiß mit diesem Verfahren nicht gewinnen kann. Das ist zwar kein Beweis, dass es kein Verfahren gibt, mit dem Weiß gewinnen kann, aber es ist sehr wahrscheinlich, dass es kein solches gibt.

Mehr als zwei Züge Was ändert sich nun, wenn Weiß mehr als zwei Züge hat? Bei der gleichen Strategie wie eben gewinnt Weiß, wenn niemals Lücken entstehen, die Weiß nicht schließen und anschließend noch vorziehen kann. Das funktioniert genau dann, wenn $l \geq \lceil 8/k \rceil$ Züge zur Verfügung stehen. Die Klammern $\lceil \cdot \rceil$ stehen dabei für „aufrunden“. Damit ergibt sich folgende Siegtabelle:

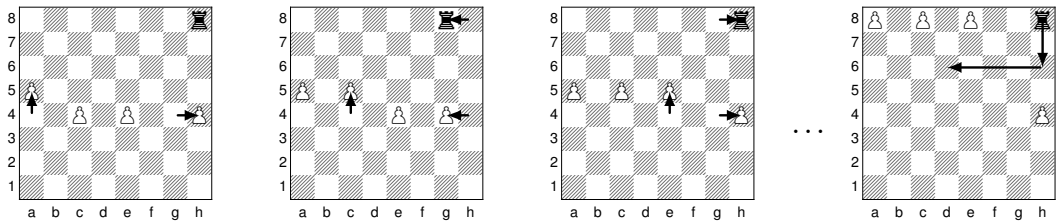
$l \setminus k$	1	2	3	4	5	6	7	8
1								X
2				X	X	X	X	X
3			X	X	X	X	X	X
4		X	X	X	X	X	X	X
5		X	X	X	X	X	X	X
6		X	X	X	X	X	X	X
7		X	X	X	X	X	X	X
8	X	X	X	X	X	X	X	X

Tabelle 3: Tabelle, in welchen Fällen von Bauernzahl k und Zugzahl l Weiß gewinnt, falls Weiß den gleichen Bauern mehrmals ziehen darf. X steht für „Weiß gewinnt“, ein leeres Feld für „Schwarz gewinnt“.

Jeder Bauer darf höchstens einmal ziehen

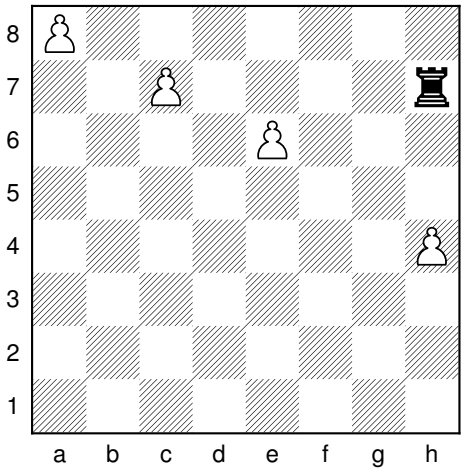
Es bleibt noch zu prüfen, was sich ändert, wenn ein Bauer nur jeweils einmal ziehen darf. Es ist klar, dass dies die Chancen von Weiß auf einen Sieg verringert, weil weniger Züge möglich sind. Kann Schwarz das ausnutzen? Außerdem müssen wir weniger Fälle prüfen, denn in dieser Variante ergeben Konstellationen mit $l > k$, also mehr Bauernzügen als verfügbaren Bauern, keinen Sinn.

Betrachten wir den Fall $l = 2$ und $k = 4$. Das war die kleinste Anzahl von Bauern im vorherigen Fall, mit der Weiß noch gewinnen konnte. Eben haben wir jeweils einen Bauern genutzt, um die Spalte von Schwarz zu blockieren und anschließend die Mauer Stück für Stück nach vorne gesetzt. Was aber passiert, wenn Schwarz immer den gleichen Bauern zwingt, ihm den Weg zu versperren? Dann kann dieser nicht im gleichen Zug nach vorne ziehen. Dadurch bricht die Mauer auseinander (Schwarz zieht zuerst):



