42. Bundeswettbewerb Informatik 2023/2024, 1. Runde

Lösungshinweise

Allgemeines

Es ist sehr erfreulich, dass wieder sehr viele die Aufgaben bearbeitet und am Bundeswettbewerb und Jugendwettbewerb Informatik teilgenommen haben. Uns von BWINF ist bewusst, dass in der Regel viel Arbeit hinter der Erstellung einer Einsendung steckt.

Diese Lösungshinweise enthalten für jede Aufgabe mögliche Lösungsideen, Ausgaben zu den vorgegebenen Beispielen sowie eine Auflistung und Erläuterung der Bewertungskriterien. Bei der Anwendung dieser Kriterien wurden die in den Einsendungen gezeigten Leistungen so gut wie möglich gewürdigt. Das war nicht immer leicht, insbesondere wenn die Dokumentation nicht die im Aufgabenblatt genannten Anforderungen erfüllt. Deshalb soll, vor der Beschreibung der Lösungsideen, im folgenden Abschnitt auf die Dokumentation näher eingegangen werden; vielleicht helfen diese Anmerkungen bei der nächsten Teilnahme.

Wie auch immer die Bewertung einer Einsendung ausfällt, sie sollte nicht entmutigen! Allein durch die Arbeit an den Aufgaben und ihren Lösungen hat jede Teilnehmerin und jeder Teilnehmer einiges gelernt; diesen Effekt sollte man nicht unterschätzen.

Die Bearbeitungszeit für die 1. Runde beträgt über zweieinhalb Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der Dokumentation zu vermeiden. Aufgaben sind gelegentlich schwerer zu bearbeiten, als es auf den ersten Blick erscheinen mag. Erst bei der konkreten Umsetzung einer Lösungsidee oder beim Testen von Beispielen kann man auf Besonderheiten oder Schwierigkeiten stoßen, die zusätzlicher Zeit bedürfen.

Noch etwas Organisatorisches: Sollte der Name auf Urkunde oder Teilnahmebescheinigung falsch geschrieben sein, ist er auch im AMS (login.bwinf.de) falsch eingetragen. Die Teilnahmeunterlagen können gerne neu angefordert werden; dann aber bitte vorher den Eintrag im AMS korrigieren.

Dokumentation

Die Zeit für die Bewertung ist leider begrenzt, weshalb es unmöglich ist, alle eingesandten Programme gründlich zu testen. Die Grundlage der Bewertung ist deshalb die Dokumentation, die, wie im Aufgabenblatt beschrieben, für jede bearbeitete Aufgabe aus den Teilen *Lösungsidee*, *Umsetzung*, *Beispielen* und *Quellcode* bestehen soll. Leider sind die Dokumentationen bei vielen Einsendungen sehr knapp ausgefallen, was oft zu Punktabzügen führte, die das Erreichen der 2. Runde verhinderten. Grundsätzlich kann die Dokumentation einer Aufgabe als „sehr unverständlich oder nicht vollständig“ bewertet werden, wenn die schriftliche Darstellung kaum verständlich ist oder Teile wie Lösungsidee, Umsetzung oder Quellcode komplett fehlen.

Die Beschreibung der *Lösungsidee* sollte keine Bedienungsanleitung des Programms oder eine Wiederholung der Aufgabenstellung sein. Stattdessen soll erläutert werden, welches fachliche Problem hinter der Aufgabe steckt und wie dieses Problem grundsätzlich mit einem Algorithmus sowie Datenstrukturen angegangen wurde. Ein einfacher Grundsatz ist, dass Bezeichner von Programmelementen wie Variablen, Methoden etc. nicht in der Beschreibung einer Lösungsidee verwendet werden, da sie unabhängig von solchen technischen Realisierungsdetails ist. Wenn die Beschreibungen in der Dokumentation nicht auf die Lösungsidee eingehen oder bzgl. der Lösungsidee kaum nachvollziehbar sind, kann es einen Punktabzug geben, weil das „Verfahren unzureichend begründet bzw. schlecht nachvollziehbar" ist.

Besonders wichtig sind die vorgegebenen (und ggf. weitere) *Beispiele*. Wenn Beispiele und die zugehörigen Ergebnisse in der Dokumentation ganz oder teilweise fehlen, führt das zu Punktabzug. Bei der Erläuterung der Bewertungskriterien ist für jede Aufgabe genannt, zu wie vielen (und teils auch zu welchen) Beispielen Programmausgaben oder Ergebnisse in der Dokumentation erwartet werden. Diese Ergebnisse sollten idealerweise korrekt sein. Punktabzug für falsche Ergebnisse gibt es aber nur, wenn nicht schon für den ursächlichen Fehler ein Punkt abgezogen wurde.

Es ist nicht ausreichend, Beispiele nur in gesonderten Dateien abzugeben, ins Programm einzubauen oder den Bewerterinnen und Bewertern sogar das Erfinden und Testen geeigneter Beispiele zu überlassen. Beispiele sollen die Korrektheit der Lösung belegen und auch zeigen, wie das Programm mit Sonderfällen umgeht.

Auch *Quellcode*, zumindest dessen für die Lösung wichtige Teile, gehört in die Dokumentation; Quellcode soll also nicht nur elektronisch als Code-Dateien (als Teil der Implementierung) der Einsendung beigefügt werden. Schließlich gehören zu einer Einsendung als Teil der Implementierung *Programme*, die durch die genaue Angabe der Programmiersprache und des verwendeten Interpreters bzw. Compilers möglichst problemlos lauffähig sind und hierfür bereits fertig (interpretierbar/kompiliert) vorliegen. Die Programme sollten vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner hinsichtlich ihrer Lauffähigkeit getestet werden.

Hilfreich ist oft, wenn die Erstellung der Dokumentation die Programmierarbeit begleitet oder ihr teilweise sogar vorangeht. Wer es nicht hinreichend schafft, seine Lösungsidee für Dritte verständlich zu formulieren, dem gelingt meist auch keine fehlerlose Implementierung, egal in welcher Programmiersprache. Daher kann es nützlich sein, von Bekannten die Dokumentation auf Verständlichkeit hin lesen zu lassen, selbst wenn sie nicht vom Fach sind.

Wer sich für die 2. Runde qualifiziert hat, beachte bitte, dass dort deutlich kritischer als in der 1. Runde bewertet wird und höhere Anforderungen an die Qualität der Dokumentationen und Programme gestellt werden. So werden in der 2. Runde unter anderem eine klar beschriebene Lösungsidee (mit geeigneten Laufzeitüberlegungen und nachvollziehbaren Begründungen), übersichtlicher Programmcode (mit verständlicher Kommentierung), genügend aussagekräftige Beispiele (zum Testen des Programms) und ggf. spannende eigene Erweiterungen der Aufgabenstellung (für zusätzliche Punkte) erwartet.

Bewertung

Bei den im Folgenden beschriebenen Lösungsideen handelt es sich um Vorschläge, aber sicher nicht um die einzigen Lösungswege, die korrekt sind. Alle Ansätze, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind, werden in der Regel akzeptiert. Einige

Dinge gibt es allerdings, die – unabhängig vom gewählten Lösungsweg – auf jeden Fall erwartet werden. Zu jeder Aufgabe werden deshalb im jeweils letzten Abschnitt die Kriterien näher erläutert, auf die bei der Bewertung dieser Aufgabe besonders geachtet wurde. Die Kriterien sind in der Bewertung, die man im AMS einsehen kann, aufgelistet. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation, die oben erklärt sind.

In der 1. Runde geht die Bewertung von fünf Punkten pro Aufgabe aus, von denen bei Mängeln Punkte abgezogen werden. Da es nur Punktabzüge gibt, sind die Bewertungskriterien meist negativ formuliert. Wenn das (Negativ-)Kriterium erfüllt ist, gibt es einen Punkt Abzug; ansonsten ist die Bearbeitung in Bezug auf dieses Kriterium in Ordnung. Wurde die Aufgabe insgesamt nur unzureichend bearbeitet, wird ein gesondertes Kriterium angewandt, bei dem es bis zu fünf Punkte Abzug geben kann. Im schlechtesten Fall wird eine Aufgabenbearbeitung mit 0 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 2. Runde qualifiziert. Kritische Fälle mit nur 11 Punkten sind bereits sehr gründlich und mit viel Wohlwollen geprüft. Leider ließ sich nicht verhindern, dass einige Teilnehmende nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden; dies ist ziemlich riskant, da sich leicht Fehler einschleichen.

Auch wurde in einigen Fällen die Regelung zur Bearbeitung von Junioraufgaben als Teil einer Einsendung zum Bundeswettbewerb Informatik nicht beachtet. Hierzu ein Zitat aus dem Mantelbogen des Aufgabenblatts: „Die etwas leichteren Junioraufgaben dürfen nur von SchülerInnen vor der Qualifikationsphase des Abiturs bearbeitet werden.“ Nur unter Einhaltung dieser Bedingung können Bearbeitungen von Junioraufgaben im Bundeswettbewerb gewertet werden.

Danksagung

Alle Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt: Peter Rossmanith (Vorsitz), Hanno Baehr, Jens Gallenbacher, Rainer Gemulla, Torben Hagerup, Christof Hanke, Thomas Kesselheim, Arno Pasternak, Holger Schlingloff, Melanie Schmidt sowie (als Gäste) Wolfgang Pohl und Hannah Rauterberg.

An der Erstellung der in diesem Dokument skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Katharina Bade (Junioraufgabe 1 und 2), Robert Czechowski (Aufgabe 1), Johann Gaulke (Aufgabe 2 und 5), Gabriel Dengler (Aufgabe 4) und Christian Schultze. Allen Beteiligten sei für ihre Mitarbeit ganz herzlich gedankt.

Junioraufgabe 1: Wundertüte

J1.1 Lösungsidee 1: Reihum verteilen

Wir haben k verschiedene Spielesorten und von jeder Spielesorte eine bestimmte Anzahl an Spielen. Alle Spiele sollen nun möglichst gleichmäßig auf n Tüten aufgeteilt werden. Für die Gleichmäßigkeit sollen zwei Kriterien beachtet werden:

1. Der Unterschied zwischen den Gesamtzahlen der Spiele in den Tüten soll höchstens 1 sein.
2. Für jede Spielesorte soll der Unterschied zwischen den Anzahlen von Spielen dieser Sorte in den Tüten ebenfalls höchstens 1 sein.

Um diese Aufgabe zu erfüllen, können wir die Spiele nacheinander auf die Tüten aufteilen. Wir gehen also alle Tüten durch und packen jeweils ein Spiel in jede Tüte. Nachdem wir bei der letzten Tüte angekommen sind, fangen wir wieder bei der ersten Tüte an und gehen wieder alle Tüten durch und so weiter. Wenn wir alle Spiele der ersten Sorte verteilt haben, dann machen wir mit der nächsten Spielesorte weiter, bei der Tüte, die als nächstes in der Reihe dran ist. Denn wenn wir mit einer neuen Spielesorte immer wieder bei der ersten Tüte anfangen würden, dann wird in Tüte 1 definitiv jedes Mal ein Spiel hinein gefüllt, während für die letzte Tüte nicht immer noch ein Spiel vorhanden ist.

Betrachten wir das Beispiel vom Aufgabenblatt: Wir haben vier Kartenspiele, vier Würfelspiele und zwei Geschicklichkeitsspiele.

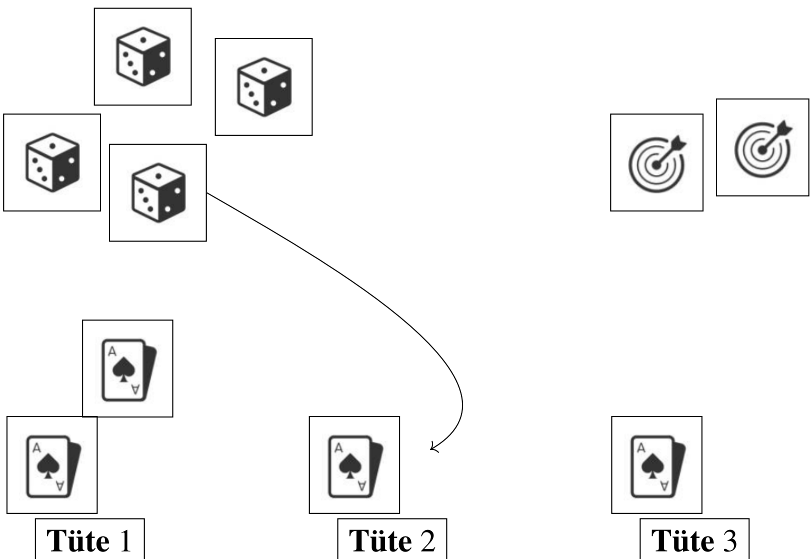


Abbildung J1.1: Wir haben zunächst in Tüte 1, Tüte 2 und Tüte 3 ein Kartenspiel gepackt. Da wir insgesamt vier Kartenspiele haben, packen wir in Tüte 1 ein weiteres Kartenspiel. Nun sind die Kartenspiele verteilt und wir machen mit den Würfelspielen weiter. Da wir als letztes in Tüte 1 ein Kartenspiel gepackt haben, packen wir das erste Würfelspiel in Tüte 2.

Wir machen weiter, bis alle Spiele verteilt sind. Die Gesamtzahlen der Spiele in den Tüten unterscheiden sich nun garantiert maximal um 1, da niemals innerhalb einer Runde mehr als ein Spiel in eine Tüte gepackt wird. Auch haben wir sicher gestellt, dass Bedingung (2) erfüllt ist und die einzelnen Sorten möglichst gleichmäßig verteilt sind.

Laufzeit Mit dem oben beschriebenen Verfahren müssen wir jedes Spiel genau einmal betrachten und einer Tüte zuweisen. Die Laufzeit für das Packen der Tüten lässt sich also mit $\mathcal{O}(x)$ beschreiben, wobei x die Gesamtanzahl der Spiele ist. Die Laufzeit verhält sich also linear zur Anzahl der Spiele.

J1.2 Lösungsidee 2: Verteilung ausrechnen

Anstatt die Spiele reihum einzeln zu verteilen, können wir auch mehrere Spiele gleichzeitig in die Tüten packen. Dazu ist es zunächst einmal sinnvoll, das Problem mathematisch abzubilden: Wenn wir eine Anzahl x von Spielen – egal ob eine bestimmte Sorte oder generell alle Spiele – gleichmäßig auf n Tüten aufteilen möchten, dann packen wir in jede Tüte erst einmal gleich viele Spiele, bis ein Rest bleibt und wir nicht mehr genügend Spiele haben, um eines in jede Tüte zu packen.

Jede Tüte beinhaltet dann bereits $x : n = g$ Spiele. Dabei ist $x : n$ die ganzzahlige Division, denn natürlich können wir nur ganze Spiele auf Tüten verteilen; sie steht in Programmiersprachen über spezielle Operatoren wie `//` (Python) zur Verfügung oder über die Anwendung des Divisionsoperators `/` auf zwei Integer-Werte. Außerdem benötigen wir den Rest der Division und verwenden dazu die Modulo-Operation, die in vielen Programmiersprachen mit dem Operator `mod` realisiert ist. Dieser liefert als Ergebnis genau den Rest der ganzzahligen Division und lässt sich identisch zum : benutzen, nämlich mit $x \mod n$. Um die Aufteilung möglichst gleichmäßig zu halten, verteilen wir den Rest Tüte für Tüte, wie wir es vorher getan haben.

Mit diesem Wissen können wir pro Spielesorte einmal alle Tüten durchlaufen und speichern, wie viele Spiele dieser Sorte in diese Tüte gefüllt werden. Indem wir für die Verteilung des Rests speichern, welches die Starttüte ist, in die wir beginnen den Rest zu füllen, können wir auch Bedingung (2) erfüllen.

Haben wir beispielsweise vier Tüten und drei verschiedene Spielesorten, könnte der Algorithmus wie folgt ablaufen:

Von Spielesorte 1 gibt es sieben Spiele. Somit können wir einmal in jede Tüte genau $7 : 4 = 1$ Spiel tun. Dann bleibt ein Rest von $7 \mod 4 = 3$. Diesen Rest füllen wir beginnend bei Tüte 1 in die Tüten. Wir erhalten nun vorerst diese Aufteilung:

Tüten	Spielesorte 1	Spielesorte 2	Spielesorte 3	Gesamtanzahl Spiele
Tüte 1	2	0	0	2
Tüte 2	2	0	0	2
Tüte 3	2	0	0	2
Tüte 4	1	0	0	1

Von Spielesorte 2 gibt es nun 12 Spiele. Diese lassen sich ohne Rest auf die Tüten aufteilen $12 : 4 = 3$. Wir erhalten folgende Aufteilung:

Tüten	Spielesorte 1	Spielesorte 2	Spielesorte 3	Gesamtanzahl Spiele
Tüte 1	2	3	0	5
Tüte 2	2	3	0	5
Tüte 3	2	3	0	5
Tüte 4	1	3	0	4

Von Spielesorte 3 gibt es 19 Spiele. 16 Spiele lassen sich gleichmäßig auf die vier Tüten aufteilen. Es bleibt erneut ein Rest von 3. Die beginnen wir jetzt allerdings nicht bei Tüte 1 zu verteilen, sondern bei Tüte 4: Das ist die erste Tüte, die bei der letzten Aufteilung vom Rest

kein weiteres Spiel bekommen hat. Damit stellen wir sicher, dass auch der Unterschied in der Gesamtzahl der Spiele möglichst gering bleibt. Die restlichen Spiele kommen also in die Tüten 4, 1 und 2. Unsere finale Aufteilung ist damit die folgende:

Tüten	Spielesorte 1	Spielesorte 2	Spielesorte 3	Gesamtanzahl Spiele
Tüte 1	2	3	5	10
Tüte 2	2	3	5	10
Tüte 3	2	3	4	9
Tüte 4	1	3	5	9

Laufzeit Bei diesem Verfahren betrachten wir für jede Spielesorte die einzelnen Tüten höchstens zweimal. Die Laufzeit können wir deshalb mit $\mathcal{O}(k * n)$ beschreiben, wobei k die Anzahl der Spielesorten und n die Anzahl der Tüten ist. (Bei einer Beschreibung der Laufzeit mit dem Operator $\mathcal{O}$ spielen konstante Faktoren keine Rolle; deshalb nur n und nicht $2n$.)

