38. Bundeswettbewerb Informatik 2019/2020, 1. Runde

Lösungshinweise und Bewertungskriterien

Bundeswettbewerb Informatik

Allgemeines

Es war sehr erfreulich, dass sich wieder besonders viele die Zeit zur Bearbeitung der Aufgaben genommen und am Bundeswettbewerb Informatik (BwInf) teilgenommen haben. Den Veranstaltern ist bewusst, dass in der Regel viel Arbeit hinter der Erstellung einer Einsendung steckt.

Die BewerterInnen würdigten die Leistungen der TeilnehmerInnen so gut wie möglich. Dies wurde ihnen nicht immer leicht gemacht, insbesondere wenn die Dokumentation nicht die im Aufgabenblatt genannten Anforderungen erfüllte. Bevor mögliche Lösungsideen zu den Aufgaben und Einzelheiten zur Bewertung beschrieben werden, soll deshalb im folgenden Abschnitt auf die Dokumentation näher eingegangen werden; vielleicht helfen diese Anmerkungen für die nächste Teilnahme.

Wie auch immer die Einsendung bewertet wurde, es 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. Selbst wenn man zum Beispiel aus Zeitmangel nur die Lösung zu einer Aufgabe einreichte, so erhält man eine Bewertung der Einsendung, die für die Zukunft hilfreich sein kann.

Die Bearbeitungszeit für die 1. Runde beträgt etwa drei Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der Dokumentation der Aufgabenlösungen zu vermeiden. Einige Aufgaben sind oft schwerer zu bearbeiten, als sie auf den ersten Blick erscheinen mögen. Erst in der konkreten Umsetzung einer Lösungsidee und Testen von Beispielen stößt man manchmal auf weitere Besonderheiten bzw. noch zu lösende Schwierigkeiten, was dann zusätzlicher Zeit bedarf.

Noch etwas Organisatorisches: Sollte der Name auf Urkunde oder Teilnahmebescheinigung falsch geschrieben sein, ist er auch im PMS falsch eingetragen. Die Teilnahmeunterlagen können gerne neu angefordert werden; dann aber bitte vorher den Eintrag im PMS 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* besteht. 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 meisten Inhalte kaum verständlich sind 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 zu einem Punktabzug führen, weil das „Verfahren unzureichend begründet bzw. schlecht nachvollziehbar“ ist.

Ganz besonders wichtig sind die vorgegebenen (und ggf. weitere) *Beispiele*. Wenn Beispiele und die zugehörigen Ergebnisse in der Dokumentation fehlen, führt das zu Punktabzug. Zur Bewertung ist für jede Aufgabe vorgegeben, zu wie vielen (und teils auch zu welchen) Beispielen korrekte Programmausgaben/-ergebnisse in der Dokumentation erwartet werden. Die Ergebnisse, die für die vorgegebenen Beispiele in der Dokumentation angegeben werden, sollten alle korrekt sein. Punktabzug für falsche Ergebnisse gibt es aber nur, wenn für den ursächlichen Mangel kein Punkt abgezogen wurde.

Es ist nicht ausreichend, Beispiele nur in gesonderten Dateien abzugeben, ins Programm einzubauen oder den Bewertern sogar das Erfinden und Testen von geeigneten Beispielen zu überlassen. Beispiele sollen die Korrektheit der Lösung belegen, und es sollen damit auch Sonderfälle gezeigt werden, die das Programm entsprechend behandeln kann.

Auch *Quellcode*, zumindest die für die Lösung wichtigen Teile des Quellcodes, 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, mit allen notwendigen Eingabedaten/-dateien (z. B. für die Beispiele). 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 ausreichend 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 des BwInf qualifiziert hat, beachte bitte, dass dort deutlich kritischer als in der 1. Runde bewertet wird und höhere Anforderungen an die Qualität der Dokumentation 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), genug 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 PMS einsehen kann, aufgelistet und geben an, inwieweit die Bearbeitung einer Aufgabe die einzelnen Bewertungskriterien erfüllt. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation, die oben erklärt sind.

In der 1. Runde des BwInf 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 oder gelegentlich auch zwei Punkte Abzug; ansonsten ist die Bearbeitung in Bezug auf dieses Kriterium in Ordnung. Wurde die Aufgabe nur unzureichend bearbeitet, wird ein gesondertes Kriterium angewandt, bei dem es bis zu fünf Punkten Abzug gibt. Im schlechtesten Fall wird eine Aufgabenbearbeitung mit 0 Punkten bewertet.

Für die Gesamtpunktzahl sind die drei am besten bewerteten Aufgaben 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 des BwInf qualifiziert. Kritische Fälle mit nur 11 Punkten sind bereits sehr gründlich und mit viel Wohlwollen geprüft. Leider ließ sich aber nicht verhindern, dass 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 BwInf-Einsendung nicht beachtet, vgl. 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 BwInf gewertet werden.

Danksagung

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

An der Erstellung der im folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Dominik Meier (Junioraufgabe 1), Jannik Kudla (Junioraufgabe 2), Michael Rosenthal (Aufgabe 1), Christian Hagemeier (Aufgaben 2 und 3), Niels Mündler (Aufgabe 4) und Julian Baldus (Aufgabe 5). Allen Beteiligten sei für ihre Mitarbeit hiermit ganz herzlich gedankt.

Junioraufgabe 1: Parallelen

J1.1 Lösungsidee

Zunächst stellt sich die Frage, welche Antwort man erwarten sollte. Hat Martin recht? Manuelles überprüfen beim Gedicht zeigt: Er hat recht. Bei allen Wörtern der ersten Hälfte kommt man bei dem Wort „verschlang“ heraus. Erst bei dem zweiten Wort in der zweiten Hälfte kommt man zum ersten Mal zu einem anderen Wort, nämlich „Ewigkeit“.

Dies liegt jedoch nicht an der speziellen Struktur des Gedichts und dem Genie des Autors, sondern ist eine einfache Konsequenz daraus, dass bei hinreichender Textlänge es sehr wahrscheinlich ist, dass die „Pfade“, die von zwei verschiedenen Wörtern ausgehen, sich irgendwann vereinen. Mit „Pfad“ ist an dieser Stelle die Abfolge von Wörtern gemeint, zu denen man bei der Anwendung des Verfahrens springt. Die gleiche Überlegung kann auch für die Konstruktion eines Kartentricks genutzt werden[1], hier sind auch mathematische Überlegungen zu Wahrscheinlichkeiten von Kollisionen gegeben.

Als Beispiel hier die ersten beiden Strophen des Gedichts. Alle Wörter, die man erreicht, wenn man mit dem ersten Wort „Es“ beginnt, sind rot markiert.

> Es gingen zwei Parallelen
> ins Endlose hinaus,
> zwei kerzengerade Seelen
> und aus solidem Haus.
>
> Sie wollten sich nicht schneiden
> bis an ihr seliges Grab:
> Das war nun einmal der beiden
> geheimer Stolz und Stab.
> ...

Nun noch einmal die ersten beiden Strophen. Diesmal wurde die Methode aber für die nächsten drei Wörter angewandt. Während es in der ersten Strophe noch Unterschiede bei den markierten Wörtern gibt, sind sie in der zweiten am Ende bereits alle identisch.

> Es gingen zwei Parallelen
> ins Endlose hinaus,
> zwei kerzengerade Seelen
> und aus solidem Haus.
>
> Sie wollten sich nicht schneiden
> bis an ihr seliges Grab:
> Das war nun einmal der beiden
> geheimer Stolz und Stab.
> ...

[1] Jeffrey C. Lagarias, Eric Rains, Robert J. Vanderbei: The Kruskal Count, 2001, https://arxiv.org/abs/math/0110143

Einfache Lösung

Die einfache Lösung des Problems ist, das beschriebene Verfahren für jedes Wort in der ersten Hälfte des Gedichtes durchzuführen und dabei zu prüfen, ob man jeweils auf das gleiche Ergebnis kommt wie das Resultat beim ersten Wort.

Dabei muss insbesondere beachtet werden, dass nach der Aufgabenstellung nur die Buchstaben gezählt werden sollen. Die Satzzeichen müssen also vorher herausgefiltert werden oder anderweitig bei der Zählung der Zeichen ignoriert werden.

Die einfache Lösung hat im schlechtesten Fall quadratische Laufzeit abhängig von der Anzahl der Wörter im Gedicht.

Lineare Lösung

Eine Verbesserung kann erzielt werden indem man realisiert, dass das Verfahren abgebrochen werden kann, sollte man auf ein Wort stoßen, dass Teil eines Pfads eines vorigen Wortes ist. In diesem Falle weiß man bereits, dass das momentan untersuchte Wort auf dem gleichen Wort enden wird wie die vorigen.

Diese Erkenntnis kann auf unterschiedliche Weise genutzt werden. Eine Möglichkeit ist beispielsweise, die Wörter während des Verfahrens zu markieren. Dazu bietet sich ein Array an, das für jedes Wort speichert, ob sein Pfad zum gleichen Wort führt wie der Pfad des ersten Wortes. Wenn das Verfahren dann für das erste Wort ausgeführt wird, werden alle besuchten Wörter markiert. Bei allen folgenden Wörtern wird auch so verfahren, nur dass gestoppt wird, wenn auf ein bereits markiertes Wort gestoßen wird. Sollte man bei einem der folgenden Wörter das Ende des Gedichts erreichen, ohne auf ein markiertes Wort gestoßen zu sein, hat man das erste Wort gefunden, für das die Behauptung nicht gilt.