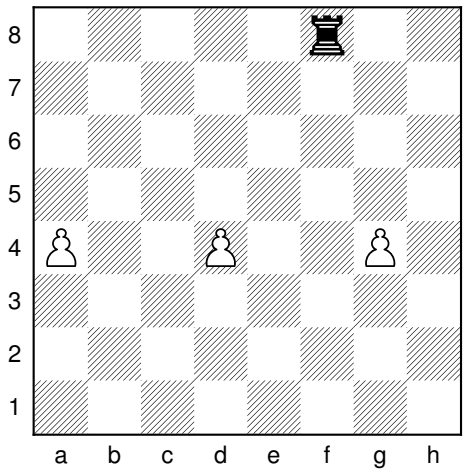
So also geht die Taktik von Weiß nicht auf. Kann Weiß also die Strategie anpassen, um dennoch zu gewinnen? Tatsächlich ist das möglich. Weiß kann seine Bauern dennoch vorziehen, muss aber darauf achten, dass zwei benachbarte Bauern höchstens eine freie Reihe zwischen sich haben. Sollte ein Bauer dabei die Reihe des schwarzen Turms erreichen und kein anderer Bauer in der gleichen Reihe stehen, sollte dieser sie aber nicht überschreiten. Dann ist noch immer sichergestellt, dass Schwarz auch nicht waagerecht entfleuchen kann. Wann immer Schwarz auf eine Reihe zieht, die leer ist, kann Weiß einen Bauern auf die gleiche Reihe ziehen und mit dem zweiten Schritt mit dem Bauern, der in der gleichen Spalte steht, einen Schritt auf den Turm zu machen. Wendet man diese Regeln an, sieht die Spielsituation schließlich wie im nächsten Bild aus.

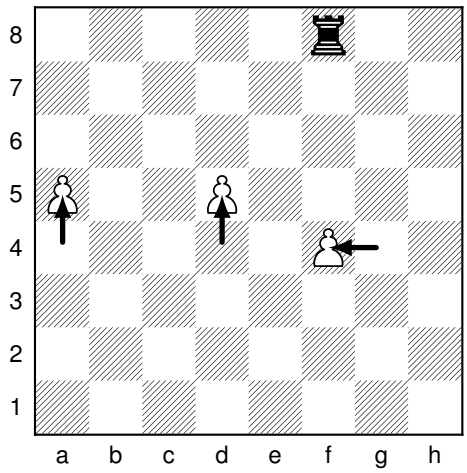
Übrigens gehen wir gerade davon aus, dass Schwarz ständig den gleichen Zug wiederholt und damit den immer gleichen Bauern fesselt. Das ist auch gerechtfertigt, denn wann immer Schwarz von dieser Taktik abweichen würde, könnten wir ihn mit einem anderen Bauern blockieren und den bisher gefesselten Bauern vorsetzen. Das wäre zum Nachteil von Schwarz.

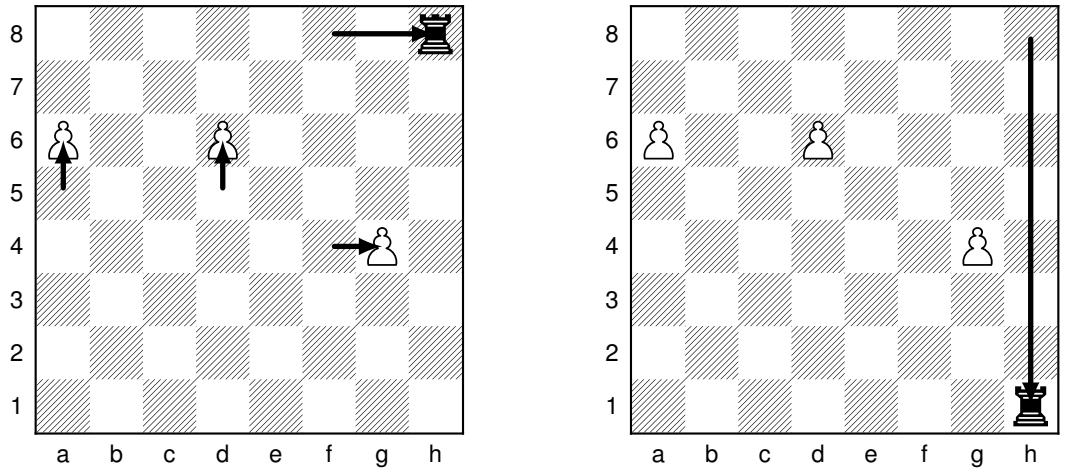


Beharrt Schwarz jetzt weiter auf seiner Taktik, können wir jetzt die links vom Turm stehenden Bauern Schritt für Schritt nach rechts auf diesen zu ziehen. Zwar gibt es eine Reihe, die von keinem der Bauern bewacht wird. Entscheidet sich Schwarz aber, auf diese zu ziehen, bleibt er in der gleichen Spalte und Weiß kann den Spaltenbauern vorziehen. So wird der Spielraum des Turms immer kleiner und er verliert schließlich.