J1.3 Beispiele

`wundertuete0.txt`

Tüten	Spiel 1	Spiel 2	Spiel 3	Gesamt
Tüte 1	2	1	1	4
Tüte 2	1	2	0	3
Tüte 3	1	1	1	3

`wundertuete1.txt`

Tüten	Spiel 1	Spiel 2	Spiel 3	Gesamt
Tüte 1	3	1	2	6
Tüte 2	3	1	2	6
Tüte 3	3	1	2	6
Tüte 4	3	1	2	6
Tüte 5	3	1	2	6
Tüte 6	3	1	2	6

`wundertuete2.txt`

Tüten	Spiel 1	Spiel 2	Spiel 3	Spiel 4	Gesamt
Tüte 1	2	1	0	0	3
Tüte 2	1	1	1	0	3
Tüte 3	1	1	1	0	3
Tüte 4	1	1	1	0	3
Tüte 5	1	1	0	1	3
Tüte 6	1	1	0	1	3
Tüte 7	1	1	0	1	3
Tüte 8	1	1	0	1	3
Tüte 9	1	1	0	1	3

`wundertuete3.txt`

Tüten	Spiel 1	Spiel 2	Spiel 3	Spiel 4	Spiel 5	Gesamt
Tüte 1	1	1	0	0	0	2
Tüte 2	1	1	0	0	0	2
Tüte 3	0	1	1	0	0	2
Tüte 4	0	1	1	0	0	2
Tüte 5	0	1	1	0	0	2
Tüte 6	0	1	1	0	0	2
Tüte 7	0	1	1	0	0	2
Tüte 8	0	1	1	0	0	2
Tüte 9	0	1	0	1	0	2
Tüte 10	0	1	0	1	0	2
Tüte 11	0	1	0	0	1	2

`wundertuete4.txt`

Tüten	Spiel 1	Spiel 2	Spiel 3	Spiel 4	Spiel 5	Spiel 6	Gesamt
Tüte 1	2	6	3	5	31	5	52
Tüte 2	2	6	3	5	30	6	52
Tüte 3	2	6	3	5	30	6	52
Tüte 4	2	6	2	6	30	6	52
Tüte 5	1	7	2	6	30	6	52
Tüte 6	1	7	2	6	30	5	51
Tüte 7	1	7	2	6	30	5	51
Tüte 8	1	7	2	6	30	5	51
Tüte 9	1	7	2	6	30	5	51
Tüte 10	1	7	2	6	30	5	51
Tüte 11	1	7	2	6	30	5	51
Tüte 12	1	7	2	6	30	5	51
Tüte 13	1	7	2	6	30	5	51
Tüte 14	1	7	2	6	30	5	51
Tüte 15	1	7	2	6	30	5	51
Tüte 16	1	7	2	5	31	5	51
Tüte 17	1	6	3	5	31	5	51

wundertuete5.txt

Tüten	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	Gesamt
Tüte 1	1	0	2	1	2	0	2	0	1	1	1	1	0	1	2	2	0	1	1	2	0	2	1	24
Tüte 2	1	0	2	1	2	0	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	2	1	24
Tüte 3	1	0	2	1	2	0	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	2	1	24
Tüte 4	1	0	2	1	2	0	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	2	1	24
Tüte 5	1	0	2	1	2	0	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	2	1	24
Tüte 6	1	0	2	1	2	0	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	2	1	24
Tüte 7	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	2	1	24
Tüte 8	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	2	1	24
Tüte 9	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	2	1	24
Tüte 10	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	2	1	24
Tüte 11	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 12	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 13	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 14	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 15	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 16	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 17	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 18	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 19	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 20	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 21	1	0	2	1	1	1	2	0	1	1	1	1	0	1	2	1	1	1	1	2	0	1	2	24
Tüte 22	1	0	2	1	1	1	2	0	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 23	1	0	2	1	1	1	2	0	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 24	1	0	2	1	1	1	2	0	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 25	1	0	2	1	1	1	1	1	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 26	1	0	2	1	1	1	1	1	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 27	1	0	2	1	1	1	1	1	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24

Tüte 28	1	0	2	1	1	1	1	1	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 29	1	0	2	1	1	1	1	1	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 30	1	0	2	1	1	1	1	1	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 31	1	0	2	1	1	1	1	1	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 32	1	0	2	1	1	1	1	1	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 33	1	0	2	1	1	1	1	1	1	0	2	1	0	1	1	2	1	1	1	2	0	1	2	24
Tüte 34	1	0	2	1	1	1	1	1	1	0	2	1	0	0	2	2	1	1	1	2	0	1	2	24
Tüte 35	1	0	2	1	1	1	1	1	1	0	2	1	0	0	2	2	1	1	1	2	0	1	1	23
Tüte 36	1	0	2	1	1	1	1	1	1	0	2	1	0	0	2	2	1	1	1	2	0	1	1	23
Tüte 37	1	0	2	1	1	1	1	1	1	0	2	1	0	0	2	2	1	1	1	2	0	1	1	23
Tüte 38	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	2	0	1	1	23
Tüte 39	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	2	0	1	1	23
Tüte 40	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	2	0	1	1	23
Tüte 41	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	2	0	1	1	23
Tüte 42	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	2	0	1	1	23
Tüte 43	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 44	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 45	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 46	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 47	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 48	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 49	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 50	1	0	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 51	0	1	2	1	1	1	1	1	1	0	2	0	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 52	0	1	2	1	1	1	1	1	1	0	1	1	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 53	0	1	2	1	1	1	1	1	1	0	1	1	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 54	0	1	2	1	1	1	1	1	1	0	1	1	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 55	0	1	2	1	1	1	1	1	1	0	1	1	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 56	0	1	2	1	1	1	1	1	1	0	1	1	1	0	2	2	1	1	1	1	1	1	1	23

Tüte 57	0	1	2	1	1	1	1	1	1	0	1	1	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 58	0	1	2	1	1	1	1	1	1	0	1	1	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 59	0	1	2	1	1	1	1	1	1	0	1	1	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 60	0	1	2	1	1	1	1	1	1	0	1	1	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 61	0	1	1	2	1	1	1	1	1	0	1	1	1	0	2	2	1	1	1	1	1	1	1	23
Tüte 62	0	1	1	2	1	1	1	1	1	0	1	1	0	1	2	2	1	1	1	1	1	1	1	23
Tüte 63	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	1	1	1	23
Tüte 64	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	1	1	1	23
Tüte 65	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	1	1	1	23
Tüte 66	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	1	1	1	23
Tüte 67	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	1	1	1	23
Tüte 68	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	1	1	1	23
Tüte 69	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	1	1	1	23
Tüte 70	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	1	1	1	23
Tüte 71	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	1	1	1	23
Tüte 72	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	0	2	1	23
Tüte 73	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	0	2	1	23
Tüte 74	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	0	2	1	23
Tüte 75	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	0	2	1	23
Tüte 76	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	0	2	1	23
Tüte 77	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	1	1	1	1	0	2	1	23
Tüte 78	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	0	2	1	1	0	2	1	23
Tüte 79	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	0	2	1	1	0	2	1	23
Tüte 80	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	0	2	1	1	0	2	1	23
Tüte 81	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	0	2	1	1	0	2	1	23
Tüte 82	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	0	2	0	2	0	2	1	23
Tüte 83	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	0	2	0	2	0	2	1	23
Tüte 84	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	0	2	0	2	0	2	1	23
Tüte 85	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	0	2	0	2	0	2	1	23

Tüte 86	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	0	2	0	2	0	2	1	23
Tüte 87	0	1	1	2	1	1	1	1	0	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23
Tüte 88	0	1	1	1	2	1	1	1	0	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23
Tüte 89	0	1	1	1	2	1	1	1	0	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23
Tüte 90	0	1	1	1	2	0	2	0	1	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23
Tüte 91	0	1	1	1	2	0	2	0	1	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23
Tüte 92	0	0	2	1	2	0	2	0	1	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23
Tüte 93	0	0	2	1	2	0	2	0	1	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23
Tüte 94	0	0	2	1	2	0	2	0	1	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23
Tüte 95	0	0	2	1	2	0	2	0	1	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23
Tüte 96	0	0	2	1	2	0	2	0	1	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23
Tüte 97	0	0	2	1	2	0	2	0	1	1	1	1	0	1	2	2	0	1	1	2	0	2	1	23

J1.4 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Modellierung ungeeignet**
  Die Tüten mit den darin enthalten Spielen sollten korrekt eingelesen und in einer dafür sinnvollen Datenstruktur, wie beispielsweise einem Array gespeichert werden.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Das verwendete Lösungsverfahren muss alle Spiele auf alle Tüten aufteilen. Dabei darf der Unterschied in der Gesamtanzahl der Spiele in den Tüten, sowie der Unterschied in der Anzahl der Spiele einer Spielsorte in den Tüten nicht mehr als 1 betragen.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Es sollte ersichtlich werden, wie viele Spiele sich von jeder Art in jeder Tüte befinden. Dazu sollte für jede Tüte sowohl die Gesamtanzahl als auch die Anzahl pro Sorte der darin enthaltenen Spiele angegeben werden. Eine Auflistung der Spiele in einer Tüte reicht nicht, da man so die Unterschiede nur sehr mühsam erkennen kann.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 4 der vorgegebenen Beispiele `wundertuete0.txt` bis `wundertuete5.txt` enthalten. Es ist in Ordnung, wenn insbesondere bei `wundertuete5.txt` die Ausgabe nur auszugsweise in der Dokumentation enthalten ist.

Junioraufgabe 2: St. Egano

J2.1 Lösungsidee

In dieser Aufgabe wurden geheime Botschaften in Fotos versteckt – bzw., aus technischer Sicht, in digitalen Bilddateien. Eine Bilddatei besteht aus mehreren Pixeln, die in Spalten und Zeilen organisiert sind. Die Position eines Pixels in der Datei wird deshalb mit den Koordinaten x (Nummer der Spalte) und y (Nummer der Zeile) angegeben. Jedes Pixel hat eine Farbe, die durch ihre Rot-, Grün- und Blauwerte $[r, g, b]$ festgelegt ist. Jeder dieser drei Farbwerte eines Pixels wird durch eine Zahl von 0 bis 255 beschrieben; je höher die Zahl, desto stärker der Beitrag der jeweiligen Grundfarbe zur Farbe des Pixels. So hat z.B. Weiß die Farbwerte $[255, 255, 255]$, Schwarz hat $[0, 0, 0]$ und Rot hat $[255, 0, 0]$. Wie in der Aufgabenstellung beschrieben wurden einige Pixel nun abgeändert, um jeweils einen Teil der Nachricht darin zu verstecken.:

- Der r-Wert ist ein kodiertes Zeichen der Nachricht.
- Der g-Wert beschreibt, wie viele Zeichen weiter rechts der nächste Nachrichtenteil zu finden ist.
- Der b-Wert beschreibt, wie viele Zeichen weiter unten der nächste Nachrichtenteil zu finden ist.

Als erstes müssen wir uns überlegen, wie wir den r-Wert in ein Zeichen umwandeln können. Laut Aufgabenstellung gibt r die ASCII-Kodierung eines einzelnen Zeichens der Nachricht an. ASCII ist eine Kodierung, die schon seit Jahrzehnten von Computern verwendet wird, um Zeichen als Zahlenwerte zu speichern. Dabei steht jeder Wert für ein bestimmtes Zeichen. Eine vollständige Tabelle des ASCII-Codes mit der Zuordnung von Zahlenwerten zu Zeichen kann hier `https://de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange` gefunden werden. Allerdings ordnet der ASCII-Code nur den Zahlen von 0 bis 127 Zeichen zu. In dieser Aufgabe sind die Zeichen also streng genommen nach Unicode[1] kodiert, da die Zahlenwerte über 127 hinaus gehen. In einigen Beispielen gab es r-Werte im Bereich von 128 bis 255, denen u. a. Umlaute und andere besondere Zeichen zugeordnet sind. Die meisten Programmiersprachen haben eine Funktion, die die Umwandlung von Zahlen in Zeichen automatisch vornimmt und sich dabei nach dem Unicode-Standard richtet - wie z. B. `chr` in Python.

Weiterhin müssen wir uns überlegen, wie wir wieder am oberen bzw. linken Bildrand starten können, wenn wir bei einer Bewegung über den unteren bzw. rechten Bildrand kommen. Nehmen wir an, dass x und y die Position des aktuell betrachteten Pixels im Bild beschreiben. Wenn $x >$ Bildbreite oder $y >$ Bildhöhe ist, dann können wir jeweils so lange die Bildbreite bzw. -höhe abziehen, bis die Werte wieder kleiner als Breite oder Höhe sind. Alternativ kann dazu der Modulo-Operator $\mod$ (Erklärung Modulo Operator Siehe Kapitel J1.2) verwendet werden, der denselben Effekt erzielt. Der Modulo-Operator gibt uns den Rest einer Ganzzahldivision zurück.

Mit diesen Informationen können wir nun die geheime Botschaft aus den Fotos auslesen.

1. Beginne am Pixel oben links, also mit Position $x = 0, y = 0$ und den Farbwerten r, g, b.
2. Füge das nach Unicode (oder ASCII-Code) dem Wert r zugeordnete Zeichen an die Nachricht an.

[1] `https://de.wikipedia.org/wiki/Unicode`

Algorithmus 1 Programm zum Auslesen der geheimen Botschaft

```
1: procedure NACHRICHT LESEN(bild)
2:     x ← 0
3:     y ← 0
4:     aktuellesPixel ← bild_{x,y}
5:     nachricht ← nachricht + dekodiere(aktuellesPixel_r)
6:     while aktuellesPixel_g und aktuellesPixel_b nicht 0 do
7:         x ← (x + aktuellesPixel_g) mod bild_breite
8:         y ← (y + aktuellesPixel_b) mod bild_hoehe
9:         aktuellesPixel ← bild_{x,y}
10:        nachricht ← nachricht + dekodiere(aktuellesPixel_r)
11:    end while
12: end procedure
```

3. g und b beschreiben, um wie viele Pixel sich im Bild nach rechts bzw. unten bewegt werden muss.
4. Wird damit über den Bildrand hinaus gegangen, setze links bzw. oben wieder an und zähle von dort weiter.
5. Sind die Werte g- und b jeweils 0, so wurde die Nachricht vollständig ausgelesen; ansonsten gehe mit den neuen Pixel-Koordinaten zurück zu Schritt 2.

J2.2 Umsetzung

Zunächst muss das jeweilige Bild des entsprechenden Beispiels eingelesen werden. Dazu können die zur Verfügung gestellten Dateien auf der BWINF-Seite genutzt werden.

Nun muss der oben informell beschrieben Algorithmus für die Programmierung konkretisiert werden. Mithilfe zweier Variablen können wir die Positionswerte x und y des zu betrachtenden Pixels speichern. Für Schritt 1 werden diese jeweils mit 0 initialisiert. Nun überprüfen wir die Werte r, g, b des Pixels an der durch x und y gegebenen Position. Wir dekodieren den r-Anteil und fügen das Ergebnis ans Ende der Nachricht an. Die g- und b-Anteile addieren wir jeweils auf die aktuellen x- und y-Werte. Die vorigen Schritte werden wiederholt, bis $g = b = 0$ gilt. Insgesamt beschreibt Algorithmus 1 die Umsetzung der Lösungsidee.

J2.3 Laufzeit

Wir nehmen an, dass die Nachricht eine begrenzte Länge hat und ordentlich kodiert ist; das heißt, es wird irgendwann ein Pixel erreicht, dessen g- und b-Werte gleich 0 sind. Dann betrachtet der Algorithmus jedes Pixel des Bildes höchstens einmal. Die Laufzeit dieses Algorithmus' kann deshalb durch $\mathcal{O}(n)$ beschrieben werden, wobei n die Gesamtanzahl der Pixel im Bild ist.

J2.4 Beispiele

Bild 01

```
Hallo Welt
```

Bild 02

```
Hallo Gloria

Wie treffen uns am Freitag um 15:00 Uhr vor der Eisdiele am
Markplatz.

Alle Liebe,
Juliane
```

Bild 03

```
Hallo Juliane,

Super, ich werde da sein! Ich freue mich schon auf den riesen
Eisbecher mit Erdbeeren.

Bis bald,
Gloria
```

Bild 04

```
Der Jugendwettbewerb Informatik ist ein Programmierwettbewerb
für alle, die erste Programmiererfahrungen sammeln und vertiefen
möchten. Programmiert wird mit Blockly, einer Bausteinorientierten
Programmiersprache. Vorkenntnisse sind nicht nötig. Um sich mit
den Aufgaben des Wettbewerbs vertraut zu machen, empfehlen wir
unsere Trainingsseite . Er richtet sich an Schülerinnen und
Schüler der Jahrgangsstufen 5 - 13, prinzipiell ist aber eine
Teilnahme ab Jahrgangsstufe 3 möglich. Der Wettbewerb besteht aus
drei Runden. Die ersten beiden Runden erfolgen online. In der 3.
Runde werden zwei Aufgaben gestellt, diese gilt es mit eigenen
Programmierwerkzeugen zuhause zu bearbeiten.
```

Bild 05

Der Bundeswettbewerb Informatik richtet sich an Jugendliche bis 21 Jahre, vor dem Studium oder einer Berufstätigkeit. Der Wettbewerb beginnt am 1. September, dauert etwa ein Jahr und besteht aus drei Runden. Dabei können die Aufgaben der 1. Runde ohne größere Informatikkenntnisse gelöst werden; die Aufgaben der 2. Runde sind deutlich schwieriger.

Der Bundeswettbewerb ist fachlich so anspruchsvoll, dass die Gewinner i.d.R. in die Studienstiftung des deutschen Volkes aufgenommen werden. Aus den Besten werden die TeilnehmerInnen für die Internationale Informatik-Olympiade ermittelt. Der Bundeswettbewerb ermöglicht den Teilnehmenden, ihr Wissen zu vertiefen und ihre Begabung weiterzuentwickeln. So trägt der Wettbewerb dazu bei, Jugendliche mit besonderem fachlichen Potenzial zu erkennen.

Bild 06

Bonn

Die Bundesstadt Bonn (im Latein der Humanisten Bonna) ist eine kreisfreie Großstadt im Regierungsbezirk Köln im Süden des Landes Nordrhein-Westfalen und Zweitregierungssitz der Bundesrepublik Deutschland. Mit 336.465 Einwohnern (31. Dezember 2022) zählt Bonn zu den zwanzig größten Städten Deutschlands. Bonn gehört zu den Metropolregionen Rheinland und Rhein-Ruhr sowie zur Region Köln/Bonn. Die Stadt an beiden Ufern des Rheins war von 1949 bis 1973 provisorischer Regierungssitz und von 1973 bis 1990 Bundeshauptstadt und bis 1999 Regierungssitz Deutschlands, danach wurde sie zweiter Regierungssitz. Die Vereinten Nationen unterhalten seit 1951 hier einen Sitz.

[...]

Arbeitsmarktbehörden
Bonn ist außerdem Standort der Zentralen Auslands- und Fachvermittlung (ZAV) der Bundesagentur für Arbeit (BA). Im Stadtteil Duisdorf befindet sich der Hauptsitz der ZAV mit ihren bundesweit 18 Standorten.

Quelle: https://de.wikipedia.org/wiki/Bonn

Bild 07

Es hatte ein Mann einen Esel, der schon lange Jahre die Säcke unverdrossen zur Mühle getragen hatte, dessen Kräfte aber nun zu Ende giengen, so daß er zur Arbeit immer untauglicher ward. Da dachte der Herr daran, ihn aus dem Futter zu schaffen, aber der Esel merkte daß kein guter Wind wehte, lief fort und machte sich auf den Weg nach Bremen: dort, meinte er, könnte er ja Stadtmusikant werden. Als er ein Weilchen fortgegangen war, fand er einen Jagdhund auf dem Wege liegen, der jappte wie einer, der sich müde gelaufen hat. "Nun, was jappst du so, Packan?" fragte der Esel. "Ach," sagte der Hund, "weil ich alt bin und jeden Tag schwächer werde, auch auf der Jagd nicht mehr fort kann, hat mich mein Herr wollen todt schlagen, da hab ich Reißaus genommen; aber womit soll ich nun mein Brot verdienen?" "Weißt du was," sprach der Esel, "ich gehe nach Bremen und werde dort Stadtmusikant, geh mit und laß dich auch bei der Musik annehmen. Ich spiele die Laute, und du schlägst die Pauken." Der Hund wars zufrieden, und sie giengen weiter. Es dauerte nicht lange, so saß da eine Katze an dem Weg und machte ein Gesicht wie drei Tage Regenwetter.

[...]

Da erschrack er gewaltig, lief und wollte zur Hinterthüre hinaus, aber der Hund, der da lag, sprang auf und biß ihn ins Bein: und als er über den Hof an dem Miste vorbei rannte, gab ihm der Esel noch einen tüchtigen Schlag mit dem Hinterfuß; der Hahn aber, der vom Lärmen aus dem Schlaf geweckt und munter geworden war, rief vom Balken herab "kikeriki!" Da lief der Räuber, was er konnte, zu seinem Hauptmann zurück und sprach "ach, in dem Haus sitzt eine gräuliche Hexe, die hat mich angehaucht und mit ihren langen Fingern mir das Gesicht zerkratzt: und vor der Thüre steht ein Mann mit einem Messer, der hat mich ins Bein gestochen: und auf dem Hof liegt ein schwarzes Ungethüm, das hat mit einer Holzkeule auf mich losgeschlagen: und oben auf dem Dache, da sitzt der Richter, der rief bringt mir den Schelm her. Da machte ich daß ich fortkam." Von nun an getrauten sich die Räuber nicht weiter in das Haus, den vier Bremer Musikanten gefiels aber so wohl darin, daß sie nicht wieder heraus wollten. Und der das zuletzt erzählt hat, dem ist der Mund noch warm.

J2.5 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Modellierung ungeeignet**
  Das Bild sollte in einer sinnvollen Datenstruktur eingelesen werden, die es nicht unnötig kompliziert macht, *rgb*-Werte auszulesen.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Die Implementierung sollte das in der Aufgabenstellung beschriebene Verfahren korrekt umsetzen. Insbesondere sollte darauf geachtet werden, dass die Berechnung der Position des folgenden Pixels (idealerweise Modulo-Addition der *g*- und *b*-Werte auf die Koordinaten, aber jede korrekte Berechnung ist akzeptabel) richtig umgesetzt ist. Wenn die Dekodierung sich auf den ASCII-Bereich von 0 – 127 beschränkt,s gibt es keinen Abzug.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Die vollständige Nachricht, umgewandelt in ASCII-Zeichen, sollte am Ende ausgegeben werden. Es ist in Ordnung, wenn insbesondere bei den Beispielen 6 und 7 die Ausgabe nur auszugsweise in der Dokumentation enthalten ist oder auf eine externe Datei verwiesen wird.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 4 der vorgegebenen Beispiele `bild01.zip` bis `bild07.zip` enthalten. Wenn die Aufgabe mit einer graphischen Programmiersprache gelöst wurde, so ist es ausreichend die Ergebnisse bis einschließlich `bild03.zip` anzugeben.