Dieses Verfahren sorgt dafür, dass jedes Wort quasi nur einmal besucht wird und nicht für jedes Wort das gesamte Verfahren ausgeführt werden muss. Zu überprüfen, ob ein Wort bereits markiert ist, erfordert einen einfachen Array-Speicherzugriff und geht sehr schnell.

J1.2 Alternative Lösungsideen

Das gleiche Verfahren kann auch von hinten nach vorne angewendet werden, man erhält somit dann etwas eher ein paar Wörter, deren Pfad auf verschiedenen Wörtern endet. Vor allem aber erstmal langweilige, wie dem letzten und vorletzten Wort. Daher sollte hier das Verfahren bei einem Gegenbeispiel nicht direkt abgebrochen werden.

Auch denkbar ist aus dem Gedicht einen Graphen explizit zu konstruieren und dann zu prüfen, ob der Graph zusammenhängend ist.

J1.3 Beispiele

Bei dem gegebenen Gedicht funktioniert das Verfahren für alle Wörter bis zum Wort „wie“ in der vorletzten Strophe. Ausgehend von „wie“ landet man beim letzten Wort des Gedichts „Seraphim“ , während alle vorigen Wörter zu „verschlang“ in dem vorletzten Vers der letzten Strophe führen.

Algorithmus 1 Lösung

```
for i in [0, Anzahl Worte) do
    markiert(i) ← false
end for
for i in [0, Anzahl Worte) do
    k ← i
    while k < Anzahl Worte do
        if markiert(k) then
            break
        else
            markiert(k) ← true
        end if
        k ← = k + wortLaenge(gedicht[k])
        if i > 0 und k ≥ Anzahl Worte then
            gib aus "Gegenbeispiel gefunden"
        end if
    end while
end for
```

Ein gutes Beispiel zum Testen des Programmes könnte der folgende oder ein ähnlicher String sein: „`aa bb aa bb aa bb aa bb aa bb`“. Bei diesem ist das Gegenbeispiel bereits das zweite „Wort“ (`aa` → `aa` → `aa` → `aa` → `aa`, `bb` → `bb` → `bb` → `bb` → `bb`).

Auch möglich ist das Testen auf anderen Gedichten oder gar Romanen; eine gute Quelle hierfür stellt Projekt Gutenberg dar. Neben den obigen Überlegungen würden auch solche Tests zeigen, dass Christian Morgenstern nicht allein steht mit Texten, für die das Verfahren funktioniert.

J1.4 Bewertungskriterien

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

- [−1] **Lösungsverfahren fehlerhaft**
  Zur Bestimmung der Sprunglänge sollen nur Buchstaben gezählt werden, keine Satzzeichen. Der Gedankenstrich im Text ist kein eigenes Wort.
- [−1] **Implementierung fehlerhaft**
  Die Implementierung sollte die beschriebene Lösung korrekt umsetzen. Wenn im Programm z. B. immer n Wörter übersprungen werden (also immer $n+1$ Wörter weitergegangen wird), statt n Wörter weiter zu gehen, ist das nur in Ordnung, wenn das auch in der Dokumentation so beschrieben ist.
- [−1] **Ausgabe schlecht nachvollziehbar**
  Die Ausgabe des Programms sollte erkennbar machen, dass die von Wörtern der ersten Hälfte des Gedichtes ausgehenden Pfade alle bei dem Wort „verschlang“ enden. Außerdem soll nachvollziehbar sein, wie von einem einzelnen Wort das End-Wort erreicht wird.
- [−1] **Ergebnis unzureichend begründet**
  Das (positive) Ergebnis für das Gedicht „Die zwei Parallelen“ muss angegeben werden. Außerdem soll es durch die Angabe einer hinreichenden Zahl von „Pfaden“, die alle zu demselben Wort führen, begründet werden. Es ist nicht nötig, für alle Wörter aus der ersten Hälfte des Gedichts die Pfade anzugeben.

Junioraufgabe 2: Kacheln

J2.1 Lösungsidee

Man stellt fest, dass immer vier Teilquadrate von vier im Quadrat zusammenliegenden Kacheln entweder alle aus Land oder alle aus Wasser bestehen müssen. Ansonsten würden zwei Kacheln in einer nicht erlaubten Weise aneinander liegen.

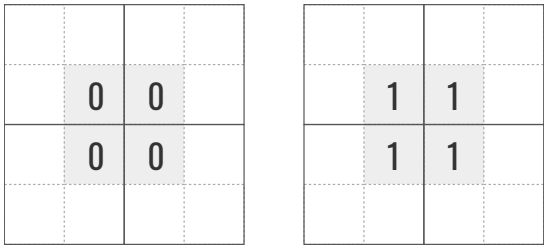


Abbildung J2.1: Mögliche Belegungen für die „inneren“ Teilquadrate von vier aneinander liegenden Kacheln

Wie oft, hilft auch hier zunächst ein Blick in die Beispieldaten. Da in den Eingabedaten für jedes Teilquadrat der Kacheln angegeben ist, ob es Land, Wasser oder noch unbestimmt ist, kann direkt für jede zusammengehörende Gruppe von Teilquadraten geprüft werden, ob sie

a) nur Land (und ggf. unbestimmte Teilquadrate) oder nur Wasser (und ggf. unbestimmte Teilquadrate) enthalten,

b) nur unbestimmte Teilquadrate enthalten, oder

c) mindestens einmal Land und einmal Wasser enthalten.

Im letzten Fall lässt sich die Landschaft nicht vervollständigen, ohne am Ende eine nichterlaubte Anordnung der Kacheln zu erhalten. Selbst wenn das Land und das Wasser nicht direkt nebeneinander liegen, lassen sich die anderen Felder nicht mehr belegen, ohne eine nichterlaubte Anordnung zur erzeugen. In diesem Fall kann das Programm also sofort abgebrochen werden.

Im ersten Fall kann die Belegung der ggf. vorhandenen noch unbestimmten Teilquadrate einfach entsprechend der schon belegten Teilquadrate festgelegt werden. Da alle Möglichkeiten, Kacheln aus Teilquadraten zu erzeugen, als vorgegebene Kacheln vorhanden sind, gibt es keine Einschränkung bei der Erstellung der Kacheln aus den Teilquadraten.

Nur im Fall b) muss man kurz überlegen, was man tun kann. Auch hier stellt man wieder fest: Solange alle vier Teilkacheln in der selben Art belegt werden, können keine Einschränkungen verletzt werden. Ob man alle vier Teilquadrate mit Wasser belegt oder alle mit Land, spielt also keine Rolle.

Am Ende muss noch berücksichtigt werden, dass es am Rand der Landschaft auch zu Fällen kommt, bei denen nur jeweils zwei Teilquadrate auf Gleichheit geprüft werden müssen. Bei den vier Teilquadraten in den Ecken der Landschaft ist jede beliebige Belegung möglich.

J2.2 Lösung mit Graph

Vorbemerkung: Diese Lösungsidee verwendet Methoden, die weit über die Anforderungen einer Junioraufgabe hinausgehen.

Unsere Karten bestehen aus Kacheln, die wiederum aus 2×2 kleineren Teilkacheln besetehen. Wir beschreiben jede Teilkachel durch ein Koordinatenpaar (x,y). Ist die Karte w Kacheln breit und h Kacheln hoch, so bezeichnet $(0,0)$ die Teilkachel oben links und $(2 \cdot w-1, 2 \cdot h-1)$ die Teilkachel unten rechts. Insgesamt gibt es somit $4 \cdot w \cdot h$ Teilkacheln. Analog zu den Teilkacheln können wir den Kacheln Koordinaten von $(0,0)$ bis $(w-1, h-1)$ zuordnen. Beschreibt (x,y) eine Teilkachel, so bezeichne $\mu(x,y) := \left(\left\lfloor \frac{x}{2} \right\rfloor, \left\lfloor \frac{y}{2} \right\rfloor\right)$ die zugehörige Kachel.

Unser Ziel besteht darin, jeder Teilkachel ein Bit (0 = überwiegend Wasser, 1 = überwiegend Land) zuzuordnen, sodass zwei *benachbarte* Teilkacheln *verschiedener* Kacheln das gleiche Bit erhalten. Hierbei sind zwei Teilkacheln (x,y) und (x',y') benachbart, wenn sie eine gemeinsame Seite haben, d. h. wenn $|x-x'|+|y-y'|=1$ gilt. So stellen wir sicher, dass stets zwei Kacheln in ihrer angrenzenden Seite übereinstimmen.