Dies ist zugegebener Maßen eine recht komplizierte Taktik, und es ist nicht ausgeschlossen, dass es einfachere gibt. Da dieses Verfahren offenbar auch für $l = 3$ und $l = 4$ funktioniert, müssen wir jetzt nur noch den Fall $l = 3$ und $k = 3$ untersuchen (denn $l > k$, also mehr verfügbare Züge als Bauern, funktioniert nicht). In dem vorherigen Fall der mehrfach ziehenden Bauern reichte dies aus, um Schwarz zu schlagen. Das ist hier jetzt anders. Stellen wir die drei Bauern möglichst gleichmäßig verteilt in der Mitte auf, haben diese zwar einen Abstand von höchstens zwei Feldern zueinander. Schwarz kann aber schnell dafür sorgen, dass sich dieser Abstand auf drei vergrößert und damit eine Lücke bleibt, durch die Schwarz entfliehen kann:







Das deutet stark darauf hin, dass es in diesem Fall keine Gewinnstrategie für Weiß gibt. Alles in allem ergibt sich also folgende Siegtabelle für den Fall, dass jeder Bauer nur einmal ziehen darf:

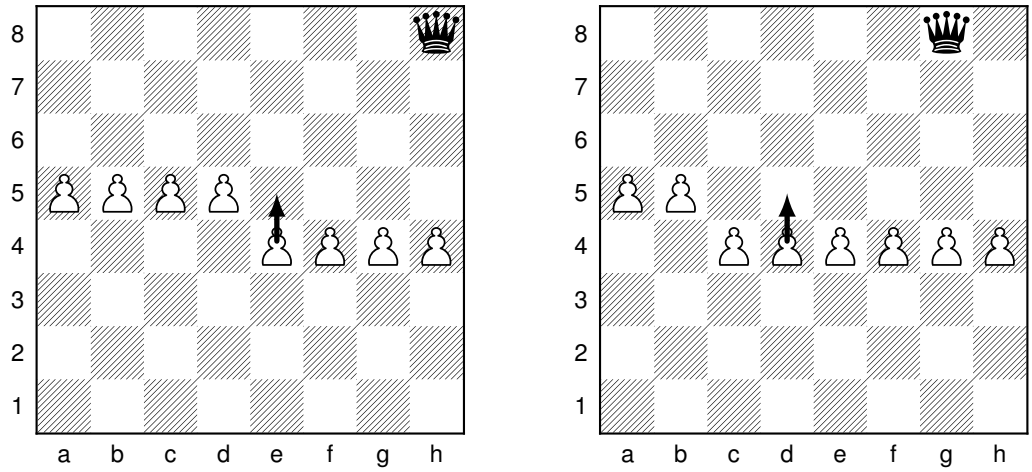
$l \setminus k$	1	2	3	4	5	6	7	8
1								X
2	·			X	X	X	X	X
3	·	·		X	X	X	X	X
4	·	·	·	X	X	X	X	X
5	·	·	·	·	X	X	X	X
6	·	·	·	·	·	X	X	X
7	·	·	·	·	·	·	X	X
8	·	·	·	·	·	·	·	X

Tabelle 5: Tabelle, in welchen Fällen von Bauernzahl k und Zugzahl l Weiß gewinnt, falls jeder Bauer nur einmal ziehen darf. X steht für „Weiß gewinnt“, ein leeres Feld für „Schwarz gewinnt“ und · für nicht zulässige Fälle, da $l > k$.

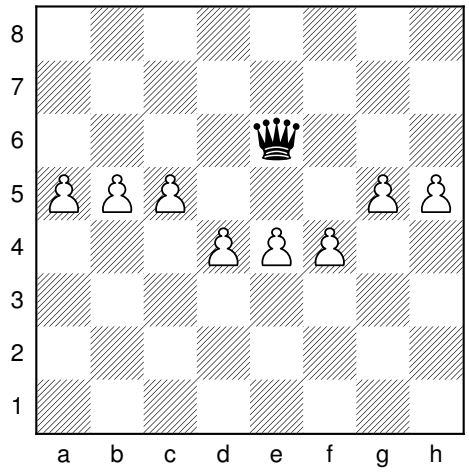
5.5 Aufgabenteil 4: Varianten

Es sind viele Erweiterungen denkbar. Man kann sowohl die Zugregeln für Weiß als auch die für Schwarz auf fast beliebige Art und Weise anpassen. Wir untersuchen hier nur die vorgeschlagene Variante, dass der schwarze Turm durch eine Dame ersetzt wird, für den einfachsten Fall mit $k = 8$ und $l = 1$. Kann Schwarz jetzt dank der zusätzlichen Zugmöglichkeiten entkommen? Die Antwort ist: nein.