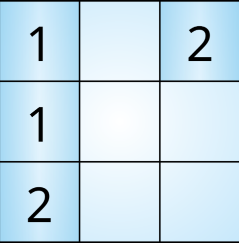
Aufgabe 1: Arukone

In dieser Aufgabe soll ein Algorithmus entwickelt werden, mit dem Arukone-Rätsel generiert werden können.

1.1 Arukone-Rätsel

In einem Arukone-Rätsel sind in einem quadratischen Gitter der Größe $n \times n$ die Zahlen von 1 bis m verteilt. Jede Zahl kommt exakt zweimal in dem Gitter vor.

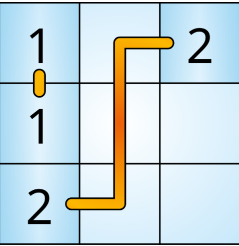
Hier ein Rätsel der Größe 3×3 mit den Zahlen 1 und 2:



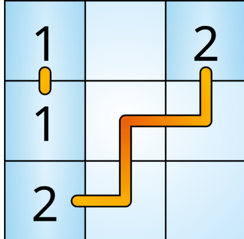
Eine Lösung für ein Arukone-Rätsel verbindet jedes Paar von Zahlen mit einer durchgehenden Verbindung im Gitter von Gitterpunkt zu Gitterpunkt, so dass

- die einzelnen Verbindungssegmente ausschließlich vertikal oder horizontal verlaufen (nicht diagonal) und
- sich keine Verbindungen überkreuzen.

Hier eine Lösung für das obige Rätsel:

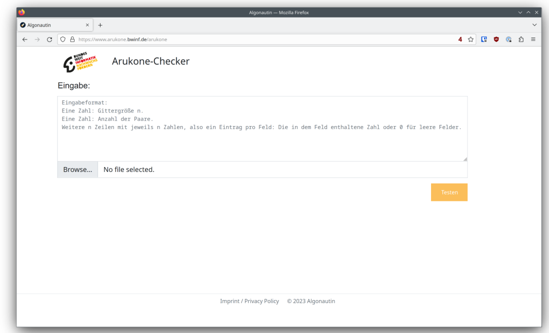


Ein Arukone-Rätsel kann mehrere Lösungen enthalten. Hier eine weitere Lösung für das obige Arukone-Rätsel:

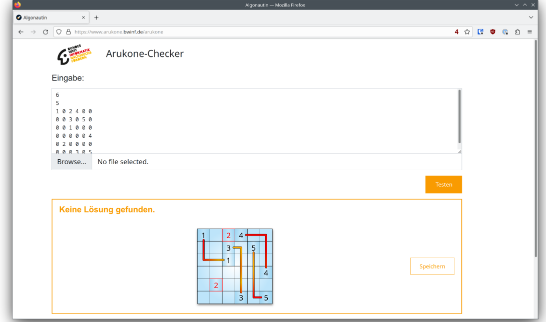


Ein Arukone-Rätsel kann auch unlösbar sein. Das heißt, dass sich keine Verbindungen zwischen allen Zahlenpaaren einzeichnen lassen, die sich nicht überkreuzen.

Hier ein unlösbares Rätsel der Größe 3×3 mit den Zahlen 1 und 2:



(a) Webseite ohne Eingaben und ...

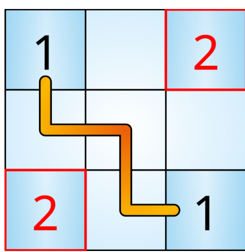


(b) ... mit dem Beispiel-Rätsel

Abbildung 1.1: **Abbildung 1**: Die Webseite „Arukone-Checker“ unter `https://www.arukone.bwinf.de/arukone`

1		2
2		1

Hier lässt sich zwar eine Verbindung zwischen den 1en oder zwischen den 2en einzeichnen, aber nicht beide gleichzeitig:



Zusätzliche Anforderung

In der Aufgabe wird auf den von BWINF bereitgestellten Arukone-Checker[2] (in der Folge *BWINF-Checker* genannt) verwiesen. Dies ist eine Webseite, in der ein Arukone-Rätsel eingegeben werden kann, dass dann von einem Arukone-Löser bearbeitet wird. Dieses Programm verwendet einen recht einfachen Algorithmus und findet nicht für jedes lösbare Rätsel eine Lösung. Zum Beispiel lässt sich das Arukone-Rätsel, das in der Aufgabenstellung vorgestellt wird, nicht von dem BWINF-Checker lösen (vgl. Abb. 1.1).

In der Aufgabe soll ein Algorithmus entwickelt werden, mit dem Arukone-Rätsel generiert werden können, die zwar eine Lösung haben, aber nicht vom BWINF-Checker gelöst werden können. Konkret wird in der Aufgabenstellung gefordert, dass ein Programm geschrieben wird, das Arukone-Rätsel generiert, unter denen „auch welche“ sein sollen, die vom BWINF-Checker nicht gelöst werden können.

[2] `https://www.arukone.bwinf.de/arukone`

1.2 Lösungsstrategien

Im Folgenden besprechen wir drei verschiedene Lösungsstrategien für diese Aufgabe:

- Generieren von zufälligen Arukone-Rätsel-Lösungen und Erzeugen des Rätsels aus der Lösung
- Generieren von zufälligen Arukone-Rätseln ohne die Lösung direkt zu kennen
- Zusammensetzen von Aurukone-Rätseln aus fertigen Mustern / Vorlagen

Alle diese Strategien haben verschiedene Vor- und Nachteile im Bezug auf die Aufgabenstellung, sie lassen sich aber kombinieren, um Nachteile auszugleichen.

Erzeugen von Arukone-Rätseln aus Lösungen

Diese Strategie beinhaltet das Erzeugen eines gelösten Arukone-Gitters und Ausgeben des Gitters ohne die Verbindungen als Arukone-Rätsel. Der Vorteil einer solchen Strategie ist, dass inhärent sicher gestellt ist, dass für das Arukone-Rätsel eine Lösung existiert.

Die einfachste Vorgehensweise dafür ist, zufällige Startpunkte für alle Zahlen von 1 bis m auf dem Gitter zu verteilen. Danach lassen sich die Verbindungen von den Startpunkten zufällig generieren; dabei muss man darauf achten, dass die Verbindungen sich nicht kreuzen. Schließlich setzt man die Zahlen von 1 bis m entsprechend auf die Endpunkte der Verbindungen und erhält dadurch ein gelöstes Arukone-Rätsel.

Beim Generieren der Verbindungen gibt es verschiedene Herangehensweisen, die verschiedene Arten von Rätseln erzeugen und unterschiedliche Nachteile haben:

- Mit dem Erzeugen von kurzen Schritten (etwa jeweils ein Gitterpunkt weiter) für die Verbindungen werden Verbindungen erzeugt, die eher einem Random-Walk[3] entsprechen und sich wenig verknoten.
- Mit dem Erzeugen von sehr langen Schritten (so viele Gitterpunkte weiter wie möglich) werden oft die Wege für andere Verbindungen abgeschnitten.

Die Nachteile dieser Herangehensweisen lassen sich vermeiden, indem zufällige Schrittlängen für die Generierung der Verbindungen verwendet werden. Statt nur ein Ende der Verbindungen zu verlängern, können auch an beiden Enden Segmente hinzugefügt werden.

Experimentieren mit dieser Strategie zeigt aber, dass selbst mit zufälligen Schrittlängen häufig nur Arukone-Rätsel erzeugt werden, die sehr einfach zu lösen sind.

Erzeugen von Arukone-Rätseln mit zufälligen Verteilungen der Zahlen

Diese Strategie ist konzeptionell die einfachste: Es werden einfach auf dem Gitter die Zahlen von 1 bis m je zwei mal zufällig platziert (natürlich so, dass keine zwei Zahlen den gleichen Gitterpunkt einnehmen).

Aber auch der Nachteil dieser Strategie ist offensichtlich: Es ist nicht ohne weiteres garantiert, dass ein so erzeugtes Rätsel auch tatsächlich eine Lösung besitzt.

[3] `https://de.wikipedia.org/wiki/Random_Walk`

Daher lässt sich dieses Verfahren nur verwenden, wenn man anschließend auch überprüft, ob ein gegebenes Arukone-Rätsel lösbar ist. Es muss hierfür also auch ein Arukone-Löser geschrieben werden. Dies ist bei der Strategie, bei der die Rätsel aus Lösungen erzeugt werden, nicht nötig.

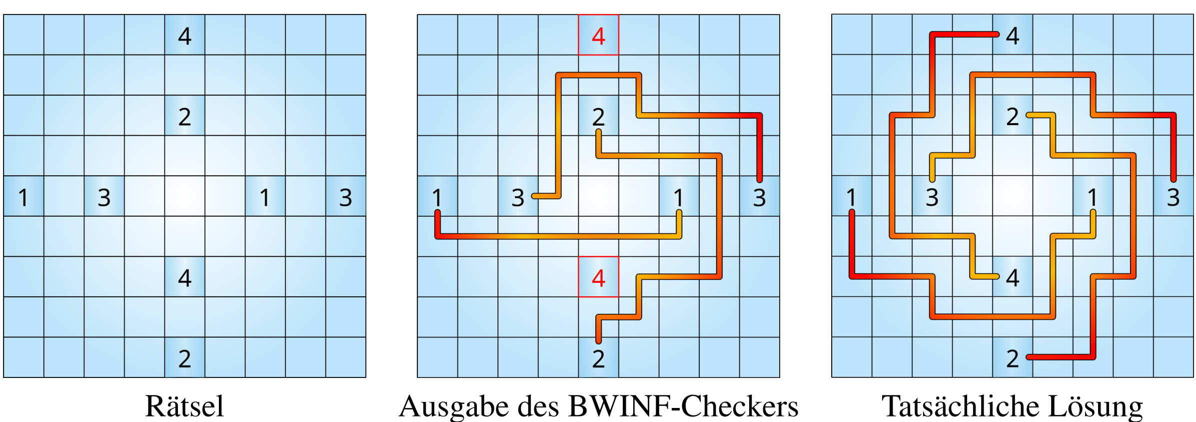
Arukone-Lösungen finden

Es ist nicht so einfach, für beliebige lösbare Arukone-Rätsel eine Lösung zu finden. Eine naive Herangehensweise könnte sein, für alle Zahlenpaare die kürzeste / einfachste Verbindung unter Berücksichtigung aller schon vorher festgelegten Verbindungen zu finden.

Algorithmus 2 Löse Rätsel (greedy)

```
 1: procedure LÖSERÄTSELGREEDY(Gitter g)
 2:     for i ← 1...m do
 3:         v ← suche die kürzeste Verbindung zwischen den beiden Zahlen i im Gitter g
 4:         if es wurde keine Verbindung gefunden then
 5:             return „keine Lösung gefunden“
 6:         else
 7:             g ← trage die Verbindung v im Gitter g ein
 8:         end if
 9:     end for
10:     return das vollständige Gitter g
11: end procedure
```

Aber selbst wenn dies für alle Permutationen in der Reihenfolge der Verbindungen probiert wird, garantiert das nicht, dass eine Lösung gefunden wird. Das folgende Beispiel illustriert dies:



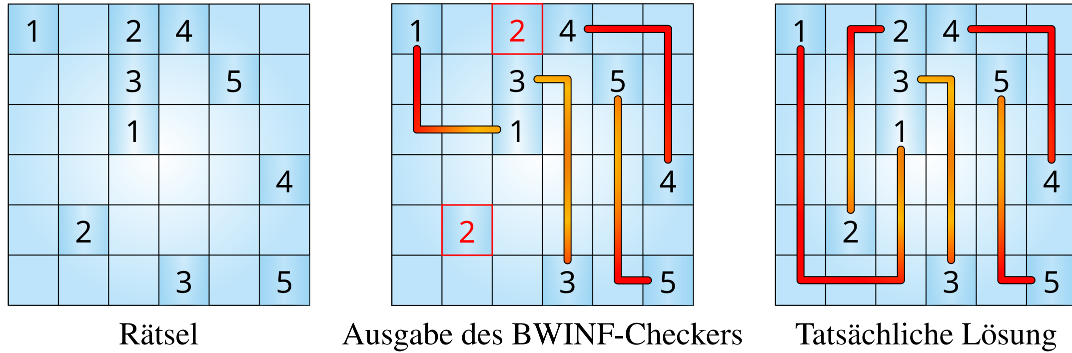
Rätsel — Ausgabe des BWINF-Checkers — Tatsächliche Lösung

Man sieht: Sobald die beiden 1en auf dem kürzesten Weg miteinander verbunden werden, können die 2en und 4en nicht mehr gleichzeitig verbunden werden. Aufgrund der Symmetrie des Rätsels entsteht dieses Problem auch für alle anderen Zahlen.

Es ist allerdings gar nicht nötig, beliebige Arukone-Rätsel lösen zu können. Nach der Aufgabenstellung genügt es, lediglich Arukone-Rätsel zu finden, die vom BWINF-Checker nicht gelöst werden können. Daher genügt es bereits, einen Löser zu haben, der *besser* ist, als der Löser des BWINF-Checkers.

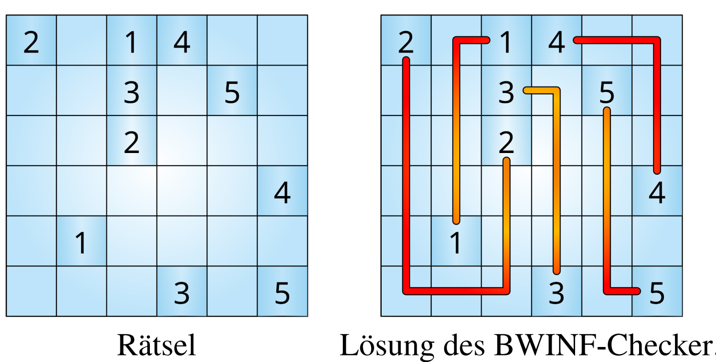
Den BWINF-Checker reverse-engineeren

Um gezielt Rätsel erzeugen zu können, die vom BWINF-Checker nicht gelöst werden können, kann es auch helfen, das Verhalten des Checkers zu untersuchen. Wenn man weiß, wie der Checker vorgeht, oder zumindest eine Regelmäßigkeit in den Rätseln findet, die von ihm nicht gelöst werden können, kann man gezielt Rätsel erzeugen, die von ihm nicht gelöst werden können – oder weniger gezielt solange zufällige Rätsel erzeugen, bis man ein lösbares findet, dass nicht von ihm gelöst werden kann. Gibt man zum Beispiel das Rätsel, das in der Aufgabenstellung vorgestellt wird, in den BWINF-Checker ein, wird keine Lösung gefunden



Rätsel Ausgabe des BWINF-Checkers Tatsächliche Lösung

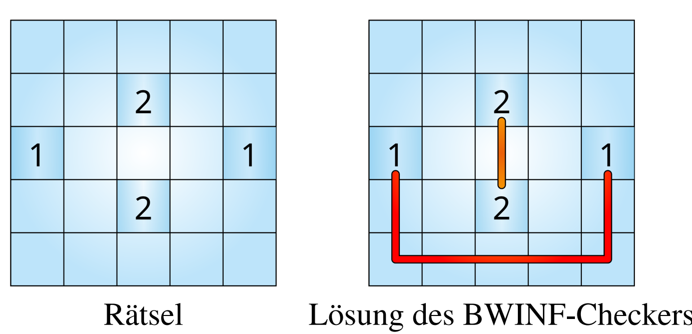
Dies deutet darauf hin, dass der Checker greedy vorgeht und die Verbindung für die 1en zuerst sucht, und anschließend nicht mehr anpasst, um die 2en verbinden zu können. Das lässt sich einfach überprüfen, indem man die Positionen der 1en und der 2en vertauscht. Hier findet der BWINF-Checker eine Lösung.



Rätsel Lösung des BWINF-Checker,

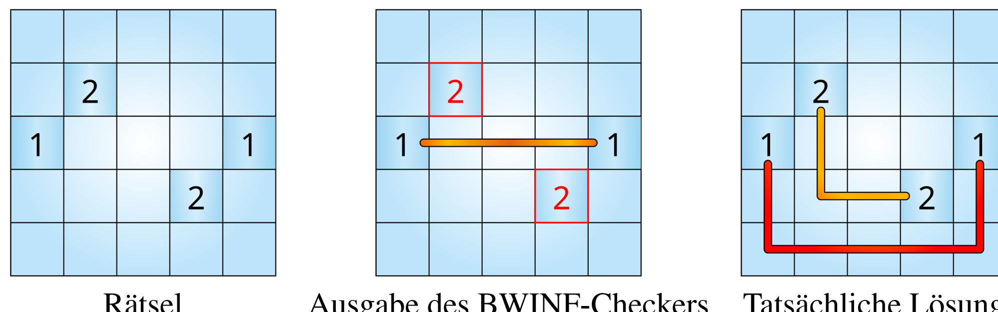
Es liegt also nahe, dass der BWINF-Checker einen Ansatz verwendet, der nacheinander versucht, die einzelnen Zahlenpaare zu verbinden.

Das scheint allerdings nicht alles zu sein. So findet er zum Beispiel für dieses Rätsel eine Lösung:



Rätsel Lösung des BWINF-Checkers

Aber für dieses Beispiel nicht:



Rätsel Ausgabe des BWINF-Checkers Tatsächliche Lösung

Es scheinen also noch weitere Heuristiken für die Probierreihenfolge verwendet zu werden, etwa die Entfernungen der Zahlen eines Zahlenpaares.

Allgemein kann dieses Wissen über die Reihenfolgeabhängigkeit genutzt werden, um zuverlässig Arukone-Rätsel zu erzeugen, die vom BWINF-Checker nicht gelöst werden können: Man erzeugt Rätsel, die in in einer Permutation der Zahlen von dem einfachen Algorithmus 3 gelöst werden können, in einer anderen Permutation jedoch nicht.

Algorithmus 3 Finde lösbares Rätsel das vom BWINF-Checker nicht gelöst werden kann

1: **procedure** ERZEUGERÄTSEL(Gitter g)
2:     **loop**
3:         $g \leftarrow$ erzeuge leeres Gitter der Größe $n \times n$
4:         $g \leftarrow$ trage die Zahlen von 1 bis m je zweimal zufällig im Gitter g ein
5:         teste das Gitter g mit **Algorithmus 2** auf eine Lösung
6:         **if** es wurde eine Lösung gefunden **then**
7:             $g' \leftarrow$ Ersetze im Gitter g jede Zahl i durch $m - i + 1$
8:             teste das Gitter g' mit **Algorithmus 2** auf eine Lösung
9:             **if** es wurde *keine* Lösung gefunden **then**
10:                 **return** das Gitter g'
11:             **end if**
12:         **end if**
13:     **end loop**
14: **end procedure**

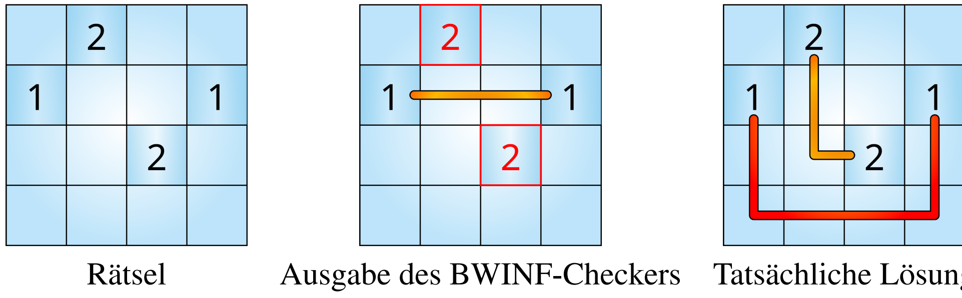
Verwenden von Mustern

Die letzte hier vorgestellte Strategie ist möglicherweise auf den ersten Blick etwas überraschend: Man findet ein kleines oder gar minimales (mit nur zwei Zahlenpaaren) Arukone-Rätsel, das nicht vom BWINF-Checker gelöst werden kann, und verwendet dieses immer wieder.

Dabei ist es wichtig, weiterhin sicherzustellen, dass, wie in der Aufgabenstellung gefordert, verschiedene Rätsel erstellt werden. Dies können wir erreichen, indem wir zum Beispiel mehrere kleine Arukone-Rätsel, die vom BWINF-Checker nicht gelöst werden können, zu einem großen Rätsel zusammensetzen.