Bevor wir jeder Teilkachel ein Bit zuordnen, konstruieren wir einen ungerichteten Graphen $G=(V,E)$ mit $V = \{0,1,\ldots,2 \cdot w-1\} \times \{0,1,\ldots,2 \cdot h-1\}$ als Knotenmenge. Es gibt also einen Knoten pro Teilkachel. Zwei Knoten $(x,y),(x',y') \in V$ verbinden wir genau dann durch eine Kante, wenn (x,y) und (x',y') benachbart sind und zu verschiedenen Kacheln gehören. Formal: $\{(x,y),(x',y')\} \in E \iff |x-x'|+|y-y'|=1$ und $\mu(x,y) \neq \mu(x',y')$. Schließlich färben wir die Knoten wie folgt:

- Knoten mit vorgegebenem Bit 0 bzw. überwiegend Wasser werden **blau** gefärbt.
- Knoten mit vorgegebenem Bit 1 bzw. überwiegend Land werden **rot** gefärbt.
- Knoten, deren Bit nicht vorgegeben ist, werden gar nicht gefärbt und bleiben schwarz.

Abbildung J2.2 veranschaulicht unsere Konstruktion.

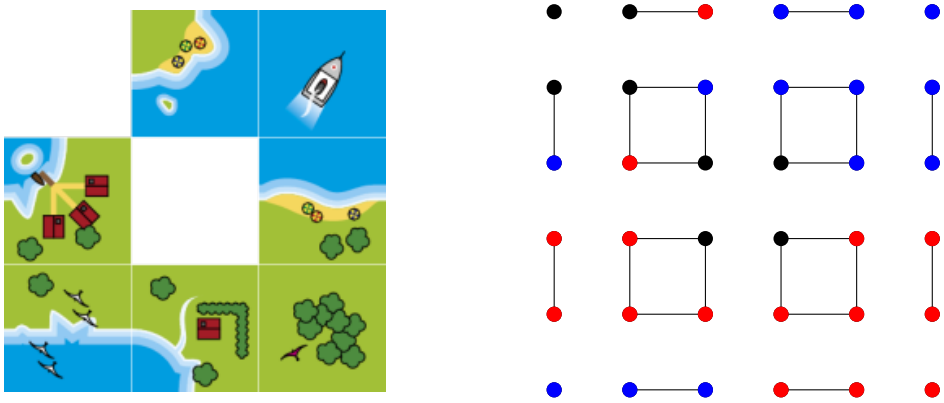


Abbildung J2.2: Konstruktion des Graphen G aus einer Karte der Größe $w = h = 3$

Der Graph besteht aus drei Typen von Zusammenhangskomponenten: Isolierte Knoten, durch eine Kante verbundene Knotenpaare und Zyklen der Länge 4. Allgemein enthält er $4 \cdot w \cdot h$ Knoten und $4 \cdot w \cdot h - 2 \cdot w - 2 \cdot h$ Kanten.

Beobachtung: Allen Knoten einer Zusammenhangskomponente müssen wir die gleiche Farbe zuordnen. Ist wie in Abbildung J2.2 ein blauer und ein roter Knoten in derselben Zusammenhangskomponente, so existiert für das Beispiel keine Lösung.

In allen anderen Fällen gibt es eine Lösung, die sich folgendermaßen bestimmen lässt.

Algorithmus

Es ist zwar möglich, aber keineswegs erforderlich, G explizit zu konstruieren und mit einem allgemeinen Algorithmus (z. B. Tiefensuche) die Zusammenhangskomponenten zu berechnen.

Algorithmus 2 Lösung mit Graph

1. Konstruiere den Graphen G inklusive Färbung der Knoten.
2. Für jede Zusammenhangskomponente Z von G...
   a) ... überprüfe, ob zwei Knoten in Z die gleiche Farbe haben und breche ggf. ab, denn es ist keine Lösung möglich.
   b) ... überprüfe, ob Z nur schwarze Knoten enthält und färbe ggf. einen beliebigen Knoten blau (oder rot).
   c) ... suche in Z einen bereits gefärbten Knoten und färbe alle Knoten in Z mit dieser Farbe.
3. Übertrage die Färbung des Graphen G in eine Landkarte und gebe diese im vorgegebenen Ausgabeformat aus.

Stattdessen können wir die in Abbildung J2.2 ersichtliche spezielle Struktur des Graphen ausnutzen und drei Fälle betrachten: Eckknoten, Randknoten (jene Knoten, die mit genau einem anderen Knoten verbunden sind) und innere Knoten, die in 2×2-Quadraten auftreten. Die vier Eckknoten können wir beliebig einfärben, sofern diese schwarz sind. Die Fälle der Randknoten und inneren Knoten lassen sich mit einigen Schleifen schnell erledigen.

Laufzeit: Die ≤ 4 Knoten einer Zusammenhangskomponente lassen sich in Zeit $\mathcal{O}(1)$ färben. Da wir jede Zusammenhangskomponente genau einmal betrachten und es höchstens so viele Zusammenhangskomponenten wie Knoten gibt, liegt die Laufzeit insgesamt in $\mathcal{O}(w \cdot h)$, ist also linear in der Anzahl der Knoten.

J2.3 Weitere Lösungswege

Es sei eine Karte der Breite w und Höhe h gegeben, wobei die Funktion

$$\tau : \{0,1,\ldots,w-1\} \times \{0,1,\ldots,h-1\} \to \{\bot,0,1,\ldots,15\} \tag{J2.1}$$

den Kacheln (x,y) ihren Typ $\tau(x,y)$ zuordnet. Hierbei markiert $\bot$ nicht vorgegebene Kacheln und die Zahlen 0 bis 15 bezeichnen die Kacheltypen von 0 = nur Wasser bis 15 = nur Land. Die Gestalt der Teilkacheln ist der Binärdarstellung dieser Nummer wie folgt zu entnehmen: Ist $\tau(x,y) = t_3 \cdot 2^3 + t_2 \cdot 2^2 + t_1 \cdot 2 + t_0$ mit $t_0,t_1,t_2,t_3 \in \{0,1\}$, dann hat Teilkachel $(2 \cdot x, 2 \cdot y)$ den Typ t_0, Teilkachel $(2 \cdot x+1, 2 \cdot y)$ den Typ t_1, Teilkachel $(2 \cdot x, 2 \cdot y+1)$ den Typ t_2 und Teilkachel $(2 \cdot x+1, 2 \cdot y+1)$ den Typ t_3. Das niederwertigste Bit entspricht also der Teilkachel oben links und das höchstwertige Bit der Teilkachel unten rechts.

Brute-Force-Lösung

Eine Brute-Force-Lösung wählt nach und nach für jede unbekannte Kachel einen passenden Typen aus und bricht ab, wenn alle Kacheln gesetzt sind oder es für eine Kachel keinen Typ mehr gibt. Im letzteren Fall wird der gesamte Algorithmus neu ausgeführt – in der Hoffnung, dass eine Lösung gefunden wird. Ist nach einer großen Zahl von C Iterationen (z. B. $C = 20\,000$) keine Lösung gefunden, gibt der Algorithmus aus, dass keine Lösung existiert.

Algorithmus 3 Brute-Force-Lösung

1. Initialisiere die Karte und setze $k \leftarrow 0$.
2. Wenn $k \geq C$, brich ab und gebe „Nein“ aus.
3. Wenn es keine Kachel (x,y) mit $\tau(x,y) = \perp$ gibt, brich ab und gebe „Ja“ aus.
4. Wähle eine beliebige Kachel (x,y) mit $\tau(x,y) = \perp$ aus.
5. Ermittle für alle $i,j \in \{0,1\}$ alle möglichen Typen der Teilkachel $(2 \cdot x + i, 2 \cdot y + j)$ und wähle – sofern beides möglich ist – zufällig Land oder Wasser aus. Falls es für eine Teilkachel keinen Typen gibt, setze $k \leftarrow k+1$ und gehe zu Schritt 2.
6. Weise der Kachel den ermittelten Typ zu.
7. Gehe zu Schritt 3.

Diese Lösung hat den großen Nachteil, nicht korrekt zu sein. Es gibt einen einseitigen Fehler: Wenn keine Lösung existiert, wird dies in jedem Fall richtig erkannt. Wenn es hingegen eine Lösung gibt, so besteht die Gefahr, dass der Algorithmus trotzdem „Nein“ ausgibt.

Rekursives Backtracking

Eine einfache und auf den BwInf-Beispielen gut funktionierende Lösung ist mit Backtracking möglich. Dazu iterieren wir über alle Kacheln und überspringen solche mit bereits vorgegebenem Typ. Hat eine Kachel den Typ $\perp$, so prüfen wir für jeden Typ, ob wir ihn der Kachel zuordnen können. Das ist genau dann der Fall, wenn die Kachel mit den benachbarten Kacheln in ihrer angrenzenden Seite übereinstimmt (d. h. die zwei Teilkacheln sind jeweils vom gleichen Typ). In diesem Fall ordnen wir der Kachel den Typ zu und fahren mit der nächsten Kachel fort. Sobald wir zum Ende kommen, ist eine Lösung gefunden.

Es kann sein, dass wir einer Kachel keinen der 16 Typen zuordnen können. Dann machen wir solange Schritte rückgängig, bis wir einer Kachel einen anderen Typen zuordnen können und setzen die Suche nach einer Lösung an dieser Stelle fort. Ist auch das nicht möglich, so haben wir alle Möglichkeiten ausprobiert und können die Existenz einer Lösung ausschließen.