Weiß stellt die Bauern in einer Reihe auf und zieht die Bauern Stück für Stück von links nach rechts nach vorne. Schwarz kann nur dann entkommen, wenn die Dame droht, schräg durch eine Bruchstelle der Mauer zu ziehen. Das kann Weiß jedoch stets verhindern, indem der entsprechende Bauer neben dem Knick vorgesetzt wird, siehe folgende Situation links:



Eine andere Möglichkeit, die Seite zu wechseln, besteht darin, sich auf ein Feld zu setzen, das auf einer Diagonalen mit einer künftigen Bruchstelle der Mauer liegt. Dann muss Weiß von der Strategie, die Mauer von links nach rechts vorzusetzen, abweichen und eine weitere Bruchstelle einfügen, indem der nächste Bauer zuerst vorgesetzt wird (oben rechts zu sehen). Kann Schwarz das nutzen, um gleichzeitig zwei Bruchstellen zu bedrohen? In diesem speziellen Fall von einem 8×8-Schachbrett nicht. Die einzige Erfolg versprechende Konfiguration für die Dame ist folgende:



Diese Stellung kann von Weiß aber verhindert werden: Wenn Weiß gezwungen wird, eine weitere Bruchstelle einzubauen, wird der nächstmögliche Bauer gewählt, die Bruchstellen sind also höchstens zwei Spalten voneinander entfernt.

5.6 Bewertungskriterien

- Aufgabenteil 1 (acht Bauern, ein Turm): Ein korrektes Verfahren zum Fangen des Turms und eine möglichst erfolgreiche Gegenstrategie für Schwarz soll nachvollziehbar angegeben sein.

- Aufgabenteil 2 (sieben Bauern, ein Turm): Es soll erkannt werden, dass Schwarz gewinnen kann. Die angegebene Strategie soll nachvollziehbar erläutert sein.
- Aufgabenteile 1 und 2: Die Treibjagden sollen korrekt, in Übereinstimmung mit der vorherigen Beschreibung, implementiert und visualisiert sein. Textuelle Ausgaben genügen als „Visualisierung“, wenn sie einigermaßen nachvollziehbar sind.
- Aufgabenteil 3: Es soll für mindestens zwei $k < 8$ mindestens ein $l > 1$ angegeben werden, für das eine Gewinnstrategie für Weiß existiert. Bei der Angabe der Strategie ist nicht unbedingt die Korrektheit, sondern vielmehr die Argumentation entscheidend. Gemeint war in der Aufgabenstellung übrigens die Variante, dass l unterschiedliche Bauern ziehen; die Variante, dass beliebige Bauern l-mal ziehen dürfen, ist aber auch akzeptabel.
- Aufgabenteil 4: Eine weitere, hinreichend komplexe Variante des Spiels soll für mindestens einen Fall k und l untersucht werden (sofern eine Angabe von k und l im geänderten Regelsatz Sinn ergibt).

Aus den Einsendungen: Perlen der Informatik

Allgemeines

Worte des Wettbewerbs: Backtrakigalgorismus, Untersuchtung, Funktionsgraf

Codeschnipsel: `if x = (1 or 2):`

Zuerst liest das Programm die vorgegebene Datei ein und liest sie aus.

Meine Lösungsidee besteht darin, nach der Lösung mit einer rekursiven Suche zu suchen.

Wie man an dem Diagramm erkennen kann, *Und hier endet die Einsendung.*

Wir waren zu faul, noch eine billige GUI hinzuzaubern, die das Bedienen sowieso nicht erleichtern würde.

Stichproben ergaben keine Fehler. *Und das soll uns jetzt überzeugen, oder was?*

Dieser Abschnitt ist eine verbale Zusammenfassung der oben aufgeführten ausführlichen algorithmischen Darstellung. Er ersetzt nicht das Lesen der vorigen Ausführungen, bringt aber auch keinen Mehrwert gegenüber selbiger.