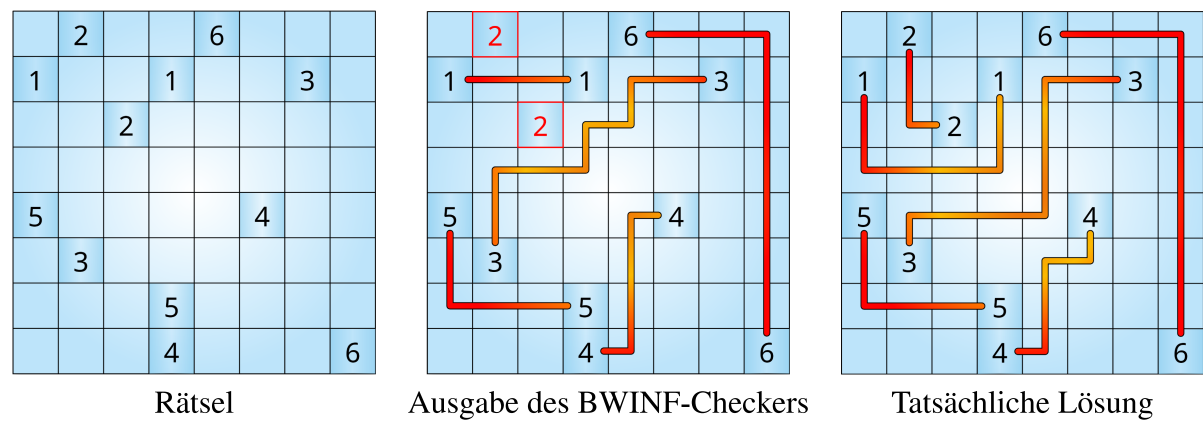
Ebenfalls lässt sich mit einem kleinen Rätsel auch ein größeres erzeugen, indem man den Rest des Gitters nach einer beliebigen der oben beschriebenen Verfahren zum Finden von Arukone-Rätseln kombiniert. Dies hat den Vorteil, dass garantiert verschiedene Rätsel generiert werden.

Ein solches minimales Rätsel könnte das folgende sein:



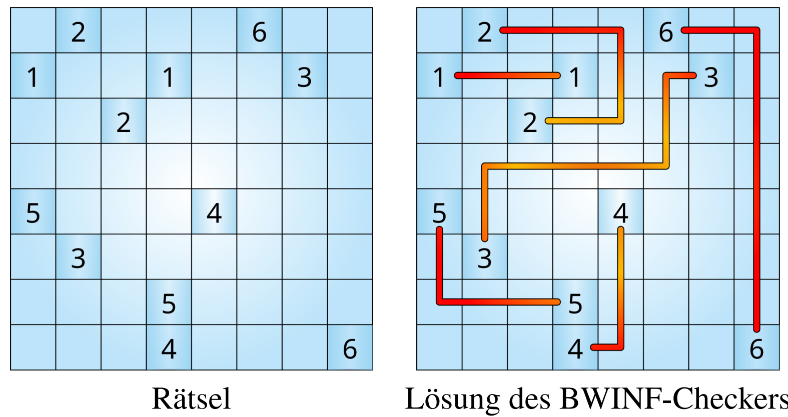
Rätsel — Ausgabe des BWINF-Checkers — Tatsächliche Lösung

Dies könnte zum Beispiel auf folgende Weise zu einem 8×8-Rätsel erweitert werden:



Rätsel — Ausgabe des BWINF-Checkers — Tatsächliche Lösung

Dabei ist für dieses Beispiel eines Muster-Rätsels wichtig, dass der Gitterpunkt an der fünften Stelle in der ersten Zeile immer von einer Zahl besetzt wird. Ansonsten kann der BWINF-Checker andere Verbindungen nutzen und eine Lösung finden:



Rätsel — Lösung des BWINF-Checkers

1.3 Abschließende Bemerkungen

Es sind noch weitere Ansätze denkbar,

- Arukone-Rätsel zu erstellen,
- Arukone-Rätsel zu lösen,
- die Wahrscheinlichkeit zu senken, dass ein Arukone-Rätsel vom BWINF-Checker gelöst werden kann, oder
- ganz zu verhindern, dass ein Arukone-Rätsel vom BWINF-Checker gelöst werden kann.

Diese lassen sich, wie die schon beschriebenen Ansätze, miteinander kombinieren oder in Kombination mit den schon beschriebenen Ansätzen verwenden.

Am Ende sollen aber genügend Arukone-Rätsel erzeugt werden können, die vom BWINF-Checker nicht gelöst werden können. Damit man nicht all zu lange auf so ein Rätsel warten muss, muss auch die Wahrscheinlichkeit, dass ein solches Rätsel erzeugt wird, hoch genug sein. Es ist aber ausreichend, wenn von vier erzeugten Rätseln im Schnitt eines dieses Kriterium erfüllt, die Wahrscheinlichkeit dafür also 25% oder höher ist.

1.4 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
  Damit Rätsel erzeugt werden, die nicht vom BWINF-Checker gelöst werden, muss entweder
  1. erklärt werden, wie der BWINF-Checker in Grundzügen funktioniert, oder
  2. ein Muster für Arukone-Rätsel angegeben werden, dass der BWINF-Checker zuverlässig nicht lösen kann, oder
  3. begründet werden, weshalb manche generierte Rätsel eine gewisse Mindest-Schwierigkeit haben.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Die Rätsel müssen mindestens $n/2$ verschiedene Zahlen enthalten für $n \geq 4$. Die Rätsel müssen lösbar sein. Unter den generierten Rätseln muss ein nicht zu vernachlässigender Anteil der erzeugten Rätsel vom BWINF-Checker nicht gelöst werden können.
- $[-1]$ **Rätsel nicht verschieden genug**
  Wenn der Ansatz darauf basiert, dass bekannte nicht lösbare Strukturen kombiniert werden, so muss gewährleistet sein, dass trotzdem verschiedene Rätsel generiert werden.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Da bei dieser Aufgabe nicht erwartet wird, besonders große Arukone-Rätsel zu erstellen, spielt die Laufzeit des Verfahrens an dieser Stelle nur eine untergeordnete Rolle. Abzüge gibt es nur, wenn selbst das Generieren von kleinen Arukone-Rätsel lange Zeit in Anspruch nimmt.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Zu mindestens 3 Rätseln muss die Ausgabe des BWINF-Checkers und ggf. die Lösung des Rätsels angegeben werden.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Es müssen mindestens 3 Beispiele für vom Programm erzeugte Rätsel unterschiedlicher Größe angegeben werden. Darunter müssen mindestens zwei sein, die vom BWINF-Checker nicht gelöst werden.

Aufgabe 2: Die goldene Mitte

2.1 Lösungsidee

Um Dirk den Triumph über das Rätsel zu ermöglichen, nutzen wir zunächst die Information, dass die Bausteine in Form von Würfeln und Quadern die Kiste komplett füllen sollen. Hat man also eine Menge von Bausteinen gegeben, deren Volumen kleiner oder größer als der Platz in der Kiste ist, kann eine Lösbarkeit des Rätsels direkt ausgeschlossen werden. Wenn das Volumen der Bausteine mit dem der Kiste übereinstimmt, muss überprüft werden, ob die Bausteine so platziert werden können, dass sie die Kiste füllen.

Der einfachste Ansatz hierfür wäre, per brute force alle möglichen Kombinationen an Positionen für die zur Verfügung stehenden Bausteine auszuprobieren. Dafür könnte man jedem der n Bausteine eine der möglichen Positionen zuordnen, dies überprüfen und solange fortfahren, bis eine Lösung gefunden oder keine Möglichkeiten mehr übrig sind. Teil der Überprüfung ist hier, ob sich die platzierten Blöcke gegenseitig überlappen oder aus der Kiste herausragen. Für jede Position gibt es 6 verschiedene Orientierungen, in denen der Baustein platziert werden kann. Bei einer Kantenlänge der Kiste von x gibt es x^3 viele Positionen (Kiste und Bausteine können in Würfel der Kantenlänge 1 aufgeteilt werden; im Fall der Kiste nennen wir jeden dieser Teilwürfel eine *Position*). Die Anzahl der möglichen Platzierungen aller Blöcke lässt sich mit dem Binomialkoeffizienten berechnen: Es gibt also insgesamt $\binom{x^3}{n}$ mögliche Kombinationen an Positionierungen der Blöcke. Zusätzlich gibt es für jede mögliche Kombination 6^n viele Kombinationen an Rotationen. Das liegt daran, dass für eine mögliche Positionierung der Blöcke jeder Block sechs mögliche Orientierungen hat. Für alle sechs Orientierungen des ersten Blockes müssen alle möglichen Orientierungen des 2. Blockes beachtet werden, für diese $6 \cdot 6$ Orientierungs-Kombinationen müssen alle sechs Orientierungen des 3. Blockes beachtet werden usw. Alleine durch die resultierenden $\binom{x^3}{n} \cdot 6^n$ möglichen Kombinationen wird dieser Ansatz schnell sehr rechenintensiv und ist für größere Kisten nicht brauchbar.

Effizienter ist es, die Kiste durch Backtracking verschiedener Bausteinplatzierungen zu beföl-

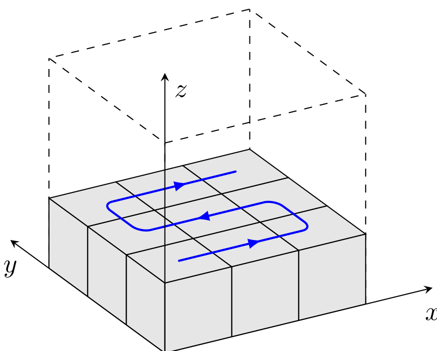


Abbildung 2.1: Beispielhafte Durchlaufrichtung (blau, entlang der Pfeile) in einem $3 \times 3 \times 3$ Behälter (gestrichelt) für die unterste Ebene mit den durch Würfel illustrierten Bausteinplätzen; die goldene Mitte ist nicht dargestellt. Um den Behälter mit Bausteinen zu füllen, beginnt man in einer Ecke, probiert entlang der blauen Linie einen Baustein unterzubringen und geht danach zur nächsten freien Stelle weiter. Passt an eine Stelle kein Baustein, muss solange zurückgegangen werden, bis ein anderer an einer vorigen Stelle eingesetzt werden kann. Ist eine Ebene gefüllt, wird das gleiche Muster auf der jeweils nächsten fortgeführt.

len. Unabhängig von der konkreten Platzierung der Bausteine muss für eine korrekte Lösung in jeder Ecke der Kiste die Ecke eines Bausteins anliegen. Es bietet sich an, dies als Ausgangspunkt zu nutzen, weil dadurch das Überprüfen von anderen Platzierungen des Bausteins entfällt. Würde man stattdessen damit anfangen, Bausteine um den goldenen Block herum zu platzieren, sind viele verschiedene Platzierungen eines Bausteins möglich.

Wir beginnen also in einer der Ecken der Kiste und platzieren dort mögliche Bausteine. Ein Baustein kann hier in verschiedenen Orientierungen anliegen, aber eine Ecke des Bausteins liegt immer an der Ecke der Kiste. Anschließend werden alle weiteren Positionen, an denen ein Baustein angelegt werden kann, wie in Abbildung 2.1 dargestellt durchlaufen. Ist eine Position noch nicht besetzt, wird dort ein noch verfügbarer Baustein in seinen möglichen Orientierungen platziert und zum nächsten Platz vorgerückt. Die in Abbildung 2.1 dargestellte Reihenfolge sorgt dafür, dass alle Positionen unterhalb und rechts vom aktuellen Punkt bereits besetzt sind. Ein neuer Baustein kann an der aktuellen Position immer nur mit einer Ecke angelegt werden. Einen Baustein der Form $(3,1,1)$ mittig anzulegen ist zum Beispiel nicht möglich, weil alle Positionen unterhalb und rechts belegt sind und auch entgegen der aktuellen Durchlaufrichtung kein Platz für die überstehende Ecke ist. Das kann auch ausgenutzt werden, um die Anzahl der zu überprüfenden Rotationen einzuschränken. Wenn eine Dimension des Bausteins in Richtung schon abgearbeiteter Positionen zeigt, muss diese Rotation nicht weiter überprüft werden.

Ansonsten ist eine Platzierung möglich, wenn der benötigte Raum nicht bereits durch einen anderen Stein eingenommen ist und wenn der angelegte Stein nicht aus der Kiste herausragt. Für das Herausragen reicht es zu überprüfen, ob eine Ecke über die Ränder der Kiste hinausgeht. Um zu überprüfen ob der Raum bereits durch einen anderen Baustein belegt ist, muss jede einzelne Position innerhalb des Bausteins darauf überprüft werden, ob sie bereits belegt ist. Eine einzige belegte Position reicht aus, um die Platzierung ungültig zu machen. Die von dem platzierten Baustein eingenommenen Positionen müssen dann als belegt markiert werden. In der goldenen Mitte befindet sich von Beginn an der goldene Stein, diese Position ist also von Anfang an als belegt markiert. Werden alle möglichen Platzierungen von Bausteinen an einer Position für ungültig befunden, gehen wir zu einer vorherigen Position zurück und versuchen, den Stein in einer anderen Orientierung oder einen anderen Stein zu platzieren.

Ausschließen redundanter Platzierungen. Bei den Rätseln sind oft mehrere Bausteine gleich. So sind beim Rätsel in der Aufgabenstellung etwa nur drei qualitativ unterschiedliche Arten von Bausteine vorhanden, insgesamt werden aber 16 aus der Beispieldatei eingelesen. Das Anlegen eines Steines sollte immer nur mit einem der gleichen Art versucht werden. Für die verschiedenen Arten müssen außerdem nicht immer alle drei räumlichen Orientierungen überprüft werden, wenn zwei oder sogar drei Seiten, wie beim Würfel, gleich sind. Sind a,b,c die drei unterschiedlichen Seitenlängen eines Bausteins, müssen die sechs Orientierungen (a,b,c), (b,a,c), (c,a,b), (a,c,b), (b,c,a) und (c,b,a) in jeweils x, y, z-Richtung überprüft werden. Sind hingegen zwei Seiten gleich, sodass der Baustein die Seitenlängen a,a,b hat, ergeben sich nur noch die drei unterschiedlichen Orientierungen (a,a,b), (a,b,a) und (b,a,a). Ist der Baustein ein Würfel mit den Seitenlängen a,a,a kommt schließlich nur die Orientierung (a,a,a) in Frage. Für die Startplatzierung muss überhaupt nur eine der möglichen Orientierungen eines Bausteins angelegt werden, da die anderen lediglich einer Rotation der gesamten würfelförmigen Kiste entsprechen würden. Entsprechend würde hier keine qualitativ neue Lösung gefunden werden. Diese Aussage gilt im Allgemeinen nicht, wenn das Problem auf nicht würfelförmige Kisten erweitert wird. Hier lassen sich nur Orientierungen ausschließen, wenn zwei Kantenlängen des Quaders gleich sind. Insgesamt ergibt sich der Algorithmus 4.

Algorithmus 4 Programm zur Lösung des Rätsels mit der goldenen Mitte

```
 1: procedure REKURSIVERSCHRITT(Position i)
 2:     if Position i ist Endposition then
 3:         Gebe Lösung aus
 4:     end if
 5:     for all Bausteinarten mit verbleibendem Block b und seinen Orientierungen o_b do
 6:         if Block b mit Orientierung o_b passt an Stelle i then
 7:             Platziere Block b an Stelle i
 8:             Markiere Block b als benutzt
 9:             REKURSIVERSCHRITT(i+1)
10:             Entferne Block b an Stelle i
11:             Markiere Block b als frei
12:         end if
13:     end for
14: end procedure
```

Überlegungen zur Laufzeit. Eine nach oben abgeschätzte Worst-Case Laufzeit von Algorithmus 4 kann mit Hilfe der folgenden Überlegung bestimmt werden. Für jeden der insgesamt n Bausteine gibt es höchstens sechs Orientierungen, die ausprobiert werden müssen. Zur Anordnung der n Bausteine gibt es $n!$ Möglichkeiten, für jede dieser Anordnungen gibt es 6^n mögliche Orientierungskonfigurationen[4] der Bausteine. Der Suchraum umfasst also zunächst $6^n \cdot n!$ Elemente. Da wir allerdings die Gleichheit von Bausteinen berücksichtigen, verringert sich diese Anzahl. Für K unterschiedliche Arten von Bausteinen mit jeweils k_i Vertretern (für $1 \leq i \leq K$) verkleinern wir den Suchraum um $\prod_{i=1}^{K} k_i!$, womit sich dessen Größe zu

$$\frac{6^n n!}{\prod_{i=1}^{K} k_i!} \tag{2.1}$$

ergibt. Die Laufzeit verhält sich also wie $\mathcal{O}(6^n \cdot n!)$.

Die Abschätzung kann noch durch die Berücksichtigung der redundanten Orientierungen verfeinert werden. Sei o_i die Anzahl der nicht-redundanten Orientierungen eines Bausteins der Art i, dann ist die Suchraumgröße

$$n! \prod_{i=1}^{K} \frac{o_i^{k_i}}{k_i!}. \tag{2.2}$$

Modellierung des Problems. Der Behälter mit den Maßen $L \times B \times H$ kann durch einen Boolean-Array mit LBH Stellen modelliert werden. Möchte man speichern und nachher ausgeben, welcher Baustein (welcher Art) wo platziert wurde, sollte der Array Nummern statt Bits fassen. Sobald ein Baustein platziert wird, werden die eingenommenen Plätze im Array markiert; anhand der aktuellen Position kann überprüft werden, ob der Block in voller Länge, Breite und Höhe in den Behälter passt. Die Maße der entsprechenden Arten von Bausteine sollten als Ganzzahlen gespeichert werden, genauso wie die Anzahl von anfänglich verfügbaren Bausteinen der jeweiligen Art. Hierbei sollten die Arten durch eine anfängliche Sortierung und Filterung der Eingabedatei bestimmt werden.

[4] Unter Berücksichtigung der oben genannten Symmetrie des Problems sind es 6^{n-1} mögliche Orientierungen, da die Orientierung des ersten Steins fixiert ist.

Wenn a_i, b_i und c_i die Seitenlängen der eingelesenen Würfel sind, müssen Kastenvolumen und Würfel die Relation

$$LBH = 1 + \sum_i a_i b_i c_i \tag{2.3}$$

erfüllen, wobei das eine zusätzliche Volumenelement durch die Existenz der goldenen Mitte kommt.

2.2 Beispiele

Die in den Beispieldateien `blockraetsel1.txt` bis `blockraetsel5.txt` bereitgestellten Rätsel sind mit Außnahme von `blockraetsel3.txt` lösbar. Für jede der lösbaren Beispiele sind in Abbildungen 2.2 bis 2.5 Schnittbilder durch eine mögliche Lösung sowie in Tabelle 2 beispielhafte Bauanleitungen zum Befüllen der Kisten gegeben. Die Lösungen sind mitunter nicht eindeutig.

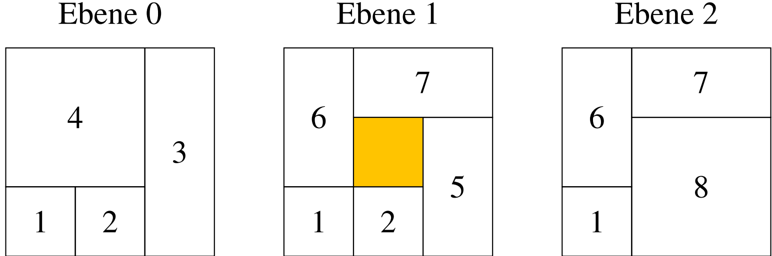


Abbildung 2.2: Schnittbilder zu `blockraetsel1.txt` nach der Anleitung in Tabelle 2a.

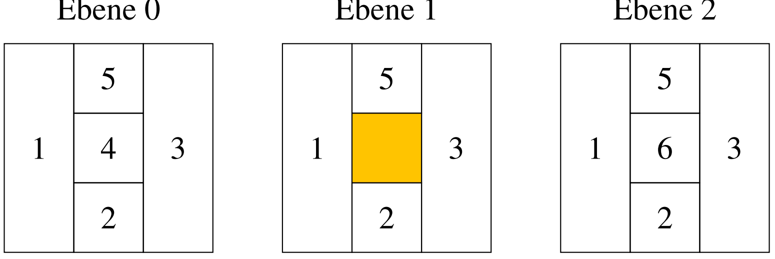


Abbildung 2.3: Schnittbilder zu `blockraetsel2.txt` nach der Anleitung in Tabelle 2b.