Algorithmus 4 beschreibt das Vorgehen formal. Hierbei kann die mit $(\star)$ gekennzeichnete Zeile mittels einer Fallunterscheidung umgesetzt werden. Die Funktionen α und β sind durch

$$\alpha(x,y) = \begin{cases} x+1 & \text{falls } x < w-1 \\ 0 & \text{falls } x = w-1 \end{cases} \qquad \beta(x,y) = \begin{cases} y & \text{falls } x < w-1 \\ y+1 & \text{falls } x = w-1 \end{cases} \tag{J2.2}$$

definiert, sodass $(\alpha(x,y), \beta(x,y))$ nach (x,y) die nächste Kachel beschreibt.

Ein Nachteil dieser Lösung ist ihre im schlechtesten Fall exponentielle Laufzeit.

J2.4 Beispiele

Die folgenden Abbildungen zeigen jeweils zwei Karten. Die linke (unvollständige) Karte ist die Eingabe, die rechte Karte eine mögliche Lösung. Hervorzuheben ist das unlösbare Beispiel in

Algorithmus 4 Lösung mit Backtracking

```
procedure BACKTRACKING(x, y)
    if y = h then
        return 1                                              ▷ Lösung gefunden
    end if
    if τ(x,y) = ⊥ then
        for t = 0 to 15 do
            if Kachel (x,y) kann Typ t zugeordnet werden then      ▷ (⋆)
                τ(x,y) ← t
                if BACKTRACKING(α(x,y), β(x,y)) = 1 then
                    return 1
                end if
            end if
        end for
        return 0                                              ▷ keine Lösung
    else
        return BACKTRACKING(α(x,y), β(x,y))
    end if
end procedure
```

Abbildung J2.8: Hier ist es z. B. nicht möglich, die Kachel $(3,1)$ festzulegen, denn ihre linke untere Teilkachel stimmt weder mit Wasser noch mit Land mit den angrenzenden Teilkacheln überein. Somit existiert keine Lösung.

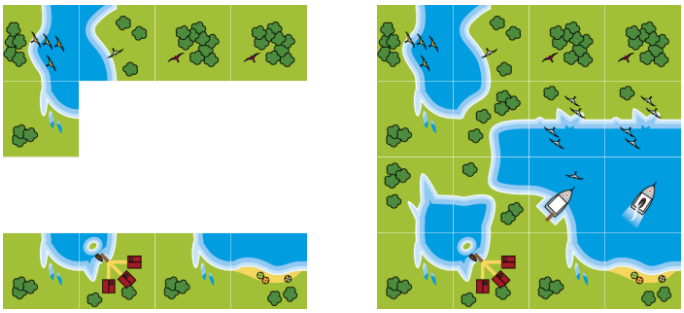


Abbildung J2.3: Eigenes Beispiel; 4×4 Landschaft

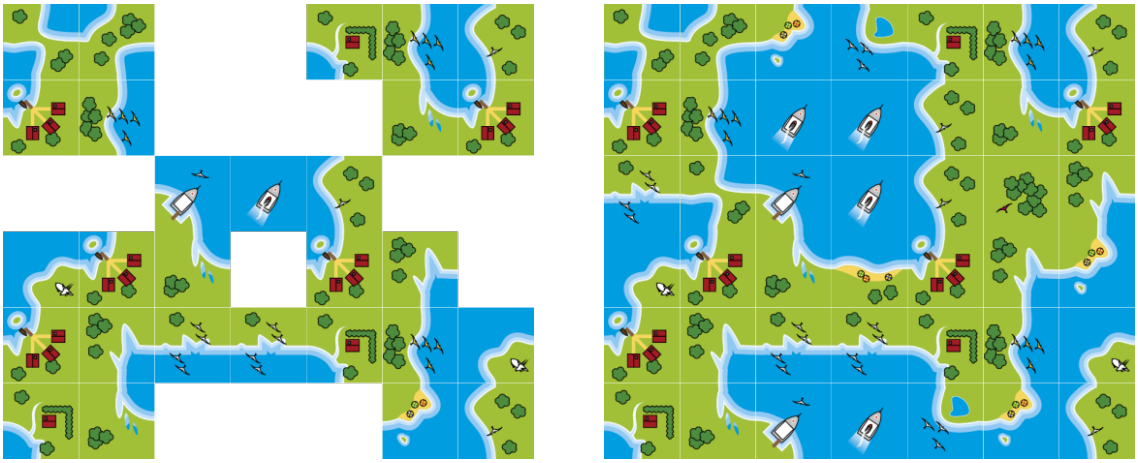


Abbildung J2.4: Beispiel 1; 6×7 Landschaft

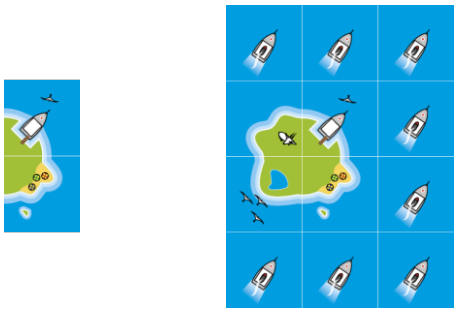


Abbildung J2.5: Beispiel 2; 4×3 Landschaft

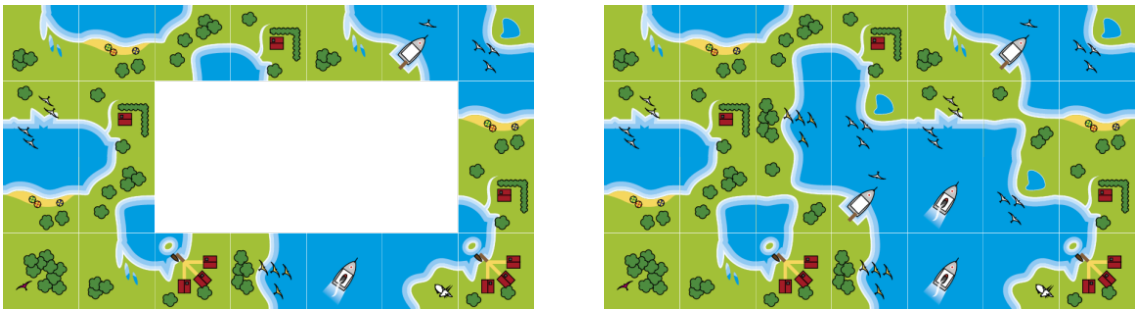


Abbildung J2.6: Beispiel 3; 4×7 Landschaft

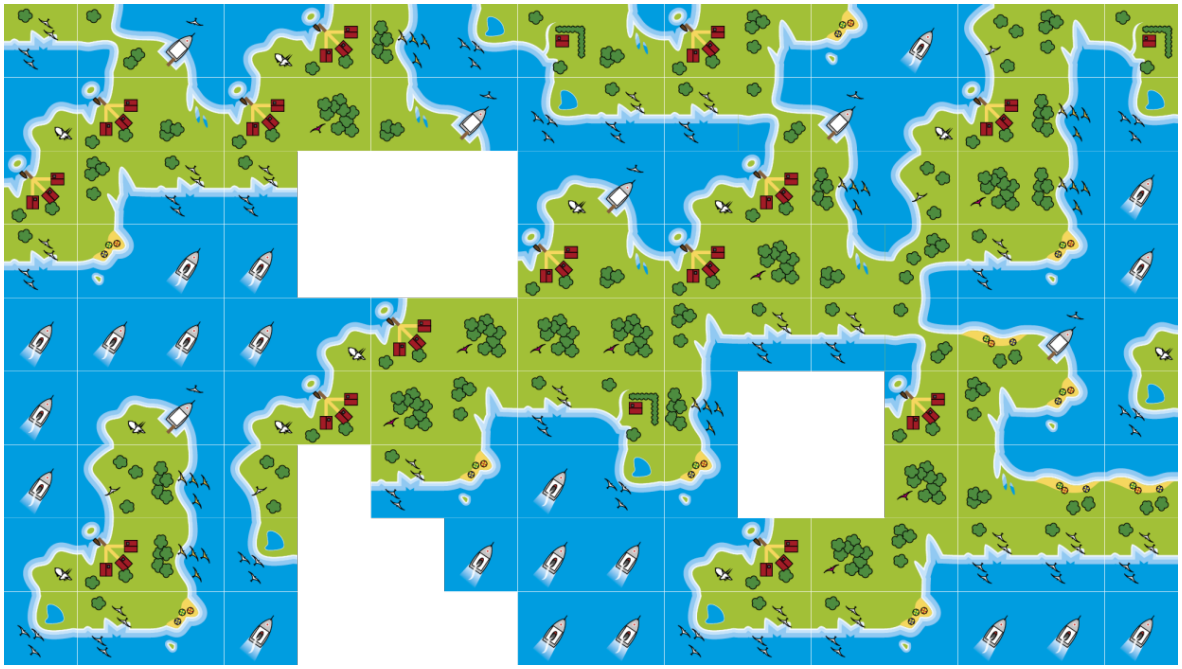


Abbildung J2.7: Beispiel 4; 9×16 Landschaft

Abbildung J2.8: Beispiel 5 (unlösbar); 5×5 Landschaft

J2.5 Bewertungskriterien

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

- $[-1]$ **Lösungsverfahren fehlerhaft**
  Ist keine Lösung möglich, so sollte dies erkannt werden. Ansonsten ist eine korrekte Lösung zu berechnen, d. h. benachbarte Kacheln stimmen in ihrer angrenzenden Seite überein und die bereits vorgegebenen Kacheln werden nicht verändert.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig**
  Es sind fast alle (korrekten) Verfahren erlaubt, solange sie keine offensichtlich unnötigen Dinge tun. Insbesondere sollte das Programm abbrechen, wenn es eine Stelle findet, an der bereits die in der Eingabekarte vorgegebenen Kacheln eine Vervollständigung unmöglich machen. So ist etwa ein Backtrackingverfahren, das bereits in der Eingabekarte vorhandene Konflikte nicht erkennt, nicht ausreichend.