Die einfachste Möglichkeit, diese Aufgabe zu lösen, war uns zu effizient und zu einfach umzusetzen, ...

Dann ist mir die Idee gekommen, mithilfe des Knotens selbst den Knoten zu lösen.

Je nach Implementierung liegt die Laufzeit in $\mathcal{O}(n+m)$.

Bücherregal

`for each in books: ...`

... dabei liest das Programm als erstes die Anzahl der Figuren, dann die Anzahl der Figuren und als letztes die Höhen der einzelnen Bücher ... *Doppelt hält eben besser.*

Trotzdem habe ich mich dafür entschieden, möglichst viele Bücher weiterhin in das Regal einzusortieren, einfach weil es geht.

Das habe ich mithilfe einer sehr komplexen if-Abfrage gemacht. Leider hängt das Programm immer. Ich weiß, dass es an der komplizierten if-Abfrage liegt.

Wintervorrat

Bitte geben sie die Anzahl der Zeilen in dem von ihnen getesteten Wald als Dezimalzahl an.

Vögel sind ziemlich einfache Wesen – sie können nur fliegen und sonst nichts.

Die Liste ist nach der Dauer des Grabens sortiert, so dass das Eichhörnchen sich je nach Risikobedürfnis das entsprechende Feld heraussuchen kann.

Zimmerbelegung

Einzellzimmer

Die Zimmer-Klasse enthält die Liste der Einwohner und eine Hassliste, welche eine Zusammenfügung der Hasslisten der Einwohner ist.

Am Beispiel Zimmerbelegung kann man sehr gut die unsoziale Seite des Algorithmus sehen: Alle Personen, die auf keiner Positivliste stehen, sind alleine in einem Zimmer.

Entweder benötigt Marie dringend Therapie, oder, was wahrscheinlicher ist, sie hat mit Absicht oder aus Versehen eine Falscheingabe gemacht.

Zusätzlich werden Warnungen für narzisstische Schüler (solche, die mit sich selbst auf ein Zimmer wollen) und selbsthassende Schüler (solche, die nicht mit sich selbst auf ein Zimmer wollen) ausgegeben.

Die nicht in Zimmern befindlichen Schülerinnen können gut in einer ArrayList gelagert werden.

. . . wäre es theoretisch möglich, alle Schülerinnen in einem Zimmer unterzubringen. Da aber so viele Mädchen und nur ein Bad zu zeitlichen Problemen führen könnte, gibt unser Algorithmus klugerweise die maximal mögliche Zimmeranzahl aus.

Im Folgenden wird der Wunsch, mit einer Person auf ein Zimmer zu kommen, „Freundschaft“ genannt (auch wenn diese traurigerweise einseitig sein kann).

Ein Haufen Objekte (Die Mädchen) . . .

Schwimmbad

Leider können keine Beispiele dokumentiert werden, da ich mein Programm nicht mehr öffnen [. . .] kann.

Wenn das Schwimmbad betreten werden darf, sind die kleinen Kinder irrelevant, da kostenlos.

Dies würde so ablaufen, dass erstmal alle Gutscheine außer einem für die Entfernung der Personen benutzt werden.

In den angegebenen Preisen werden im Folgenden die Euro-Zeichen weggelassen. Dies dient dem Schutz der Umwelt.

Unter der Annahme [. . .], dass ihre Mutter älter als 16 ist (Begründung biologisch möglich, allerdings kann ich nicht ins Detail gehen, da es 23:10 am Abgabetag ist) . . .

Dreiecke zählen

Der Schnittpunkt wird durch zwei Muttergeraden bestimmt.

Der Test bestätigt, dass das Programm nicht funktioniert.

Falls nein, wird mittels einer Typkonvertierung zum bool getestet, ob das Dreieck existieren kann.

Auto-Scrabble

Aber interessant, dass jemand überhaupt auf die Idee kam, ein Gesetz mit einem solch überlangen Titel vorzuschlagen.

Wurde kein Kennzeichen gefunden, gibt das Programm auch diese aus.

Tatsächlich existiert Y als Kennzeichen, und zwar für Autos der Bundeswehr. Dies kann in Quelle [2] nachgelesen werden.

Bauernopfer

Leider keine Perlen. Bauern zu opfern ist offensichtlich eine ernste Angelegenheit.