	Position			Maße		
Nummer	*x*	*y*	*z*	*l*	*b*	*h*
1	0	0	0	1	1	3
2	1	0	0	1	1	2
3	2	0	0	1	3	1
4	0	1	0	2	2	1
5	2	0	1	1	2	1
6	0	1	1	1	2	2
7	1	2	1	2	1	2
8	1	0	2	2	2	1

(a) `blockraetsel1.txt`

	Position			Maße		
Nummer	*x*	*y*	*z*	*l*	*b*	*h*
1	0	0	0	1	3	3
2	1	0	0	1	1	3
3	2	0	0	1	3	3
4	1	1	0	1	1	1
5	1	2	0	1	1	3
6	1	1	2	1	1	1

(b) `blockraetsel2.txt`

	Position			Maße		
Nummer	*x*	*y*	*z*	*l*	*b*	*h*
1	0	0	0	1	2	4
2	1	0	0	2	2	2
3	3	0	0	2	2	2
4	0	2	0	2	2	2
5	2	2	0	3	1	2
6	2	3	0	1	1	5
7	3	3	0	1	1	5
8	4	3	0	1	1	5
9	0	4	0	5	1	1
10	0	4	1	5	1	1
11	1	0	2	1	5	1
12	2	0	2	1	2	3
13	3	0	2	2	3	1
14	0	2	2	1	3	2
15	2	4	2	3	1	3
16	1	0	3	1	5	1
17	3	0	3	2	2	2
18	2	2	3	3	1	2
19	0	0	4	1	5	1
20	1	0	4	1	5	1

(c) `blockraetsel4.txt`

	Position			Maße		
Nummer	*x*	*y*	*z*	*l*	*b*	*h*
1	0	0	0	1	1	1
2	1	0	0	4	1	2
3	0	1	0	2	4	1
4	2	1	0	3	2	2
5	2	3	0	2	2	3
6	4	3	0	1	2	4
7	0	0	1	1	2	4
8	1	1	1	1	1	1
9	0	2	1	2	3	2
10	1	0	2	2	2	3
11	3	0	2	2	3	2
12	0	2	3	3	2	2
13	3	3	3	1	1	1
14	0	4	3	4	1	2
15	3	0	4	2	4	1
16	4	4	4	1	1	1

(d) `blockraetsel5.txt`

Tabelle 2: Anleitungen, um die Kisten der Beispieldateien mit den jeweiligen Bausteinen zu füllen. Angegeben sind jeweils die Steine anhand einer durchlaufenden Nummer mit einer Position (als x, y und z Koordinaten relativ zu einer der Ecken der Kiste), an der der jeweilige Stein zu platzieren ist, und die Orientierung des Steins mit Länge l, Breite b und Höhe h in jeweils x, y und z Richtung.

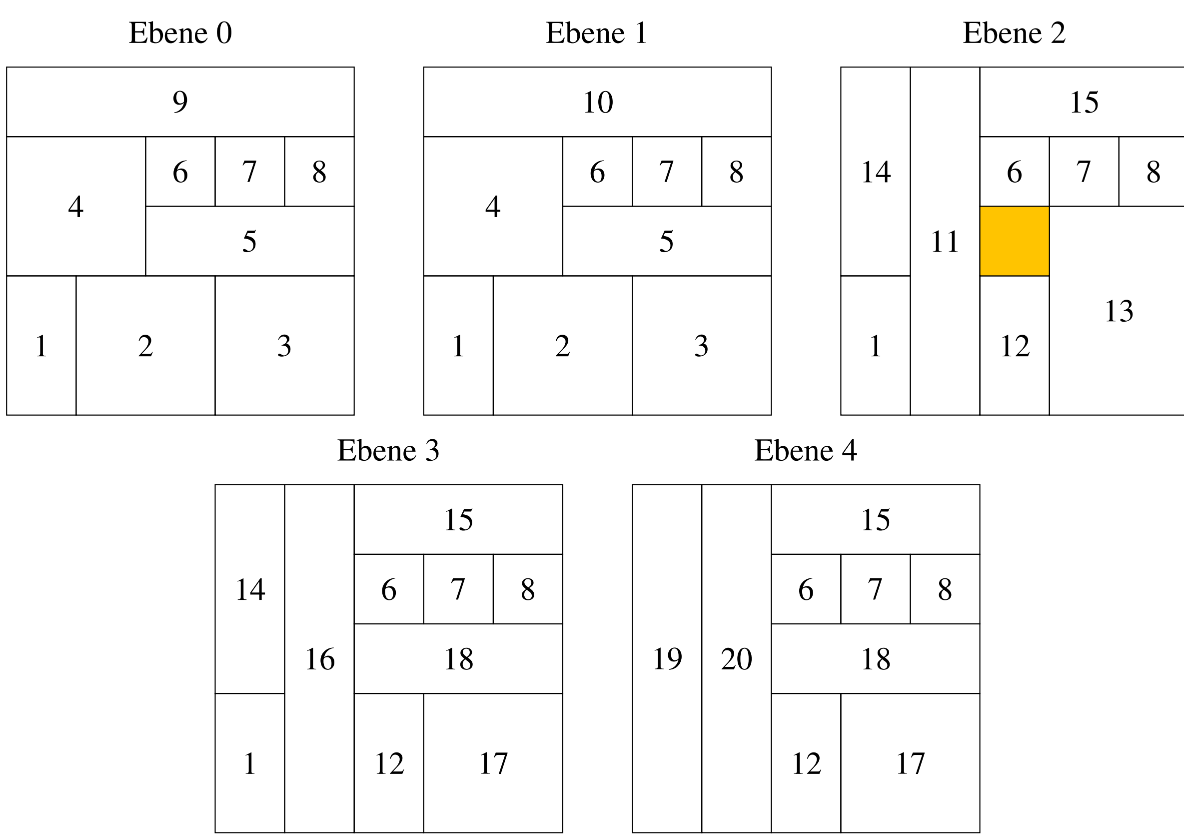


Abbildung 2.4: Schnittbilder zu `blockraetsel4.txt` nach der Anleitung in Tabelle 2c.

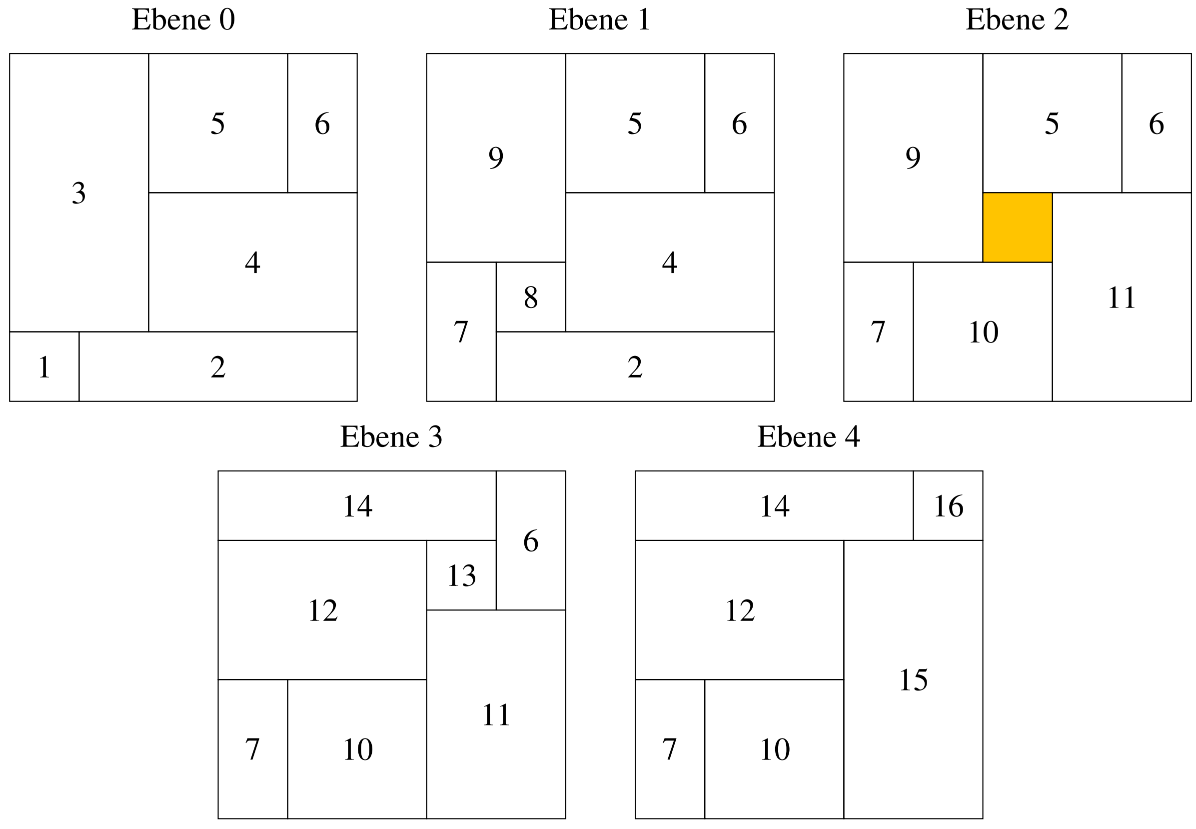


Abbildung 2.5: Schnittbilder zu `blockraetsel5.txt` nach der Anleitung in Tabelle 2d.

2.3 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
  Es soll deutlich werden, warum das Verfahren die gegebenen Rätsel lösen kann.
- $[-1]$ **Modellierung ungeeignet**
  Es muss möglich sein zu überprüfen, ob sich Steine überlappen oder aus der Kiste herausstehen.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Das Verfahren muss bestimmen können, dass ein Rätsel eventuell keine Lösung besitzt. Es muss eine gültige Zusammensetzung der Steine berechnet und ausgegeben werden, mit einem Würfel in der Mitte. Steine dürfen nicht überlappen oder aus der Kiste hinaus stehen. Für ein lösbares Rätsel muss eine Lösung gefunden werden; dafür müssen unter anderen alle möglichen Orientierungen der Steine bedacht werden. Wenn das Verfahren in der Lage ist die Kiste mit einem Teil der Bausteine zu befüllen, so gibt es keinen Punktabzug.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Ein „nacktes“ Brute-Force-Verfahren ohne Versuche von Verbesserungen ist nicht akzeptabel. Der gleiche Baustein sollte an einer Position nicht mehrfach in der selben Orientierung angelegt werden. Ebenso sollten nicht wahllos Lücken beim Befüllen der Kiste gelassen werden, die ohne System zu einem späteren Zeitpunkt gefüllt werden.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Die Platzierung der Bausteine im Würfel soll nachvollziehbar angegeben werden; ein mögliches textuelles Ausgabeformat findet sich auf der BWINF-Website. Die Ergebnisse können auch mit Hilfe von Bildern dargestellt werden, die auch von Hand erstellt sein dürfen. Beispielhaft kann eine Schnittgrafik mit den zu verwendenden Steinen angegeben werden. Eine Auflistung von Koordinaten und Bausteinen bzw. Bausteinnummern ist nicht akzeptabel.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 4 der vorgegebenen Beispiele enthalten.

Aufgabe 3: Zauberschule

3.1 Lösungsidee

Es geht darum, einen schnellstmöglichen Weg durch das Labyrinth von A nach B zu bestimmen. Wenn man so etwas noch nie gesehen hat, kann das sehr anspruchsvoll wirken. Vielleicht möchte man einfach alle möglichen Wege durchprobieren (brute force). Es gibt allerdings sehr viele davon und der Algorithmus benötigt dadurch sehr viel Zeit. Außerdem ist es nicht so einfach, alle möglichen Wege zu berechnen. Stattdessen wollen wir zwei andere Ansätze vorstellen:

Für Aufgaben, bei denen es um kürzeste Wege in einem 2D-Labyrinth geht, ist es häufig hilfreich, eine Breitensuche[5] zu verwenden. Die Idee dabei ist, erst die direkten Nachbarn des Startpunktes, danach die Nachbarn der Nachbarn und so weiter zu besuchen, bis man am Ziel ist. Das bedeutet, die Knoten werden abhängig von der Entfernung zum Startknoten abgearbeitet: zuerst alle, die eine Entfernung zum Startknoten von 1 haben, dann alle mit einer Entfernung von 2 usw. Dadurch findet man den kürzesten Weg zum Ziel jedoch nur unter der Voraussetzung, dass es immer gleich lang dauert, von einem Feld zu einem benachbarten Feld zu gelangen. Für Bugwarts ist dies leider nicht der Fall, da der Weg durch die Decke dreimal so lang ist wie der Weg zum nächsten Feld im gleichen Stockwerk.

Um das zu beheben und die Breitensuche immer noch verwenden zu können, können wir zwischen zwei übereinanderliegenden freien Feldern noch zwei virtuelle Felder einfügen. Diese Felder führen nirgendwo hin, außer dass sie die vertikale Verbindung verlängern, sodass die Breitensuche für den Weg drei Schritte braucht.

Für die Umsetzung dieser Idee eignen sich Graphen[6]. Jedes Feld wird mit einen Knoten v dargestellt. Wenn zwei freie Felder u und v innerhalb eines Stockwerks benachbart sind, verbinden wir sie mit der Kante (u,v). Wenn wir zwischen zwei benachbarten Feldern u und v aus verschiedenen Stockwerken hin und her gehen wollen, müssen wir zwei neue Knoten x und y einfügen, die keinen echten Feldern entsprechen. Dazu kommen die Kanten (u,x), (x,y) und (y,v). Dadurch wird der zusätzliche Zeitaufwand für Ron[7] simuliert.

In allen Beispielen existiert mindestens ein Weg von A nach B. Für den Fall, dass kein Weg gefunden wird, sollte aber eine Abbruchbedingung vorhanden sein, die eine Endlosschleife verhindert. Algorithmus 5 implementiert die Breitensuche.

Indem man speichert, von welchem Feld man auf ein Feld gekommen ist, kann man die kürzeste Route in linearer Zeit rekonstruieren.

Die Laufzeitkomplexität der Breitensuche ist $O(\text{Anzahl Knoten} + \text{Anzahl Kanten})$. Es gibt höchstens $4nm$ Knoten, weil es pro Stockwerk höchstens nm Felder gibt und unsere zusätzlichen Knoten höchstens zwei weitere Stockwerke bilden. Von jedem Knoten gehen höchstens 5 Kanten aus, womit die Anzahl der Kanten ebenfalls in $O(nm)$ liegt. Damit läuft der gesamte Algorithmus in $O(nm)$.

Alternative Lösung mit dem Dijkstra-Algorithmus

Alternativ kann man bei der Graphenmodellierung auch einen gewichteten Graphen verwenden, wobei alle Kanten innerhalb eines Stockwerks das Gewicht 1 haben und alle zwischen zwei

[5] https://soi.ch/wiki/bfs/

[6] https://soi.ch/wiki/graphs/

[7] Hermine hätte schließlich nicht vergessen, wie sie durch Wände kommt.

Algorithmus 5 Breitensuche für das Labyrinth

```
 1: procedure BREITENSUCHE(Startknoten s, Zielknoten z)
 2:     q ← leere FIFO-Queue
 3:     dist ← [∞ für jeden Knoten]              ▷ Anfangs ist jeder Knoten unbesucht
 4:     dist[s] ← 0                              ▷ Das Startfeld hat die Distanz 0
 5:     q.pushRight(s)
 6:     while dist[z] = ∞ and not q ist leer do  ▷ Solange das Zielfeld nicht erreicht wurde
        und es noch nicht bearbeitete Felder gibt
 7:         v ← q.popLeft()
 8:         for alle u ist Nachbar von v do
 9:             if dist[u] = ∞ then
10:                 dist[u] ← dist[v] + 1
11:                 q.pushRight(u)
12:             end if
13:         end for
14:     end while
15:     return dist[z]
16: end procedure
```

Stockwerken das Gewicht 3. Dazu muss man einen anderen Algorithmus für den kürzesten Weg verwenden, wie den Dijkstra-Algorithmus[8]. Die Laufzeit dafür wäre dann $O(nm\log(nm))$.

Dieser Ansatz lässt sich optimieren, indem man die Prioritätswarteschlange, gewissermaßen das Herz des Dijkstras, modifiziert. Da es nur die beiden Kantengewichte 1 und 3 gibt, kann es zu jedem Zeitpunkt des Algorithmus höchsten 4 verschiedene Abstände geben, die in dieser Warteschlange verwaltet werden müssen. Statt einer Warteschlange kann man damit auch für jeden der vier Werte ein Array verwenden, in denen man in $O(1)$ den nächsten Knoten findet. Damit kommt man auch hier auf eine Gesamtlaufzeit von $O(nm)$.

3.2 Beispiele

Angewandt auf die Beispiele auf der BWINF-Website liefert das Programm folgende Ausgaben. Angegeben ist für jedes Beispiel die kürzeste Distanz von *A* nach *B* sowie ein Bild, welches diesen Weg als Farbverlauf in der Schule zeigt. Der Startpunkt *A* ist Orange und geht zum Zielpunkt *B*, welcher hier pink ist. Das linke Bild ist immer das erste Stockwerk der Eingabedatei, das rechte das untere.

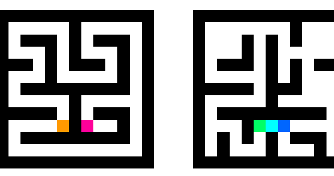


Abbildung 3.1: `zauberschule0.txt`, Kürzester Weg: 8 Sekunden.

[8] `https://soi.ch/wiki/dijkstra/`

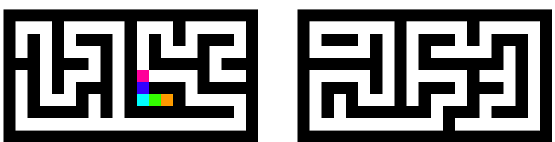


Abbildung 3.2: `zauberschule1.txt`, Kürzester Weg: 4 Sekunden.

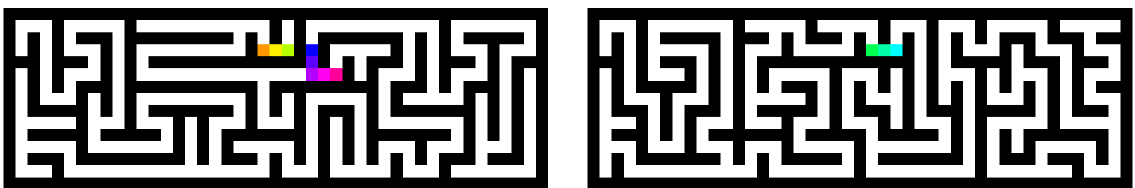


Abbildung 3.3: `zauberschule2.txt`, Kürzester Weg: 14 Sekunden.

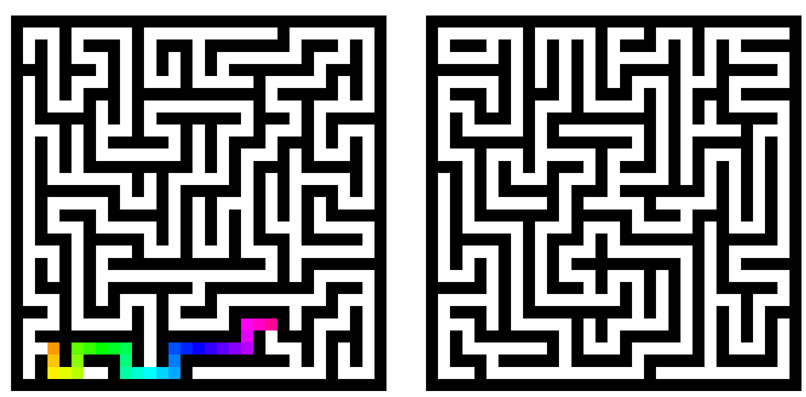


Abbildung 3.4: `zauberschule3.txt`, Kürzester Weg: 28 Sekunden.