- $[-1]$ **Ausgabe schlecht nachvollziehbar**
  Die Programmausgabe muss die Korrektheit der Lösung leicht überprüfbar machen. Dazu reicht ein Ausgabeformat, das dem Eingabeformat entspricht und binäre Werte (0 und 1, W und L, …) in einem passenden Raster ausgibt.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Ergebnisse zu den vorgegebenen Beispielen sollen mehrheitlich (mindestens drei von fünf, darunter das unlösbare Beispiel 5) dokumentiert und korrekt sein.

Aufgabe 1: Blumenbeet

1.1 Lösungsidee

Wir betrachten alle Hochbeete, die mit den Blumenfarben möglich sind, und rechnen jeweils aus, wie viele Punkte der Kunde für das Beet vergeben würde. Von allen (möglichen) Beeten mit höchster Punktzahl geben wir dann das zuerst gefundene aus. Wir lösen die Aufgabe also per Brute Force.

1.2 Modellierung

Im ersten Schritt modellieren wir ein leeres Blumenbeet. Wir nummerieren die Blumenplätze Zeile für Zeile jeweils von links nach rechts von 0 bis 8. Für jeden Platz geben wir dann an, welche zu ihm benachbart ist. Dies machen wir in einer Tabelle mit 9 Spalten und 9 Zeilen. Für jeden Platz gibt es eine Zeile und eine Spalte. Wenn der Platz, der zu einer Zeile gehört, mit einem Platz benachbart ist, der zu einer Spalte gehört, dann steht an der Stelle, an der sich diese Zeile und Spalte kreuzen, eine 1, ansonsten eine 0. Eine solche Tabelle heißt *Adjazenzmatrix*:

```
Blumenbeet =
[0 1 1 0 0 0 0 0 0
 1 0 1 1 1 0 0 0 0
 1 1 0 0 1 1 0 0 0
 0 1 0 0 1 0 1 0 0
 0 1 1 1 0 1 1 1 0
 0 0 1 0 1 0 0 1 0
 0 0 0 1 1 0 0 1 1
 0 0 0 0 1 1 1 0 1
 0 0 0 0 0 0 1 1 0]
```

Die Tabelle lässt sich zum Beispiel als zweidimensionales Array speichern.

Wir machen folgende Beobachtung: Die Tabelle hat eine Symmetrieachse, nämlich von der linken oberen Ecke in die rechte untere. Dies ist auch klar, denn wenn z. B. Platz 0 mit Platz 1 benachbart ist, so ist automatisch Platz 1 mit Platz 0 benachbart. Wir sagen, dass die Eigenschaft „benachbart“ *symmetrisch* ist.

Für die möglichen Blumenfarben definieren wir eine Liste:

```
Farben = [blau, gelb, gruen, orange, rosa, rot, tuerkis]
```

Jede Farbe bekommt eine Nummer, nämlich ihre Position in der Liste (blau ist 0, gelb ist 1, usw.). Viele Programmiersprachen bieten für solche Listen eine spezielle Unterstützung an, z. B. Python und Java mit *Enum*.

Als Nächstes müssen wir uns noch die Kundenangaben merken. Diese bestehen aus zwei Teilen: die Anzahl der gewünschten Farben sowie die Bewertung, die der Kunde für zwei benachbarte Farben gibt. Für Letzteres stellen wir eine Tabelle auf. Zu jeder Farbe gehört eine Zeile und eine Spalte. An der Stelle, an der sich die Zeilen und Spalten von zwei Farben kreuzen, steht die Bewertung des Kunden. Wenn keine Bewertung angegeben ist, gehen wir von 0 aus, da es dem Kunden egal ist.

Für das Beispiel `blumen.txt` aus der Aufgabenstellung sähe das wie folgt aus:

```
Farbanzahl = 7
Wuensche = 2
rot blau 3
rot tuerkis 2
Wunschmatrix =
[0 0 0 0 0 3 0
 0 0 0 0 0 0 0
 0 0 0 0 0 0 0
 0 0 0 0 0 0 0
 0 0 0 0 0 0 0
 3 0 0 0 0 0 2
 0 0 0 0 0 2 0]
```

Auch hier stellen wir fest, dass es eine Symmetrieachse von links oben nach rechts unten gibt. An dieser Stelle sollte man sich überlegen, was mit widersprüchlichen Angaben des Kunden passiert. Im Zweifel sollten diese abgewiesen werden. Ein Beispiel hierfür wäre die folgende Eingabe:

```
6
2
rot blau 3
blau rot 1
```

Ein Auswahl an Blumenzwiebeln, die an den Kunden geliefert wird (also das Ergebnis unserer Berechnung), speichern wir in einer Liste, in der Reihenfolge, in der wir die Blumenplätze nummeriert haben. Das Beispiel aus der Aufgabe ist also:

```
Zwiebeln = [rot, tuerkis, rosa, gelb, blau, orange, gruen, rot, tuerkis]
```

1.3 Umsetzung

Damit haben wir alles zusammen, um die Bewertung einer Zwiebelauswahl in der folgenden Funktion `Kundenbewertung` auszurechnen. Dies ist die in der Aufgabe definierte Zahl. Falls die Zwiebelauswahl nicht die gewünschte Anzahl an Farben hat, geben wir als Bewertung -1 zurück. Zu beachten ist, dass wir jedes Benachbartsein doppelt zählen, daher wird das Ergebnis zum Schluss noch durch 2 geteilt.

Damit können wir jetzt alle Möglichkeiten durchtesten. Hierfür verwenden wir eine *rekursive Tiefensuche*: Die Funktion startet mit dem ersten Steckplatz. Für jede der möglichen Farben ruft sie sich selbst wieder auf, beginnt jedoch beim nächsten Steckplatz. Dies geht so lange, bis wir bei der letzten Stelle angekommen sind. Dort prüfen wir dann, ob wir eine bessere Belegung gefunden haben. Dann geht es jeweils mit der nächsten Farbe weiter. So erhalten wir nach und nach alle möglichen Belegungen.

Wir starten mit dem Platz 0. Als Anfangsbelegung für `Startzwiebeln` wählen wir eine Liste der Länge 8, bei der die Einträge nicht gesetzt sind. Wir erhalten die beste gefundene Liste zurück.

Algorithmus 5 Funktion Kundenbewertung

```
procedure KUNDENBEWERTUNG(Blumenbeet, Zwiebeln, Farbzahl, Wünsche)
    if Anzahl verschiedener Elemente(Zwiebeln) ≠ Farbzahl then
        return −1
    end if
    Bewertung ← 0
    for i in 1..|Zwiebeln| do
        for j in 1..|Zwiebeln| do
            Bewertung ← Bewertung + Wünsche[Zwiebeln[i]][Zwiebeln[j]] * Blumenbeet[i][j]
        end for
    end for
    return Bewertung / 2
end procedure
```

Algorithmus 6 Suche

```
procedure SUCHE(Blumenbeet, Startplatz, Farbzahl, Wünsche, Zwiebeln)
    if Startplatz = |Zwiebeln| then
        return Zwiebeln                          ▷ Maximale Rekursionstiefe erreicht
    end if
    BesteVariante ← Zwiebeln
    for f in 1..Farbzahl do
        Zwiebeln[Startplatz] ← f
            ▷ Was ist das bestmögliche Beet mit der aktuellen Farben an dieser Position?
        AktuelleVariante ← Suche(Blumenbeet, Startplatz+1, Farbzahl, Wünsche, Zwiebeln)
        if KUNDENBEWERTUNG(Blumenbeet, AktuelleVariante, Farbzahl, Wünsche) >
KUNDENBEWERTUNG(Blumenbeet, BesteVariante, Farbzahl, Wünsche) then
            BesteVariante ← AktuelleVariante
        end if
    end for
    return BesteVariante
end procedure
```

1.4 Optimierungen

Die angegebene Funktion `Suche` kopiert die Zwiebelliste bei jedem Aufruf (also für jedes mögliche Beet). Außerdem verändern sich zwischen zwei hintereinander folgenden Aufrufen der Funktion `Kundenbewertung` meist nur wenige Einträge in der übergebenen Liste. Viele Rechenschritte werden also unnötigerweise mehrfach identisch ausgeführt. Beides lässt sich dadurch verbessern, dass immer die selbe Liste verwendet und verändert wird. Jedes Mal, wenn eine Veränderung der Blumenfarbe vorgenommen wird, muss nur die Differenz der Bewertung dieser veränderten Blume berechnet werden.