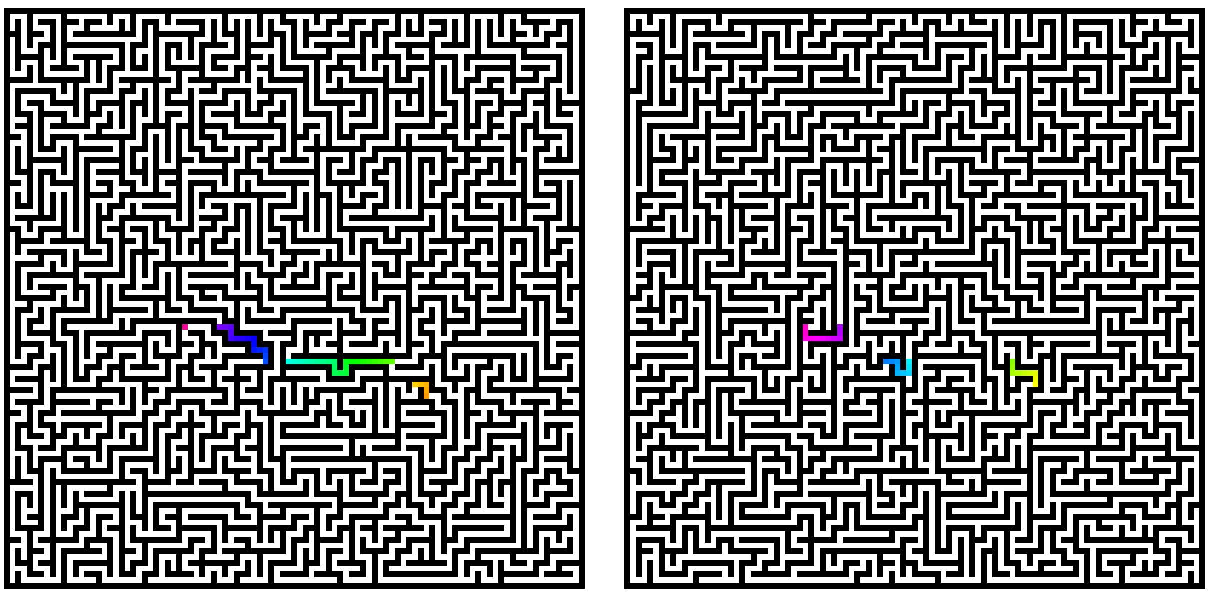


Abbildung 3.5: `zauberschule4.txt`, Kürzester Weg: 84 Sekunden.

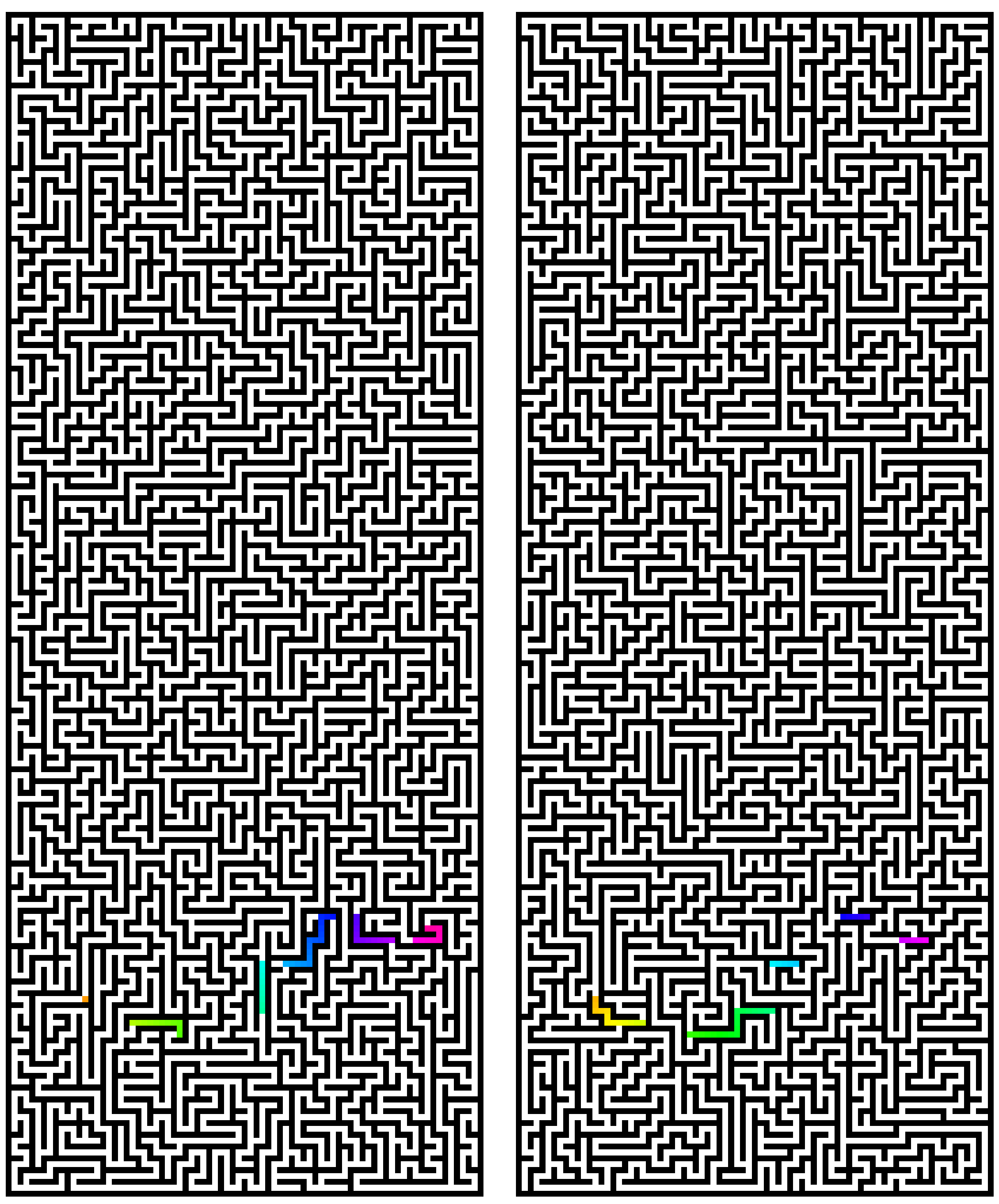


Abbildung 3.6: `zauberschule5.txt`, Kürzester Weg: 124 Sekunden.

3.3 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
  Insbesondere sollte klar werden, warum das Verfahren wirklich den kürzesten Weg ermittelt und wie es mit den unterschiedlichen Dauern der Schritte (innerhalb des gleichen Stockwerks bzw. beim Wechsel des Stockwerks) umgeht.
- $[-1]$ **Modellierung ungeeignet**
  Die Eingabe muss korrekt in eine Datenstruktur übertragen werden. Dabei muss insbesondere darauf geachtet werden, dass
  - es nicht möglich ist, durch Wände zu laufen,
  - und dass der Wechsel der Stockwerke dreimal so lange dauert wie ein Feldwechsel innerhalb eines Stockwerks.

  Es darf davon ausgegangen werden, dass Start- und Endfeld nicht in Durchgangsfeldern liegen, die aufgrund des Dateieingabeformates existieren. Auch darf davon ausgegangen werden, dass A und B auf der gleichen Ebene liegen. Wenn jemand versucht die Eingabe auf ein Format wie auf dem Aufgabenblatt zurück zurechnen (Wände haben Dicke 0), gibt es keinen Punktabzug.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Die Länge des kürzesten Weges soll korrekt sein (keine Näherung). Der kürzeste Weg soll tatsächlich so möglich sein und hat die angegebene kürzeste Länge. Felder können nicht diagonal durchlaufen werden.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Das gewählte Verfahren sollte nicht langsamer sein als $O(n^2m^2)$
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Für jedes Ergebnis sollte die Länge des kürzesten Weges angegeben sein, sowie mindestens für die vorgegebenen Beispiele `zauberschule0.txt` bis `zauberschule3.txt` auch der konkrete Weg. Der Weg muss nachvollziehbar dargestellt werden, eine Liste von Koordinaten ist nicht in Ordnung, eine Liste von Richtungen hingegen schon.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 5 der vorgegebenen Beispiele 0 bis 5 enthalten.

Aufgabe 4: Nandu

4.1 Lösungsidee

In dieser Aufgabe betrachten wir eine Konstruktion der Breite n und der Höhe m, wobei die Bausteine waagrecht platziert werden. Die erste Reihe enthält Lichtquellen. Die Bausteine lassen sich reihenweise abspeichern. Die letzte Reihe der Bausteine stellt die Ausgabelichter dar. Die interne Repräsentation der Bausteine im Programmcode stellt sich allerdings aufgrund der Breite von zwei etwas schwieriger dar, da man nicht jeden Baustein genau einem Feld zuordnen kann. Man kann aber beispielsweise für jeden Baustein die Position des linken Teils speichern. Für die roten Bausteine lässt sich die Ausrichtung durch die Verwendung zweier verschiedener Typen realisieren.

Einlesen und Datenstrukturen

Das Eingabeformat gibt für jedes Feld den Typ des darauf liegenden Bausteins an. Dadurch, dass alle Bausteine eine Breite von zwei haben, lässt sich durch das gegebene Eingabeformat für ein Feld nicht direkt feststellen, in welche Richtung sich der darauf liegende Baustein erstreckt. Dies ist insbesondere dann der Fall, wenn zwei Bausteine des gleichen Typs nebeneinander liegen. Ein Beispiel wären zwei aneinander liegende weiße Bausteine, die folglich mit `W W W W` in der Eingabe abgebildet werden. Würde man also willkürlich ein Feld und seinen Nachbarn betrachten, wäre nicht klar, ob die beiden Felder zu demselben Baustein gehören.

Zur Lösung bietet es sich an, über jede Reihe von links nach rechts (bzw. von rechts nach links) zu iterieren. Immer, wenn man auf einen Baustein statt auf ein nicht belegtes Feld `X` trifft, weiß man, dass dort ein neuer Baustein beginnt und dass dieser sich ein weiteres Feld nach rechts erstreckt. In diesem Fall überspringt man das rechte Feld des betrachteten Bausteins, denn dort fängt dementsprechend kein neuer Baustein an. Dadurch kann man die verwendeten Bausteine eindeutig aus der Aufgabe rekonstruieren; auch die beiden Arten roter Bausteine lassen sich alleine anhand des ersten Feldes bestimmen. Die eingelesenen Bausteine kann man nun direkt in das oben beschriebene Eingabeformat abspeichern.

Erzeugen der Eingabezustände

Zunächst muss man alle Konstellationen aufzählen, welche die Taschenlampen haben können, also welche Lampen an bzw. aus sind. Sei Q die Menge der Taschenlampen. Dann gibt es insgesamt $2^{|Q|}$ Möglichkeiten, denn jede Taschenlampe ist entweder an oder aus. Dies könnte man beispielsweise mittels Rekursion realisieren, wo in jedem Rekursionsschritt der Zustand einer Taschenlampe festgelegt wird. Je nach Programmiersprache gibt es hier aber schon passende Bibliotheksfunktionen, die man aufrufen kann.[9]

Simulation einer Eingabe

Hat man eine Konfiguration der Taschenlampen gegeben, muss man nun simulieren, welchen Zustand die geforderten LEDs am Ende haben. Hier kann man ausnutzen, dass man schrittweise

[9] In Python kann man die Zustände beispielsweise mittels `itertools.product([False, True], repeat=num_lights)` erzeugen.

über jede Reihe gehen kann, denn das Verhalten jedes Bausteins ist nur von der vorherigen Reihe abhängig. Zusätzlich geht man davon aus, dass ein Licht nur weitergegeben wird, wenn der Sensor in der nächsten Reihe direkt von einer Taschenlampe oder einem anderen Block bestrahlt wird, ohne einen Hohlraum dazwischen.

Mit Algorithmus 6 kann für eine Bausteinreihe *reihe* bestimmt werden, an welchen Positionen sie Licht an die nächste Reihe weitergibt – *lichter_reiheAktuell* sind sozusagen die „Licht-Ausgabe-Positionen“ der Reihe. Dabei ist *lichter_reiheVor* die Menge der Positionen, an der in der vorigen Reihe LEDs leuchten, also die Menge der „Licht-Eingabe-Positionen“ für die aktuelle Reihe. Ein Baustein hat immer einen *typ*, z.B. Weiß, und zwei Sensoren, *Baustein.sensorRechts* und *Baustein.sensorLinks*. Am Anfang sind das die Positionen der Quellen. Wenn also die 1 in *lichter_reiheVor* vorhanden ist, bedeutet es, dass auf den Sensor an Position 1 ein Licht fällt.

Der Algorithmus iteriert über die vorhin vorbereitete Liste der Bausteine in jeder Reihe und bestimmt die neuen aktiven Indizes der Lichter basierend auf der vorherigen Reihe. Das Verhalten der gesamten Konstruktion kann nun simuliert werden, indem dieser Algorithmus schrittweise auf die Bausteinreihen angewandt wird.

Algorithmus 6 Bestimmen der Licht-Ausgabe-Positionen einer Reihe

```
procedure SIMULATEROW(reihe, lichter_reiheVor)
    lichter_reiheAktuell ← ∅
    for each Baustein in reihe do
        if Baustein.typ is Rot_sensorRechts then
            if Baustein.sensorRechts is not in lichter_reiheVor then
                add Baustein.linkesLicht and Baustein.rechtesLicht to lichter_reiheAktuell
            end if
        else if Baustein.typ is Rot_sensorLinks then
            if Baustein.sensorLinks is not in lichter_reiheVor then
                add Baustein.linkesLicht and Baustein.rechtesLicht to lichter_reiheAktuell
            end if
        else if Baustein.typ is Weiß then
            if Baustein.sensorLinks or Baustein.sensorRechts is in lichter_reiheVor then
                add Baustein.linkesLicht and Baustein.rechtesLicht to lichter_reiheAktuell
            end if
        else if Baustein.typ is Blau then
            if Baustein.sensorLinks is in lichter_reiheVor then
                add Baustein.linkesLicht to lichter_reiheAktuell
            end if
            if Baustein.sensorRechts is in lichter_reiheVor then
                add Baustein.rechtesLicht to lichter_reiheAktuell
            end if
        end if
    end for
    return lichter_reiheAktuell
end procedure
```

Laufzeitkomplexität

Die Laufzeit der Operationen ist davon abhängig, welche Datenstrukturen man für die definierten Mengen verwendet. Sinnvoll ist es, für die Indizes-Mengen der Quellen Q und der Baustein Sensoren ein Bitarray (d.h. ein Array von Bits, dessen i-tes Bit angibt, ob das mit i indizierte Element in der Menge enthalten ist) der Länge n zu verwenden, bei der man ein Element in $\mathcal{O}(1)$ auslesen oder ersetzen kann. Eine Reihe speichert man am besten als Liste ab, da man lediglich über diese iterieren muss.

Für jede Reihe ergäbe sich dann eine Laufzeit von $\mathcal{O}(n)$ für das Aufsetzen der leeren Reihe sowie $\mathcal{O}(|reihe|)$ für die Simulation der Bausteine. Nachdem $|reihe| \leq n/2$ aufgrund der Bausteingröße gilt, lässt sich der Gesamtaufwand für eine Reihe mit $\mathcal{O}(n)$ und somit für das gesamte Feld bei einem festgelegten Taschenlampenzustand mit $\mathcal{O}(nm)$ abschätzen. Unter Betrachtung der erzeugten Eingabezustände folgt ein Gesamtaufwand von $\mathcal{O}(2^{|Q|} \cdot nm)$.

4.2 Beispiele

Im Folgenden wird für eine eingeschaltete Taschenlampe/LED eine 1 verwendet, im ausgeschalteten Zustand eine 0.

nandu1.txt

Q1	Q2	L1	L2
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

nandu2.txt

Q1	Q2	L1	L2
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

nandu3.txt

Q1	Q2	Q3	L1	L2	L3	L4
0	0	0	1	0	0	1
0	0	1	1	0	0	0
0	1	0	1	0	1	1
0	1	1	1	0	1	0
1	0	0	0	1	0	1

1	0	1	0	1	0	0
1	1	0	0	1	1	1
1	1	1	0	1	1	0

nandu4.txt

Q1	Q2	Q3	Q4	L1	L2
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	1
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	1
1	1	1	1	0	0

nandu5.txt

Q1	Q2	Q3	Q4	Q5	Q6	L1	L2	L3	L4	L5
0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	0	1	1	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	0	1	0	1	0	0	1	0	0
0	0	0	1	1	0	0	0	0	1	1
0	0	0	1	1	1	0	0	0	1	1
0	0	1	0	0	0	0	0	0	1	0
0	0	1	0	0	1	0	0	0	1	0
0	0	1	0	1	0	0	0	0	1	1
0	0	1	0	1	1	0	0	0	1	1
0	0	1	1	0	0	0	0	1	0	0
0	0	1	1	0	1	0	0	1	0	0
0	0	1	1	1	0	0	0	0	1	1
0	0	1	1	1	1	0	0	0	1	1
0	1	0	0	0	0	0	0	0	1	0

0	1	0	0	0	1	0	0	0	1	0
0	1	0	0	1	0	0	0	0	1	1
0	1	0	0	1	1	0	0	0	1	1
0	1	0	1	0	0	0	0	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	0	1	1	0	0	0	0	1	1
0	1	0	1	1	1	0	0	0	1	1
0	1	1	0	0	0	0	0	0	1	0
0	1	1	0	0	1	0	0	0	1	0
0	1	1	0	1	0	0	0	0	1	1
0	1	1	0	1	1	0	0	0	1	1
0	1	1	1	0	0	0	0	1	0	0
0	1	1	1	0	1	0	0	1	0	0
0	1	1	1	1	0	0	0	0	1	1
0	1	1	1	1	1	0	0	0	1	1
1	0	0	0	0	0	1	0	0	1	0
1	0	0	0	0	1	1	0	0	1	0
1	0	0	0	1	0	1	0	0	1	1
1	0	0	0	1	1	1	0	0	1	1
1	0	0	1	0	0	1	0	1	0	0
1	0	0	1	0	1	1	0	1	0	0
1	0	0	1	1	0	1	0	0	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	0	0	0	1	0	0	1	0
1	0	1	0	0	1	1	0	0	1	0
1	0	1	0	1	0	1	0	0	1	1
1	0	1	0	1	1	1	0	0	1	1
1	0	1	1	0	0	1	0	1	0	0
1	0	1	1	0	1	1	0	1	0	0
1	0	1	1	1	0	1	0	0	1	1
1	0	1	1	1	1	1	0	0	1	1
1	1	0	0	0	0	1	0	0	1	0
1	1	0	0	0	1	1	0	0	1	0
1	1	0	0	1	0	1	0	0	1	1
1	1	0	0	1	1	1	0	0	1	1
1	1	0	1	0	0	1	0	1	0	0
1	1	0	1	0	1	1	0	1	0	0
1	1	0	1	1	0	1	0	0	1	1
1	1	0	1	1	1	1	0	0	1	1
1	1	1	0	0	0	1	0	0	1	0
1	1	1	0	0	1	1	0	0	1	0
1	1	1	0	1	0	1	0	0	1	1
1	1	1	0	1	1	1	0	0	1	1
1	1	1	1	0	0	1	0	1	0	0
1	1	1	1	0	1	1	0	1	0	0
1	1	1	1	1	0	1	0	0	1	1

1	1	1	1	1	1	1	0	0	1	1

4.3 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**

- $[-1]$ **Modellierung ungeeignet**
  Es sollte klar werden, wie die Bausteine und die gesamte Konstruktion im Programm repräsentiert werden.
- $[-1]$ **Zustände der Taschenlampen fehlerhaft aufgezählt**
  Das Programm soll alle $2^{|Q|}$ möglichen Zustände der Taschenlampen korrekt aufzählen und anschließend testen. Dafür darf auch eine Bibliotheksfunktion verwendet werden. Wenn eine Taschenlampe auf Grund der Konstruktion nicht berücksichtigt werden muss, ist das in Ordnung diese wegzulassen.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Das Verhalten der einzelnen Bausteine soll korrekt eingehalten und Abhängigkeiten zwischen hintereinander geschalteten Bausteinen sollen berücksichtigt werden (beispielsweise, indem man das Modell reihenweise simuliert). Letztlich müssen die Simulationsergebnisse korrekt sein. Es ist in Ordnung, wenn ein `X` als Hohlraum gewertet und das Licht weiter gegeben wird.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Das gewählte Verfahren sollte pro Zustand der Lichtquellen (Taschenlampen) nicht langsamer sein als $\mathcal{O}(n^2m)$, wo beispielsweise zu jedem Feld über die möglichen Bausteine iteriert wird. Bei einer noch schlechteren Laufzeit wird ein Punkt abgezogen.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Die Ergebnisse sollen z. B. in einer Tabelle mit allen möglichen Zuständen der Taschenlampen und den zugehörigen LEDs dargestellt werden. Andere Darstellungen mit identischem Informationsgehalt und ähnlicher Übersichtlichkeit werden aber auch akzeptiert.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 4 der 5 vorgegebenen Beispiele enthalten.

Aufgabe 5: Stadtführung

5.1 Lösungsidee

Vorbemerkung

Der erste, nicht unwesentliche Teil der Aufgabe besteht darin zu verstehen, was eigentlich das Ziel ist und welche Operationen man ausführen darf. Gegeben ist eine Stadtführung in chronologischer Reihenfolge. Gesucht ist eine gekürzte Stadtführung, die zum einen diese Chronologie beibehält und zum anderen eine Auswahl an essentiellen Tourpunkten besucht. Eine Neuordnung der Tourpunkte, bei der die Jahreszahlen nicht mehr nur aufsteigen, ist nicht erlaubt. Man kann also einen essentiellen Tourpunkt nicht erst auslassen, um ihn später nachzuholen. Beim Kürzen dürfen nur geschlossene Teiltouren gestrichen werden, also diejenigen Wegabschnitte, die zwischen zwei Tourpunkten am gleichen Ort in der ursprünglichen Tour gegangen werden. Nach obiger Überlegung darf diese Teiltour keine essentiellen Tourpunkte enthalten.

Das Schwierige an einer Lösung dieser Aufgabe ist, dass es sehr viele verschiedene Fälle geben kann und man ein Programm entwickeln muss, welches mit allen gut zurechtkommt und die tatsächliche kürzeste Tour findet. Ein einfacher Greedy-Algorithmus, der beispielsweise immer die größte geschlossene Teiltour auswählt und entfernt, wird auf vielen Instanzen eine falsche Antwort liefern, da es zwei kleinere Teiltouren geben kann, die zusammen eine größere Ersparnis liefern (zur Illustration siehe Abbildung 5.1). Die beiden kurzen Pfade jeweils von A nach A, in der Abbildung blau dargestellt, kürzen die Tour um $6+3+2+3 = 14$ Einheiten, während der größere Pfad von B nach B, dargestellt in orange, nur um $3+2+3+5 = 13$ Einheiten kürzt.

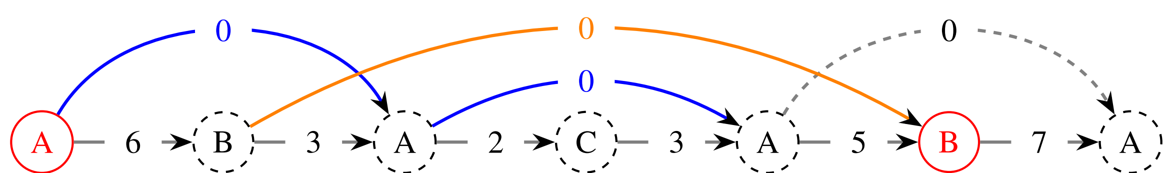


Abbildung 5.1: Beispielhaft einige Tourpunkte in chronologischer Reihenfolge. Die Zahlen entlang der Pfeile sind die Abstände zwischen den Orten. Nicht essentielle Tourpunkte sind gestrichelt dargestellt, essentielle Tourpunkte sind rot. Die horizontalen Pfeile zeigen die ursprüngliche Tour an, während die gebogenen Pfeile die Abkürzungen durch das Entfernen von Teiltouren darstellen. Gestrichelt sind jene geschlossenen Teiltouren, die einen essentiellen Tourpunkt enthalten und entsprechend nicht gekürzt werden dürfen.

Nun kann man natürlich probieren, das auszubessern und eine Fallunterscheidung einzubauen, in der man prüft, ob es eine sparsamere Möglichkeit innerhalb der größten Teiltour gibt, und dann das Bessere von beiden wählt. Jedoch kann man sich leicht vorstellen, dass dies gerade mit stärker verschachtelten Teiltouren sehr schnell sehr aufwändig und unübersichtlich wird. Damit wird auch die Gefahr für Fehler immer größer, dass doch ein Fall übersehen wurde.

Stattdessen wollen wir einige Beobachtungen zu dem Problem anstellen, aus denen wir einen Algorithmus ableiten können, der verhältnismäßig leicht zu implementieren ist und sogar in linearer Zeit läuft.

Beobachtungen

Vorneweg sei vorausgesetzt, dass es mindestens zwei essentielle Tourpunkte an mindestens zwei verschiedenen Orten gibt, da wir sonst einfach eine stationäre Tour ohne Laufanteil als optimale Lösung ausgeben können.

Wir können die für eine Kürzung der Tour in Frage kommenden geschlossenen Teiltouren eingrenzen auf jene, die keinen essentiellen Punkt enthalten, da wir keine essentiellen Tourpunkte entfernen dürfen. Ferner können wir die Teiltouren auf die minimal kleinen Teiltouren eingrenzen. Jedoch könnte der Start- und Endpunkt der geschlossenen Teiltour möglicherweise essentiell sein, weil er in der Tour enthalten bleibt.

Veränderung des Start- und Endpunktes

Wenn wir den Begriff der Teiltour streng auslegen und ausschließlich *geschlossene* Teiltouren betrachten (im Sinne des in der Aufgabenstellung beschriebenen „Schlenkers“), dann kann sich der Start- und Endpunkt der Tour nicht verändern.

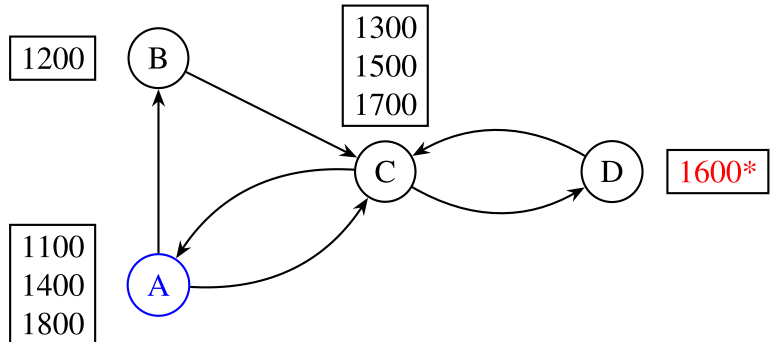


Abbildung 5.2: Beispieltour für eine Stadtführung mit Sehenswürdigkeiten A, B, C und D. Start- und Endpunkt ist blau markiert. Essentielle Tourpunkte sind rot und mit Sternchen markiert.

Betrachten wir dafür das Beispiel in Abbildung 5.2, mit dem einzigen essentiellen Tourpunkt D 1600. Das Bild beschreibt die Tour

$$A\ 1100 \to B\ 1200 \to C\ 1300 \to A\ 1400 \to C\ 1500 \to D\ 1600 \to C\ 1700 \to A\ 1800$$

Da an den Orten A, B und C keine essentiellen Tourpunkte sind, könnte man denken, dass sich die Tour zu einer kürzesten Teiltour

$$C\ 1500 \to D\ 1600 \to C\ 1700$$

kürzen ließe[10]. Damit würde man allerdings am Anfang die Teiltour

$$A\ 1100 \to B\ 1200 \to C\ 1300 \to A\ 1400 \to C\ 1500$$

weglassen (also den Schlenker $C\ 1300 \to A\ 1400 \to C\ 1500$ und das Anfangsstück $A\ 1100 \to B\ 1200 \to C\ 1300$) sowie das Endstück $C\ 1700 \to A\ 1800$. Beides sind keine *geschlossenen* Teil-

[10] In diesem Extremfall einer Tour mit nur einem essentiellen Tourpunkt kann man sogar der Auffassung sein, dass die kürzeste Teiltour allein aus diesem einen Tourpunkt besteht; auch diese Auffassung lassen wir gelten

touren, da ihre Start- und Endpunkte unterschiedlich sind. Lässt man wirklich nur geschlossene Teiltouren weg, kann man höchstens

$$A\ 1100 \rightarrow B\ 1200 \rightarrow C\ 1300 \rightarrow A\ 1400$$

weglassen, also nur zu dieser neuen Tour kürzen:

$$A\ 1400 \rightarrow C\ 1500 \rightarrow D\ 1600 \rightarrow C\ 1700 \rightarrow A\ 1800$$

Man kann allerdings den Begriff der (geschlossenen) Teiltour etwas „freier“ auslegen und argumentieren, dass eine Verkürzung auch durch das Weglassen allgemeinerer Teiltouren (oder durch Weglassen geschlossener Teiltouren plus Anfangs- und Endstücken) entstehen kann. Schließlich legt die Aufgabenstellung nahe, dass sich der Startpunkt verändern kann:

> Der Startort der ursprünglichen Tour ist aber nicht unbedingt essentiell, und Alina hätte nichts dagegen, einen neuen Startort zu wählen.

Wenn der Startpunkt sich doch ändern soll, ist es wichtig, dass die Tour am gleichen Punkt startet, an dem sie endet. Dafür fügen wir einen Weg vom Endpunkt zum Startpunkt mit Länge 0 ein und betrachten die Tour als Kreis anstelle einer Strecke. Nun berechnen wir die kürzeste Entfernung vom letzten zum ersten essentiellen Punkt. Der neue Startpunkt ist nun der erste Punkt der ursprünglichen Tour, der auch auf der neuen gekürzten Kreistour liegt. Da beim Entfernen von geschlossenen Teiltouren anschließend zwei Tourpunkte mit dem gleichen Ort direkt aufeinanderfolgen, ist auch sichergestellt, dass die neue Tour am gleichen Ort startet und endet, nämlich an den Endpunkten der gekürzten geschlossenen Teiltour, die den ursprünglichen Startpunkt beinhaltet oder entfernt hat.

Insgesamt reduziert sich das Problem darauf, zwischen je zwei aufeinanderfolgenden essentiellen Tourpunkten den kürzesten Weg unter der Operation des Entfernens von geschlossenen Teiltouren zu finden. Die Summe dieser Teilprobleme ist dann die Lösung für die Aufgabe.

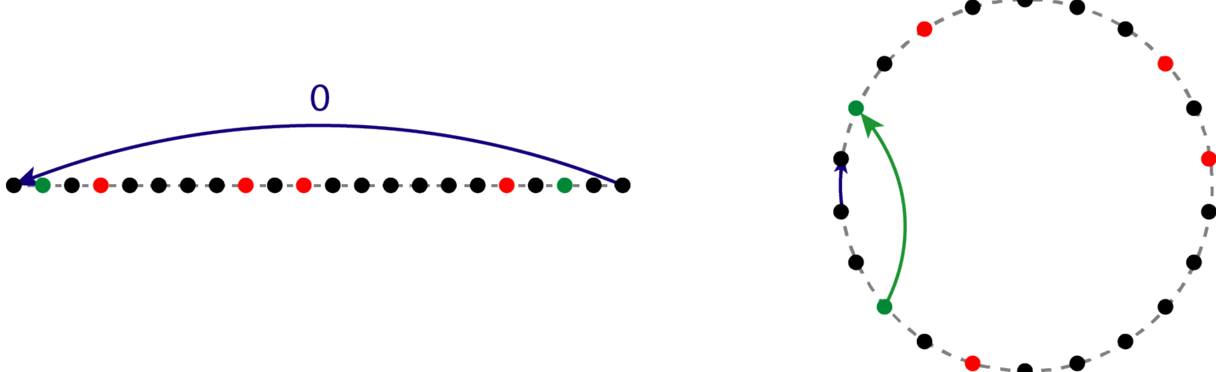


Abbildung 5.3: Um einen potentiell neuen Startpunkt für die Tour herauszufinden betrachten wir die Tour als Kreis (siehe dunkelblauer Pfeil) und berechnen den kürzesten Weg von dem letzten zum ersten essentiellen (roten) Punkt der Tour. Dabei nehmen wir eine Kürzung über den grünen Pfeil vor. In der neuen Tour wären diese grünen Punkte die neuen Start und Endpunkte.

Kürzester Weg

Von jedem Tourpunkt aus kann man der ursprünglichen Tour folgen und zum nächsten Tourpunkt gehen. Dies kostet so viel Zeit, wie in der ursprünglichen Tour. Alternativ kann es sein, eine geschlossene Teiltour zu kürzen. Voraussetzung dafür ist, dass es einen weiteren Tourpunkt am gleichen Ort gibt, der vor oder auf dem nächsten essenziellen Tourpunkt liegt. Es gibt also von jedem Knoten aus höchstens 2 mögliche Wege, zwischen denen man sich entscheiden kann. Ob die Abkürzung (falls vorhanden) besser ist als der direkte Weg, hängt davon ab, welche anderen verschachtelten Teiltouren es gibt.

Wenn wir jedoch die Wege betrachten, durch die ein Knoten erreicht werden kann, lässt sich das Problem noch weiter unterteilen. Der kürzeste Weg zu einem Knoten E kann berechnet werden, indem der kürzeste Weg zu allen vorangegangenen Knoten berechnet wird, die eine direkte Verbindung zu E haben. Wenn die Entfernung dieser vorangegangenen Knoten bereits bekannt ist, reicht es, die Gewichte der Wege zu diesen Knoten aufzuaddieren und den Weg mit geringeren Kosten zu nehmen.

Der 3. Knoten von links in Abbildung 5.3 beispielsweise ist durch 2 Wege zu erreichen. Einmal über den direkten Weg, vom 2. Knoten ausgehend und einmal über die Abkürzung direkt vom Start-Knoten. Der kürzeste Weg zum 3. Knoten ist einfach berechenbar, wenn bereits bekannt ist, wie groß die kürzeste Entfernung zum 1. und 2. Knoten sind. In diesem Fall ist der 1. Knoten bereits der Startknoten und der 2. Knoten hat eine Entfernung von 6 zum Startknoten.

Um den kürzesten Weg zwischen 2 essenziellen Knoten zu berechnen, kann man die Tour vom Start aus abgehen und bei jeder Abkürzung, die gefunden wird überprüfen, welcher der beiden Optionen für den aktuellen Knoten kostengünstiger ist. Um eine Abkürzung zu identifizieren, muss gespeichert werden, welche Orte bereits besucht wurden. Ist der aktuelle Knoten an einem Ort, der bereits besucht wurde, ist eine Abkürzung möglich. Um entscheiden zu können, welcher Weg zum aktuellen Knoten am günstigsten ist, müssen die Kosten zu den bisherigen Knoten gespeichert werden. Mit diesen Informationen kann für jeden Knoten, der 2 eingehende Wege hat, entschieden werden, welcher gewählt wird, und damit im Nachhinein der kürzeste Weg rekonstruiert werden.

Alternativer Ansatz mit Dijkstra

Um das ganze stattdessen als kürzeste-Wege-Problem in einem Graphen zu betrachten, kann man für jeden Knoten die beiden eingehenden Wege als Kanten in einem Graph auffassen (siehe zur Veranschaulichung auch Abbildung 5.3). Auf dem gewichteten Graph muss man nun noch einen Algorithmus für den kürzesten Pfad anwenden (z.B. den Dijkstra-Algorithmus[11]). Beide Ansätze führen zu sehr ähnlichen Algorithmen.

Für den Fall, dass es drei (oder mehr) Tourpunkte gibt, die den gleichen Ort A besuchen, so reicht es, Kanten zwischen dem ersten und dem zweiten sowie dem zweiten und dem dritten einzuzeichnen. Die Kante vom ersten zum dritten ergibt sich einfach, indem man die anderen beiden Kanten direkt hintereinander anwendet. Damit garantiert dieser Ansatz zusätzlich, dass die Anzahl der Tourpunkte auf einer minimal kurzen Tour maximiert wird (ohne die Zeit zu verlängern), was nach einem guten Nebeneffekt für Alina klingt.

Für dieses konkrete Problem lässt sich die Laufzeit von $O(n \log n)$ des Dijkstra-Algorithmus' noch auf $O(n)$ verbessern. Dafür nutzen wir dynamische Programmierung und verwenden den

[11] `https://soi.ch/wiki/dijkstra/`

Graphen nur implizit. Algorithmus 7 skizziert eine Implementierung. Dabei hat das Intervall, auf dem wir den Weg berechnen, die Länge n, und nur der erste und letzte Tourpunkt sind essentiell. Das Array `dpDist[i]` stellt die Länge der Strecke vom linken Ende des Intervalls bis i dar.

Algorithmus 7 Kürzester Weg mit Entfernen geschlossener Teiltouren

1: **procedure** KÜRZESTERWEG
2: dpDist ← [∞ für alle n] ▷ Speichert für jeden Tourpunkt den Abstand zum Start
3: dpDist[0] ← 0
4: lastLoc ← [none für alle Orte] ▷ Speichert den letzten Tourpunkt für jeden Ort
5: **for** $i = 1 \dots n-1$ **do**
6: dpDist[i] ← dpDist[$i-1$] + Abstand($i-1$, i) ▷ alte Tour
7: **if** lastLoc[Ort von i] ≠ none **and** dpDist[i] > dpDist[lastLoc[Ort von i]] **then**
8: dpDist[i] ← dpDist[lastLoc[Ort von i]] ▷ bessere Abkürzung nach i
9: **end if**
10: lastLoc[Ort von i] = i
11: **end for**
12: **return** dpDist[$n-1$]
13: **end procedure**

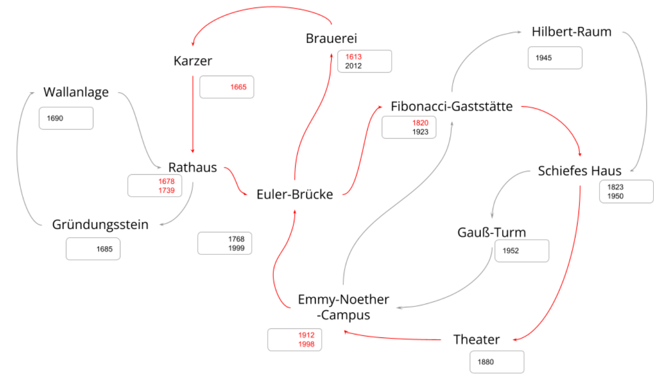
Dieser Algorithmus berechnet in erster Linie die kürzeste Distanz als Zahl. Wenn man jedoch alle `dpDist` Werte speichert, kann man daraus in linearer Zeit die genaue Tour rekonstruieren.

5.2 Beispiele

Angewandt auf die Eingaben auf der BWINF-Website liefert das Programm folgende gekürzte Touren. Ausgegeben werden jeweils alle praktisch relevanten Informationen für die neue Tour: die Strecke, die Anzahl der Tourpunkte, sowie die Tourpunkte selbst mit der Information, ob sie essentiell sind. Die Touren wurden unter der Annahme erstellt, dass der Startpunkt sich verändern kann.

tour1.txt

```
Routenlänge: 1020 (davor 2060)
Anzahl Tourpunkte: 12 (davor 18)
Brauerei (1613) X
Karzer (1665) X
Rathaus (1678) X
Rathaus (1739) X
Euler-Brücke (1768)
Fibonacci-Gaststätte (1820) X
Schiefes Haus (1823)
Theater (1880)
Emmy-Noether-Campus (1912) X
Emmy-Noether-Campus (1998) X
Euler-Brücke (1999)
Brauerei (2012)
```



In rot sind die essentiellen Tourpunkte (Ort in Kombination mit Datum) markiert, sowie die Wege die auch bei der gekürzten Tour erhalten bleiben.

tour2.txt

```
Routenlänge: 1020 (davor 2060)
Anzahl Tourpunkte: 12 (davor 18)
Brauerei (1613)
Karzer (1665) X
Rathaus (1678)
Rathaus (1739)
Euler-Brücke (1768)
Fibonacci-Gaststätte (1820) X
Schiefes Haus (1823)
Theater (1880)
Emmy-Noether-Campus (1912) X
Emmy-Noether-Campus (1998) X
Euler-Brücke (1999)
Brauerei (2012)
```

tour3.txt

```
Routenlänge: 2670 (davor 4560)
Anzahl Tourpunkte: 10 (davor 14)
Talstation (1768)
Wäldle (1805)
Mittlere Alp (1823)
Observatorium (1833)
Observatorium (1874) X
Piz Spitz (1898)
Panoramasteg (1912) X
Panoramasteg (1952)
Ziegenbrücke (1979) X
Talstation (2005)
```

tour4.txt

```
Routenlänge: 1420 (davor 3200)
Anzahl Tourpunkte: 16 (davor 29)
Marktplatz (1549)
Marktplatz (1562)
Springbrunnen (1571)
Dom (1596) X
Bogenschütze (1610)
Bogenschütze (1683)
Schnecke (1698) X
Fischweiher (1710)
Reiterhof (1728) X
Schnecke (1742)
Schmiede (1765)
```

```
Große Gabel (1794)
Große Gabel (1874)
Fingerhut (1917) X
Stadion (1934)
Marktplatz (1962)
```

tour5.txt

```
Routenlänge: 2620 (davor 5000)
Anzahl Tourpunkte: 29 (davor 42)
Gabelhaus (1638)
Gabelhaus (1699)
Hexentanzplatz (1703) X
Eselsbrücke (1711)
Dreibannstein (1724)
Dreibannstein (1752)
Schmetterling (1760) X
Dreibannstein (1781)
Märchenwald (1793) X
Märchenwald (1840)
Eselsbrücke (1855)
Eselsbrücke (1877)
Reiterdenkmal (1880)
Riesenrad (1881)
Riesenrad (1902)
Dreibannstein (1911) X
Olympisches Dorf (1924)
Haus der Zukunft (1927) X
Stellwerk (1931)
Stellwerk (1942)
Labyrinth (1955)
Gauklerstadl (1961)
Planetarium (1971) X
Känguruhfarm (1976)
Balzplatz (1978)
Dreibannstein (1998) X
Labyrinth (2013)
CO2-Speicher (2022)
Gabelhaus (2023)
```