Weiterhin kann bei jedem Aufruf der Funktion `Suche` überprüft werden, ob überhaupt noch gültige Lösungen (also mit Bewertung größer als -1) möglich sind: Falls bereits zu viele verschiedene Farben verwendet wurden oder nicht mehr genügend Blumen zur Verfügung stehen, um die gewünschte Farbanzahl zu erreichen, kann abgebrochen werden. Dies ist eine Variante der Branch-and-Bound-Methode.

Außerdem reicht es aus, nur die in der Präferenzliste angegebenen Farben zu benutzen (ggf. mit zusätzlichen, beliebig gewählten Farben, um die gewünschte Farbanzahl zu erreichen). Durch die Verwendung von mehr Farben kann kein besseres Ergebnis erreicht werden.

1.5 Beispiele

Im Folgenden wird für die Beispiele eine Bepflanzung mit maximaler Bewertung mit der jeweils erreichten Punktzahl angegeben, außerdem noch die Anzahl der überprüften Beete unter Verwendung der besprochenen Optimierungen. Ohne diese wären jeweils $7^9 = 40.353.607$ Beete zu betrachten.

Beispiel 1:

```
- Bewertung: 36
- Hochbeet: [rot, tuerkis, tuerkis, rot, rot, rot, tuerkis, tuerkis, rot]
- Anzahl überprüfter Beete: 2.295
```

Beispiel 2:

```
- Bewertung: 32
- Hochbeet: [tuerkis, rot, rot, tuerkis, tuerkis, tuerkis, rot, rot, gruen]
- Anzahl überprüfter Beete: 19.674
```

Beispiel 3:

```
- Bewertung: 13
- Hochbeet: [rot, tuerkis, tuerkis, rosa, rot, gelb, blau, gruen, orange]
- Anzahl überprüfter Beete: 10.372.320
```

Beispiel 4:

```
- Bewertung: 22
- Hochbeet: [rot, tuerkis, rosa, gruen, rot, blau, orange, rosa, gelb]
- Anzahl überprüfter Beete: 10.372.320
```

Beispiel 5:

```
- Bewertung: 22
- Hochbeet: [rot, tuerkis, tuerkis, blau, rot, gruen, rosa, orange, gelb]
- Anzahl überprüfter Beete: 10.372.320
```

1.6 Bewertungskriterien

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

- $[-1]$ **Modellierung ungeeignet**
  Das vorgegebene Eingabeformat sollte eingehalten und korrekt in eine geeignete interne Darstellung umgesetzt werden. Auch das Beet und die Nachbarschaftsbeziehung der Blumen müssen korrekt umgesetzt sein. Insbesondere sind Listen geeignet. Es reichen aber auch einzelne Variablen, die über verschachtelte Schleifen belegt werden.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Das Programm muss für jede Eingabe eine Bepflanzung mit maximal möglicher Punktzahl finden.
- $[-1]$ **Bewertung der Bepflanzung fehlerhaft**
  Die Punktsumme einer Bepflanzung muss korrekt berechnet werden. Es passiert leicht, dass die (gegenseitige) Nachbarschaft zwischen zwei Blumen doppelt eingerechnet wird.
- $[-1]$ **Ausgabe schlecht nachvollziehbar**
  Die Ausgabe sollte darstellen, an welcher Stelle welche Blume zu pflanzen ist. Eine Liste der Blumenfarben reicht dazu aus, wenn beschrieben ist, welche Position in der Liste welcher Position im Beet entspricht. Außerdem sollte das Programm die Punktsumme der Bepflanzung ausgeben.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Ergebnisse zu den vorgegebenen Beispielen sollen mehrheitlich (mindestens drei von fünf) dokumentiert und korrekt sein.

Aufgabe 2: Nummernmerker

2.1 Vorbemerkung

Zuerst einmal stellt sich die Frage, ob wir jede Nummer nach den Regeln aufspalten können. Man überlegt sich leicht, dass z. B. die Nummer 00060 nicht ohne führende Nullen nach unseren Regeln aufgespalten werden kann, da, egal wie man den Anfang wählt, immer 0en vorne im ersten Block stehen. Auch wenn mehr als vier aufeinanderfolgende Nullen im Wort vorkommen, kann es nicht mehr ohne führende Null aufgespalten werden. Aber auch andere Wort wie z. B. 12000 lassen sich nicht ohne führende Nullen in mindestens einem Block aufspalten. Wir können auch feststellen, dass die Aufteilungen nicht immer eindeutig sind. So kann man die Nummer 120220 sowohl als 120 220 als auch als 1202 20 aufspalten. Natürlich wollen wir die Anzahl der Blöcke, die mit 0 beginnen, minimieren.

2.2 Lösungsidee

Eine erste Idee kann es sein, immer Blöcke fester Länge zu wählen, doch dies liefert z. B. bei 60040400 suboptimale Ergebnisse. Allerdings ist z. B. 600 40 400 eine gültige Aufspaltung. Wir stellen im folgenden eine Lösung vor, welche eine optimale Aufspaltung berechnet.

Rekursiver Algorithmus

Vielleicht kann man einfach alle Möglichkeiten durchgehen und später überprüfen, welche eine gültige Aufspaltung sind. Diese Form der naiven Suche hat leider exponentielle Laufzeit: Man hat mindestens bei jedem zweiten Zeichen die Entscheidung, dahinter aufzuspalten oder es nicht zu tun. Wir könnten am Ende der Nummer anfangen und zuerst die Größe des letzen Blockes bestimmen. Dann suchen wir rekursiv die optimale Lösung für die jeweils kürzere Nummer und nehmen das Minimum. Wollen wir z. B. die Nummer 234000 aufspalten, so versuchen wir rekursiv die Nummern 2340, 234 und 23 aufzuspalten. Für alle Nummern würden wir Kosten 0 berechnen, da aber nur bei der Aufspaltung 23 der nächste Block nicht mit einer 0 beginnt, würden wir diese Aufteilung wählen.

Dynamische Programmierung

Im Folgenden sprechen wir, wie in der theoretischen Informatik bei Zeichenketten üblich, von einem *Wort* statt einer Nummer. Es stellt sich heraus, dass wir das Problem durch Zerlegen in Teilprobleme lösen können. Wir lösen in der rekursiven Implementierung immer wieder die Frage, ob sich ein bestimmtes Präfix[2] des Wortes zerlegen lassen kann. Dabei ist es irrelevant, von wo aus der Rekursion heraus wir wissen möchten, ob dies so ist. D. h. wir können einfach die Antworten in einer Tabelle speichern (das nennt man auch *Memoization*) und würden unsere Laufzeit so senken.

Anstatt diese Tabelle rekursiv aufzufüllen, kann man sie auch direkt iterativ berechnen. Eine solches Vorgehen ist als *dynamische Programmierung* bekannt. Wir speichern in jeder Tabellenzelle $A[i]$ die minimale Anzahl von Nullen am Anfang eines Wortes, die wir benötigen um

[2] Ein Wort p heißt Präfix eines Wortes w, wenn ein Wort s existiert, sodass $w = ps$ ist. So sind z. B. M, Ma, Mau, Maus alle Präfixe von Maus.

Algorithmus 7 Rekursiver Algorithmus

```
function ZERTEILE(w, l)
    spaltK ← ∞
    spaltListe ← []                    ▷ Speichere aktuelle minimale Kosten und Abspaltung
    for i ← 2...4 do
        tk, tl ← ZERTEILE(w, l − i)
        if w[i] = 0 then
            tk ← tk + 1                ▷ Erhöhe Kosten um 1, da Aufspaltung mit 0 anfängt
        end if
        if tk.kosten < spaltK then
            spaltK = tk.kosten
            spalteListe = tk.liste ⊕ [l − i]          ▷ Hänge neuen Spaltpunkt an
        end if
    end for
    return (spalteK,spalteListe)
end function
```

die ersten i Ziffern aufzuspalten. Möchten wir dann z. B. $dp[j]$ berechnen, so können wir zwei, drei oder vier Positionen vor j eine neue Ziffer anfangen. Die Kosten errechnen sich dann als Kosten des neuen Blocks (die nur davon abhängen, ob die $i-j$-te Ziffer eine 0 ist) plus die alten Kosten (die wir schon berechnet haben).

Um die Lösung zu rekonstruieren, speichern wir uns, wenn immer wir $dp[i]$ setzen, wo der Block, welcher bei i endet, begann. Dann können wir am Ende die Aufspaltung rekonstruieren.

```
procedure ZERTEILE-DP(n)
    dp ← [∞, ..., ∞]
    prev ← [−1, ..., −1]                          ▷ Initialisere das Array
    for i ← 1...|n| do
        for j ← 2...4 do                          ▷ Schaue 2 bis 4 Zeichen zurück
            if i-j > 0 then
                nk ← dp[i − j]
                if n[i-j] = 1 then
                    nk ← nk + 1                   ▷ Erhöhe Kosten um 1, da Aufspaltung mit 0 anfängt
                end if
                if nk < dp[i] then                ▷ Besser als bisherige Aufspaltungen?
                    dp[i] = nk
                    prev[i] = i − j
                end if
            end if
        end for
    end for
end procedure
```

Laufzeit

Für eine Wort der Länge n, also eine Nummer mit n Ziffern, läuft unser vorgestellter Algorithmus in $O(n)$. Denn unser Array hat n Einträge und wir iterieren nur einmal über dieses. In jedem Iterationsschritt führen wir nur konstant viele Anweisungen aus; so ergibt sich die Laufzeit.