Bei Route 4 ändert sich die Lösung, wenn die Definition von Teiltouren streng ausgelegt wird. Das führt dazu, dass sich Start- und Endpunkt nicht ändern können:

tour4.txt

```
Routenlänge: 2000 (davor 3200)
Anzahl Tourpunkte: 21 (davor 29)
```

```
Blaues Pferd (1523)
Alte Mühle (1544)
Marktplatz (1549)
Marktplatz (1562)
Springbrunnen (1571)
Dom (1596) X
Bogenschütze (1610)
Bogenschütze (1683)
Schnecke (1698) X
Fischweiher (1710)
Reiterhof (1728) X
Schnecke (1742)
Schmiede (1765)
Große Gabel (1794)
Große Gabel (1874)
Fingerhut (1917) X
Stadion (1934)
Marktplatz (1962)
Baumschule (1974)
Polizeipräsidium (1991)
Blaues Pferd (2004)
```

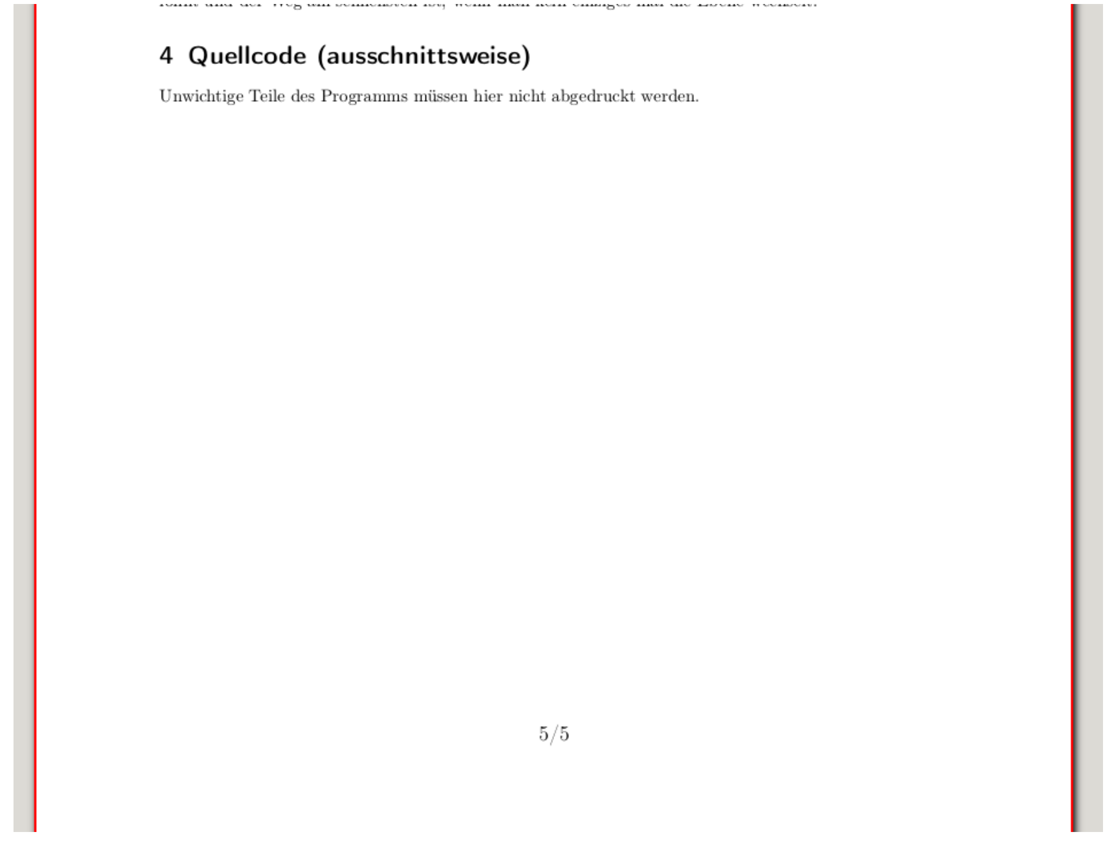
5.3 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
  Insbesondere soll deutlich werden, ob nur *geschlossene* Teiltouren im strengen Sinne weggelassen werden (womit Start- und Endpunkt der ursprünglichen Tour beibehalten werden müssen) oder ob und wie die Lösung darüber hinausgeht. Beides ist in Ordnung. Weitere Ideen zur Kürzung von Touren (z.B. wenn Wege andersherum durchlaufen werden) müssen gut begründet werden.
- $[-1]$ **Modellierung ungeeignet**
  Die Eingabe muss korrekt in Datenstrukturen übertragen werden. Dabei muss beachtet werden, dass (a) nur aufeinanderfolgende Tourpunkte direkt verbunden sind, (b) der Abstand der Tourpunkte kumuliert gegeben ist und (c) mehrfach auftretende Orte korrekt erfasst werden.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Alle essentiellen Tourpunkte müssen in der Tour vorhanden sein Dazu dürfen nur Teiltouren entfernt werden, die keine essentiellen Tourpunkte enthalten. Die gefundenen Touren müssen grundsätzlich korrekt sein: Sie müssen streng chronologisch aufgebaut sein und am gleichen Ort starten und enden.
- $[-1]$ **Tour nicht minimal**
  Die Tour muss möglichst kurz sein. Verfahren, welche die erste gefundene Teiltour entfernen, erzielen keine optimalen Ergebnisse. Bei überlappenden Teiltouren müssen die entfernt werden, welche die Tour am stärksten kürzen. Verfahren, die nur geschlossene Teiltouren weglassen, haben in manchen Fällen (etwa bei Beispiel 4) schlechtere Ergebnisse; das ist in Ordnung.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Ein Betrachten von allen möglichen Kombinationen von überlappenden Teiltouren, ist in Ordnung. Bei noch schlechteren Laufzeiten wird abgezogen.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Es soll ersichtlich werden, wie lange die neue Tour ist (nur die Anzahl der Tourpunkte genügt) und welche Tourpunkte (Ort und Jahr) in welcher Reihenfolge besucht werden. Wir akzeptieren auch, wenn Orte anstatt Tourpunkte aufgelistet werden und Tourpunkte mit gleichem Ort zusammengefasst werden.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 4 der vorgegebenen Beispiele 1 bis 5 enthalten, darunter `tour4.txt`, bei der klar wird, welcher „Auslegung“ des Begriffs der Teiltour die Lösung folgt.

Aus den Einsendungen: Perlen der Informatik

Allgemein



4 Quellcode (ausschnittsweise)

Unwichtige Teile des Programms müssen hier nicht abgedruckt werden.

5/5

Abbildung 5.1: Kommentar des Bewerters: Anscheinend war nichts wichtig.

Die Umsetzung war nicht ganz so leicht, da ständig irgendwelche Bugs aufgetreten sind.

Zur Umsetzung dieser Lösung wird die funktionale Programmiersprache Haskell verwendet. Dafür gibt es allerdings keinen vernünftigen Grund.

Die Implementation erwies sich als anstrengender als gedacht.

Man erkennt am Verlauf des Codes, dass ich beim Programmieren die Syntax gelernt habe.

Junioraufgabe 1: Wundertüte

Als zweiten Schritt wird bei der Tüte mit dem gierigsten Inhalt startend jeweils ein Spiel der Sorte der Tüte hinzugefügt und bei der nächsten Tüte weitergemacht.

Die Tüten werden grundsätzlich als ’Kisten’ bezeichnet.

Insgesamt stellt dieses Programm eine solide Grundlage für die effiziente und ausgewogene Zusammenstellung von Wundertüten dar und kann leicht an verschiedene Szenarien angepasst werden. Es verdeutlicht die Anwendung von Grundlagen der Mathematik und Programmierung in der Lösung von realen Problemen, wie sie in Informatikwettbewerben auftreten können.

Gemacht habe ich das in Blockly. Die Aufgabe war sehr schwer, viel schwerer als die anderen Runden. Das Ergebnis stimmt nicht ganz.

Variablennamen: `tmpp`, `tessa` und `fredi`

Eine für den Endanwender sicherlich sehr nützliche Verbesserung wäre das Hinzufügen einer grafischen Benutzeroberfläche. So könnten Nutzer mit einer gewissen Angst vor dem Terminal das Programm auch in vollen Zügen genießen. (Fußnote: Diese Doku wurde auf dem Fußboden eines vollen Zugs geschrieben.)

Abbildung 5.2: Ein Ausschnitt des Programms. Kommentar des Bewerters: Ich wusste nicht, dass Emojis als Variablenbezeichnung funktionieren!

Junioraufgabe 2: St. Egano

Die Umsetzung meines Programmes entsprach ziemlich genau meiner Lösungsidee, wobei ich durch einen kleinen Konzentrationsfehler den Bildern lange Zeit keine Botschaft entlocken konnte.

Ein Programm kann aber nicht wirklich so wie Menschen merken, ob es über den Bildrand hinausgeht oder nicht. Deshalb muss es dies errechnen.

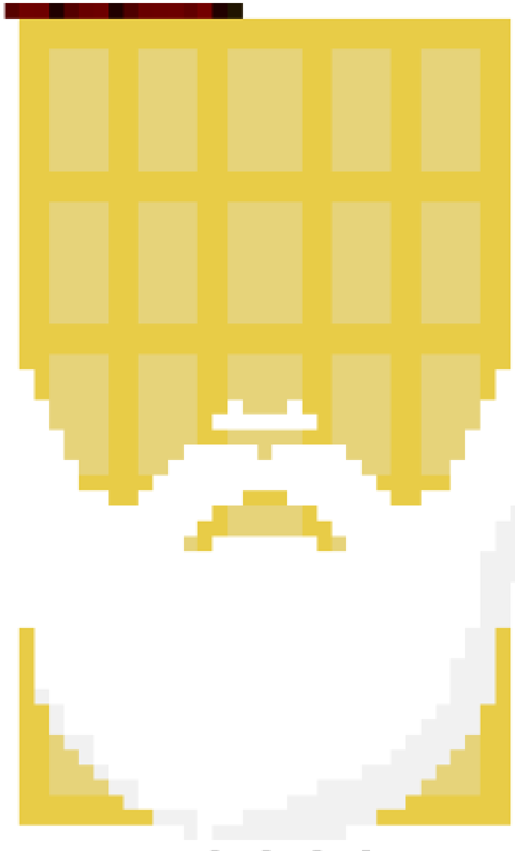
Die Aufgabe können wir uns als eine Art Schatzsuche vorstellen, in der die Pixel die einzelnen Stationen sind.

Variablendeklarationen im Quellcode:

```
Ausgabe = ""
AUSGABE = ""
AUsgabe = ""
AUSgabe = ""
AUSGabe = ""
AUSGAbe = ""
AUSGABe = ""
```

Dann, so war der Plan, fange ich oben links im Bild an, ...

das Mudolo (Kommentar des Bewerters: Eine neu entdeckte Tierart? Im Netz war nichts dazu zu finden ...)



weiseschokolade.png

Aufgabe 1: Arukone

Auszug aus dem Quellcode (Kommentar des Bewerters: Hier musste leider bei ’Rätsel nicht verschieden genug’ abgezogen werden...)

```
public void erstelleGitter(){
    if(n == 4){
        System.out.println("0001");
        System.out.println("0200");
        System.out.println("0000");
        System.out.println("2010");
    }else{
        if(n == 5){
            System.out.println("01000");
            System.out.println("10002");
            System.out.println("20000");
            System.out.println("00000");
            System.out.println("00000");
```

```
        }else{
            if(n == 6){
                System.out.println("010000");
                ...
            }else{
                if(n == 7){
                    System.out.println("0000020");
                    ...

                }else{
                if(n == 8){
                    System.out.println("01000004");
                    ...

                }
            }
        }
    }
```

Links die Ausgabe des Arukone-Checkers und rechts die Lösung des Programms (erstellt durch Manipulation des Quellcodes der Seite über die Entwicklerwerkzeuge).

Teampartner: ChatGPT. Zur Umsetzung dieser Idee habe ich ChatGPT verwendet, denn ich habe nicht genügend Programmierkenntnisse, um diese Ideen selbständig umzusetzen.

Anschließend wird auf die Lösbarkeit überprüft (hier scheiterten wir). Deshalb wird immer weiter ein zufälliges Gitter generiert, auf Lösbarkeit überprüft und falls es nicht lösbar ist, wird ein neues generiert. Da dies zu lange dauert, weil die Wahrscheinlichkeit, zufällig ein lösbares Rätsel zu generieren, zu gering ist, tut unser Programm nichts.

Wir rennen also einfach normalen Dijkstra darauf.

Aufgabe 2: Die goldene Mitte

Meine Erfahrung mit dieser Aufgabe ist ein purer Albtraum. Ich hatte mir mit den anderen Aufgaben zu viel Zeit gelassen und musste unter großem Zeitdruck arbeiten. Zusätzlich scheint die Aufgabe, soweit ich verstehe eine Unterart des *3D Bin Packing Problem*, kaum lösbar zu sein. Bei einer Internetrecherche stieß ich immer wieder auf die Aussage, dass solche Probleme *NP-hard* seien. Ich weiß nicht genau, was das ist, jedoch wurde auf den gleichen Seiten mit wahnsinnig viel Mathematik gearbeitet. Ich habe mich lange an einer effizienten Lösung versucht, doch ist im Endeffekt ein Versuch an *Brude Force*, der für Rätsel 5 jedoch keine Lösung generiert. Somit kann meine Lösung für die Aufgabe keineswegs mit dem Attribut 'gut' betitelt werden, beschäftigt sich aber dann doch in ihrem Irrweg mit einem faszinierenden mathematischen Problem, der Errechnung aller möglichen Permutationen eines *Multisets*. Für mich als Lernender ist die Bearbeitung der Aufgabe somit insgesamt doch sehr lohnenswert gewesen, auch wenn ich mit dem Lösungskonzept sehr unzufrieden bin. Außerdem habe ich hoffentlich aus meinem schlechten Zeitmanagement Erfahrungen für die Zukunft gesammelt.

Es wird zum Lesen dieses Dokuments Dark-Mode empfohlen.

Aufgabe 2: Die goldene Mitte

3 Beispiele

3.1 Beispiel 1

Das Programm behauptet zwar, hierfür eine Lösung zu finden, diese ergibt aber gar keinen Sinn. Das ist die Ausgabe:
Folgende Lösung wurde gefunden:
Ebene 0
E E E
E E E
E E E

Ebene 1
E E E
E E E
E E E

Ebene 2
E E E
E E E
E E E

Alle Beispiele haben keine Lösung, zumindest sagt das mein Programm, ob das wirklich so ist, bin ich mir nicht ganz sicher.

Das Programm ist in der Lage, beliebig viele Quader in eine würfelförmige Kiste einer beliebigen Kantenlänge zu puzzeln, [...]. Bending spacetime for fun and profit.

Weil ich dachte, dass Listen eventuell ungeeignet sind, da es lange dauert, sie auszulesen und zu speichern, hatte ich zudem die Idee, den Würfel als Liste von Platzierungen zu speichern.

Im Code: // TODO hier Quelltext einfügen"

Außerdem haben wir [...] den Brute-Force-Ansatz verwendet, da wir diesen aktuell im Unterricht erlernt haben.

Dass field ein Array ist und auf field[x] zugegriffen wird, liegt daran, dass field ein Array ist [...]

Aufgabe 3: Zauberschule

Bugwarts ist ein Graf.

Um Ron schnellstmöglich durch die Zauberschule zu lotsen, dachte ich an einen A*-Algorithmus in einer dreidimensionalen Karte. Außerdem sollen Start- und Endpunkt deutlich definiert sein. Dann beginnt auch schon die Magie.

Immer wenn ich 'schnellsten Weg' höre, muss ich sofort an meinen liebsten Wegfindungsalgorithmus denken: und zwar Dijkstra.

Ist beim Anfang eines Durchgangs die derzeitige z-Position größer als 1, aber nicht gleich 6 oder 7, dann steckt man noch in der Decke und wird der Queue wieder angehangen.

Der Parameter "private static String findShortestPath()"nimmt zwei Labyrinthe und Start-/Endpositionen an.

```
zauberschule1.txt
#####################
#...#.....#...#.....#
#.#.#.###.#.#.#.###.#
#.#.#...#.#.#...#...#
###.###.#.#.#####.###
#.#.#...#.#B....#...#
#.#.#.###.#^###.#####
#.#...#.#.#^<<#.....#
#.#####.#.#########.#
#...................#
#####################
Ich bin mir sicher, Ron hätte den Weg auch so
gefunden …
#####################
#.......#.....#.....#
#.###.#.#.###.#.###.#
#.....#.#.#.....#.#.#
#######.#.#######.#.#
#.....#.#.....#...#.#
#.###.#.#.###.###.#.#
#.#.#...#.#...#...#.#
#.#.#######.###.###.#
#...........#.......#
#####################
```

Diese Idee musste auch wieder verworfen werden, da der RAM zu begrenzt wäre, um rekursiv wirklich ALLE Pfade zu finden.

Eine möglichst simple und doch effiziente Implementierung mit einer verschachtelten while() Schleife.

Damit der Code weiß, wo er 'loslaufen' muss und wo er enden soll, muss das Labyrinth einmal durchlaufen werden.

Aus irgendeinem Grund habe ich nur ein eindimensionales Array verwendet.

```
step :: ([((Int, Int, Int), (Int, Int, Int))], [((Int, Int, Int),
[((Int, Int, Int), Int)])]) ->
([((Int, Int,Int), (Int, Int, Int))], [((Int, Int, Int),
[((Int, Int, Int), Int)])])
```

Aufgabe 4: Nandu

Ich habe mich hier dafür entschieden, dass das Licht den Sensor des zweiten Steines erreicht, da LEDs heutzutage eine hohe Helligkeit bei wenig Energieverbrauch haben und die Lichtsensoren normalerweise sehr genau und fein reagieren können.

In Beispiel 3 wird das Programm auf einmal rebellisch und entscheidet sich, nur noch die Durchläufe mit ungerader Ordnungszahl auszugeben. Im 4. Beispiel scheint es sich noch einmal einzukriegen und sinnvollen Output zu generieren. Doch in Beispiel 5 quittiert es schließlich endgültig den Dienst und gibt nun gar keine Ausgabe-Signale mehr aus. Mir ist klar, dass sich hinter all diesem ungewollten Verhalten des Programms irgendwelche abstrusen Bugs verstecken müssen, jedoch bin ich müde und habe schon etwa 2 Dutzend Bugs in den letzten drei Stunden behoben, weshalb ich es dabei belassen werde und stattdessen zu Bett gehe.

Diese Aufgabe hat mich wirklich nochmal in die 8. Klasse zurückversetzt, als Boolesche Algebra unser Stoffgebiet war. Umso mehr habe ich mich gefreut, diese Aufgabe zu lösen und einen Nandu-Builder zu programmieren, der eventuell sogar verwendet werden könnte.

Die zweite Variante der Aufgabe wurde ebenfalls in Java programmiert, wird aber mithilfe eines selbst programmierten Plugins in Minecraft visuell dargestellt.

Die Vorgaben der vierten Aufgabe des 42. Bundeswettbewerbs Informatik konnten leider nicht eingehalten werden. Dies ist sowohl meiner mangelnden Erfahrung in der Programmierung geschuldet als auch der mangelhaften Vorbereitung seitens des Ausbildungsbetriebs und der Berufsschule.

Minecraft eignet sich hervorragend für die Visualisierung von Logik-Gattern, da es Turing Complete ist.

Die Datei hat eine klare Struktur. Die erste Zeile müssen die Dimensionen sein, dennoch müssen sie nicht stimmen, da sie im Code nachher selbst herausgefunden werden.

Aufgabe 5: Stadtführung

Bei einer solchen Auswahl wäre die rein logisch korrekte Tour gerade so einen Stopp lang, also ungefähr so lange, wie die Aufmerksamkeitsspanne heutzutage.

Um zu beginnen, werden wir das Array berechnen, dann werden wir unnötige Knoten entfernen, es sortieren und die Antwort berechnen. Da wir sortieren, erhalten wir eine Asymptotik von $\mathcal{O}(n\log(n))$. [Ende der Einsendung]

Naiver Ansatz: alle möglichen Kombinationen (2^n) ausprobieren. Guter Ansatz: Problem lässt sich als weighted activity selection erkennen. Google nach der Lösung fragen. Keinen blassen Schimmer haben. Die Lösung ist zu schwer. Ein Link zu StackOverflow muss her. (Kommentar: Es folgt tatsächlich ein dubioser Link)