2.3 Alternative Lösungen

Der oben beschriebene Ansatz mit dynamischer Programmierung ist bei weitem effizient genug, um alle Beispiele in angemessener Zeit zu lösen. Es gibt aber noch andere Möglichkeiten dieses Problem zu lösen, wovon hier eine vorgestellt wird.

Dijkstra

Man kann auch einen gewichteten Graphen aufbauen, dessen kürzester Pfad einer Lösung entspricht. Hierzu erstellen wir für jede Ziffer einen Knoten und fügen eine Kante von Ziffer a_i zu Ziffer a_j hinzu, falls ihr Abstand zwischen 2 und 4 (inklusive) liegt. Wir gewichten die Kanten so, dass ein möglichst kurzer Pfad einer optimalen Aufteilung entspricht. Finden wir am Ende einen Pfad $(a_0, a_{i_1}, \ldots, a_{i_k})$ so wollen wir das so interpretieren, dass wir jeweils vor jeder der Ziffern a_{i_l} abschneiden. Da Nummern möglichst nicht mit 0 beginnen sollen, setzen wir das Kantengewicht auf 1 an jeder Kante, die zu einem Knoten a_i mit $a_i = 0$ führt. Dadurch erreichen wir, dass diese Kanten nur wenn es unbedingt nötig ist ausgewählt werden. Da wir den kürzesten Pfad finden, finden wir mit diesem Algorithmus eine optimale Lösung.

Dieser Algorithmus hat eine Laufzeit von $O(n \log n)$, was man in der Praxis fast nicht von unserem $O(n)$ Algorithmus unterscheiden kann. Man könnte mithilfe des Dijkstra-Algorithmus aber auch die Ausgabe verbessern: Indem man das Kantengewicht für beginnende 0en z. B. auf 1000 setzt und gleichzeitig das Gewicht für Kanten, die *vor* einer Null enden, als Quadrat der Länge nimmt, kann man erzwingen, dass statt 4040 (das Gewicht wäre hier $4^2 = 16$) die besser zu merkende Aufspaltung 40 40 (Gewicht: $2^2 + 2^2 = 8$) gewählt wird. Man kann also besonders schöne Auspaltungen finden.

2.4 Beispiele

Beispiel 1

```
Man muss mindestens mit 2 0en beginnen.
 Eine mögliche Aufteilung ist:
00 54 8000 0005 17 97 34
```

Beispiel 2

```
Man muss mindestens mit 1 0en beginnen.
 Eine mögliche Aufteilung ist:
03 49 59 29 53 37 901 54 41 26 60
```

Beispiel 3

```
Man muss mindestens mit 0 0en beginnen.
 Eine mögliche Aufteilung ist:
531 99 74 87 902 27 25 60 76 201 79
```

Beispiel 4

```
Man muss mindestens mit 0 0en beginnen.
 Eine mögliche Aufteilung ist:
90 88 76 10 51 69 94 82 78 90 38 33 12 67
```

Beispiel 5

```
Man muss mindestens mit 3 0en beginnen.
 Eine mögliche Aufteilung ist:
01 1000 0000 110 00 100 11 11 11 10 10 11
```

Schwierige Beispiele

Zu einfache Lösungsverfahren finden für u.a. die folgenden Beispiele nicht die Aufspaltung mit der kleinsten (in Klammern angegebenen) Zahl von Blöcken, die mit der Ziffer 0 beginnen: 111100011 (0), 111101001 (0), 111110001000 (0), 00000100010 (2) und 100010000101 (1).

2.5 Bewertungskriterien

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

- $[-1]$ **Lösungsverfahren fehlerhaft**
  Die Aufteilung einer Nummer muss regelkonform sein: 2–4 Ziffern pro Block.
- $[-1]$ **Aufteilungen nicht optimal**
  Das Lösungsverfahren muss immer eine optimale Aufteilung finden; die Anzahl der Blöcke, die mit einer Null beginnen, muss also minimal sein.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Ideal sind DP- oder Greedy-Verfahren. Aber auch Brute-Force ist ausreichend, wenn dabei nur die nach den Regeln möglichen Aufteilungsstellen berücksichtigt werden.
- $[-1]$ **Ausgabe schlecht nachvollziehbar**
  Die gesamte Nummer soll mit erkennbarer Aufteilung ausgegeben werden. Es genügt nicht, nur die Aufteilungspositionen auszugeben.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Ergebnisse zu den vorgegebenen Beispielen sollen mehrheitlich (mindestens drei von fünf) dokumentiert und korrekt sein.

Aufgabe 3: Telepaartie

Lösungsidee

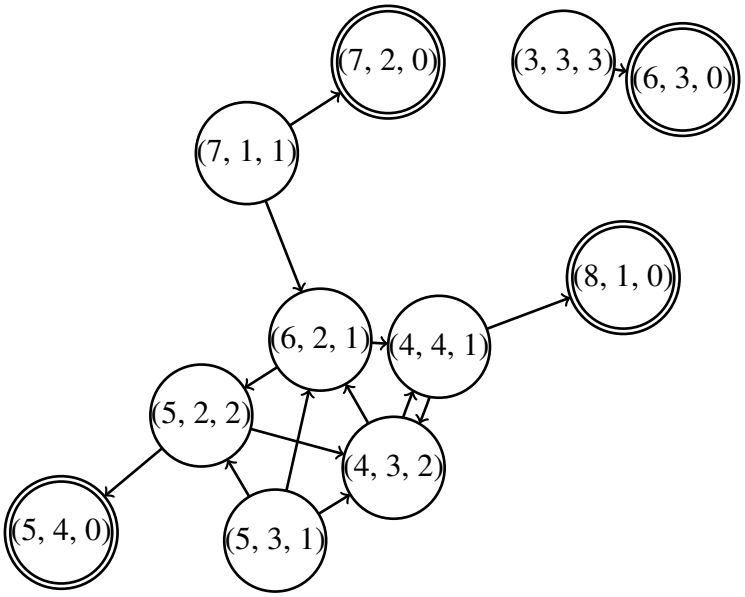
Es gibt keine einfache Formel, mit der wir auf einen Blick erkennen können, wie viele Telepaartien nötig sind, um einen Biber-Behälter zu leeren. Uns bleibt nichts Besseres übrig, als nach der Lösung systematisch zu suchen.

Um das Leeren eines Behälters zu simulieren, benötigen wir nur wenige Informationen, nämlich nur die Anzahl der Biber in den drei Behältern. Diese Information bezeichnen wir als *Konfiguration*. Wir nennen eine Konfiguration a in eine Konfiguration b *überführbar* (geschrieben $a \succ b$), wenn b aus a durch eine Telepaartie entsteht. Eine Konfiguration, in der ein Biber-Behälter leer ist, bezeichnen wir als *Endkonfiguration*. Man kann nun einen Graphen $G = (V,E)$ zeichnen, dessen Knoten V die Konfigurationen sind und dessen Kanten E gerade die überführbaren Konfigurationen verbinden. Diesen Graphen nennen wir *Konfigurationsgraphen*, für jede Anzahl n von Bibern ist er unterschiedlich. In Abbildung 3.1 ist der Konfigurationsgraph für $n = 9$ abgebildet.

Wie groß kann dieser Graph werden? Wir können zunächst feststellen, dass die Anzahl von Kanten nur linear in der Anzahl von Knoten wächst. Man kann maximal 3 unterschiedliche Telepaartien auf einer Verteilung durchführen ($\binom{3}{2} = 3$). Bei n Konfigurationen gibt es also maximal $3n$ Kanten.

Auch die Anzahl von Konfigurationen wächst nicht zu stark. Wenn wir n Biber haben, gibt es maximal n^2 viele Konfigurationen, da man für die Anzahl von Bibern in den ersten beiden Behältern jeweils maximal n Wahlmöglichkeiten hat und dann die Anzahl im dritten Behälter feststeht. Genauere Untersuchungen liefern auch nur quadratische Abschätzungen[3].

Abbildung 3.1: Graph für $n = 9$



[3] Betrachtet man die Anzahl von Konfigurationen für n, so erhält man die Sequenz A001399 in der OEIS, die explizite Formel wächst auch quadratisch.

3.1 Breitensuche

Um die LLL einer Konfiguration k_s zu bestimmen, muss man eine Folge von Konfigurationen $k_1, \ldots, k_l$ finden, sodass $k_i \succ k_{i+1}$ gilt und k_l eine Endkonfiguration ist. Bildlich gesprochen muss man im Konfigurationsgraphen die Anzahl von Schritten zur nächstgelegenen Endkonfiguration bestimmen. Ein in der Informatik oft benutzter Algorithmus zur Bestimmung von kürzesten Wegen in (ungewichteten) Graphen ist die sogenannte Breitensuche. Hierbei beginnt man bei der Startkonfiguration und bestimmt Stück für Stück die Knoten, welche sich in genau einem, zwei, drei usw. vielen Schritten erreichen lassen. Wenn man die Konfigurationen K_i, welche in i Schritten erreicht werden können kennt, so kann man die in $i+1$ Schritten erreichbaren Konfigurationen berechnen, indem man alle Nachfolgekonfigurationen von Konfigurationen in K_i betrachtet, welche noch nicht vorher besucht wurden. Merkt man sich zusätzlich für jeden Knoten den Vorgänger, so kann man im Anschluss auch den Pfad rekonstruieren.

Algorithmus 8 Breitensuche (ohne Pfadrekonstruktion)

```
1:  function BFS(G = (V,E), s)
2:      Initialisiere dist[v] ← ∞ für alle v ∈ V
3:      Initialisiere Queue Q = {s}
4:      dist[start] ← 0
5:      while q ≠ ∅ do
6:          v ← Q.pop()
7:          if v ist Endkonfiguration then
8:              return dist[v]
9:          end if
10:         for Nachfolger v von u do                    ▷ , d. h. (u,v) ∈ E
11:             if dist[u] = ∞ then
12:                 dist[u] ← dist[v]+1
13:                 Q.push(u)
14:             end if
15:         end for
16:     end while
17: end function
```

Anstatt den kompletten Konfigurationsgraphen aufzubauen, genügt es bei dieser Teilaufgabe, sich immer nur den besuchten Teilgraphen zu speichern. Wenn der kürzeste Pfad Länge l hat und es maximal k Verzweigungen gibt, hat die Breitensuche Laufzeit $O(k^l)$. Auf der anderen Seite ist die Anzahl von Knoten im Graphen eine obere Schranke für die Laufzeit, wir erreichen also mit Optimierung eine Laufzeit von $O(\min(3^n, n^2))$, ohne Optimierung nur $bigOn^2$.

3.2 Abgelegenste Konfiguration

Im zweiten Aufgabenteil sollen wir eine Verteilung (d. h. Konfiguration) finden, welche eine besonders lange LLL hat. Eine mögliche (aber langsame) Lösung wäre es, für jede Konfiguration den Algorithmus aus Aufgabenteil a) auszuführen.

Wenn wir im Graphen für jede Konfiguration k den Abstand d_k zur nächstgelegenen Endkonfiguration kennen würden, so hätten wir das Problem gelöst, da wir dann einfach eine Konfiguration mit höchstem d_k benutzen könnten. Wir möchten also diesen Abstand möglichst schnell

berechnen. Oft kann man Zählprobleme rekursiv lösen. Das heißt, man zerlegt das Problem in einfache Basisfälle und benutzt für die anderen Fälle die Lösung des Problems für kleinere Teilprobleme. Für Endkonfigurationen wissen wir, dass $d_e = 0$ ist, da Endkonfigurationen in keinem Schritt Endkonfigurationen erreichen können. Für alle Konfigurationen, die Endkonfigurationen in einem Schritt erreichen können, wissen wir nun, dass sie Distanz 1 haben. Dieses Argument kann man iterieren und so die Distanz für alle Knoten bestimmen. Wir verwenden im Endeffekt folgende Rekursionsgleichung zur Berechnung:

$$d_k = \begin{cases} 0 & \text{falls } k \text{ Endkonfiguration} \\ \min_{(k,u)\in E} d_u + 1 & \text{sonst} \end{cases}$$

Um den Algorithmus schnell auszuführen, müssen wir auf die Vorgänger eines Knotens möglichst schnell zugreifen können. Hierzu kann man sich zu jedem Knoten am Anfang einmal die Liste der Vorgängerkonfigurationen berechnen (den so entstehenden Graphen nennt man auch *transponierten*[4] Graphen). Vergleicht man den Pseudocode, so fällt auf, dass Algorithmus 9 einer Breitensuche im transponierten Graphen ausgehend von allen Endkonfigurationen entspricht. Dieser Algorithmus hat lineare Laufzeit in der Anzahl der Knoten und Kanten, läuft also in $O(n^2)$.

Algorithmus 9 Berechne d_k Distanz von Endkonfiguration

```
function DISTANZ(G = (V,E), s)
    Initialisiere d[v] ← ∞ für alle v ∈ V
    Initialisiere Queue Q = ∅
    for Endkonfiguration k do
        d[k] ← 0
        Q.push(k)
    end for
    while q ≠ ∅ do
        v ← Q.pop()
        for Vorgänger v von u do          ▷ nutze hier vorberechnete Liste
            dist[v] ← dist[u]+1
            Q.push(v)
        end for
    end while
    return d
end function
```

3.3 Beispiele

Erster Aufgabenteil

```
~/bwinf_38/biber python3 teilA.py 2 4 7
Die Lösung ist 2
(7, 4, 2) (5, 4, 4) (8, 5, 0)
```

[4] Der Name rührt daher, dass die Adjazenzmatrix des transponierten Graphen gerade die Transponierte Adjazenzmatrix ist.

```
~/bwinf_38/biber python3 teilA.py 3 5 7
Die Lösung ist 3
(7, 5, 3) (10, 3, 2) (7, 6, 2) (7, 4, 4) (8, 7, 0)

~/bwinf_38/biber python3 teilA.py 80 64 32
Die Lösung ist 2
(80, 64, 32) (64, 64, 48) (128, 48, 0)
```

Zweiter Aufgabenteil

Wir geben einige Beispiele bis $n = 200$ an.

```
Maximum für 1 Biber: 0, Konfiguration (1, 0, 0)
Maximum für 2 Biber: 0, Konfiguration (1, 1, 0)
Maximum für 3 Biber: 1, Konfiguration (1, 1, 1)
Maximum für 4 Biber: 1, Konfiguration (2, 1, 1)
Maximum für 5 Biber: 1, Konfiguration (2, 2, 1)
Maximum für 6 Biber: 2, Konfiguration (3, 2, 1)
Maximum für 7 Biber: 2, Konfiguration (4, 2, 1)
Maximum für 8 Biber: 2, Konfiguration (4, 3, 1)
Maximum für 9 Biber: 2, Konfiguration (6, 2, 1)
Maximum für 10 Biber: 2, Konfiguration (5, 4, 1)
Maximum für 11 Biber: 3, Konfiguration (6, 4, 1)
Maximum für 12 Biber: 3, Konfiguration (5, 4, 3)
Maximum für 13 Biber: 3, Konfiguration (8, 3, 2)
Maximum für 14 Biber: 3, Konfiguration (9, 4, 1)
Maximum für 15 Biber: 4, Konfiguration (8, 4, 3)
Maximum für 16 Biber: 3, Konfiguration (11, 4, 1)
Maximum für 17 Biber: 3, Konfiguration (7, 6, 4)
Maximum für 18 Biber: 3, Konfiguration (13, 4, 1)
Maximum für 19 Biber: 4, Konfiguration (10, 5, 4)
Maximum für 20 Biber: 3, Konfiguration (14, 5, 1)
...
Maximum für 30 Biber: 5, Konfiguration (19, 8, 3)
Maximum für 40 Biber: 4, Konfiguration (29, 6, 5)
Maximum für 50 Biber: 6, Konfiguration (32, 13, 5)
Maximum für 60 Biber: 6, Konfiguration (34, 21, 5)
Maximum für 70 Biber: 6, Konfiguration (51, 12, 7)
Maximum für 80 Biber: 5, Konfiguration (44, 31, 5)
Maximum für 90 Biber: 7, Konfiguration (49, 32, 9)
Maximum für 100 Biber: 7, Konfiguration (64, 31, 5)
Maximum für 110 Biber: 7, Konfiguration (63, 37, 10)
Maximum für 120 Biber: 7, Konfiguration (79, 36, 5)
Maximum für 130 Biber: 7, Konfiguration (67, 34, 29)
Maximum für 140 Biber: 7, Konfiguration (103, 23, 14)
Maximum für 150 Biber: 8, Konfiguration (79, 50, 21)
Maximum für 160 Biber: 6, Konfiguration (84, 71, 5)
Maximum für 170 Biber: 8, Konfiguration (105, 48, 17)
Maximum für 180 Biber: 8, Konfiguration (113, 49, 18)
Maximum für 190 Biber: 9, Konfiguration (111, 60, 19)
Maximum für 200 Biber: 8, Konfiguration (128, 67, 5)
```

3.4 Bewertungskriterien

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

- $[-1]$ **Modellierung der Zustände ungeeignet**
  Das Problem sollte geeignet modelliert werden. Dies bedeutet insbesondere, dass eine sinnvolle Repräsentation der Zustände gefunden wurde. Es ist alles OK (wie Listen oder Tupel mit 3 Elementen), was die Berechnung der Leerlauflänge nicht unnötig verkompliziert.
- $[-1]$ **Lösungsverfahren bzw. Implementierung fehlerhaft**
  Insbesondere der Zustandsübergang durch Telepaartie muss korrekt berechnet werden.
- $[-1]$ **Verfahren unnötig aufwendig / ineffizient**
  Beim ersten Aufgabenteil, also der Berechnung einer LLL, ist es ausreichend, eine Breitensuche vom Startzustand aus durchzuführen. Im zweiten Teil, also der Berechnung der Werte $L(1),\ldots,L(n)$, sollte das Verfahren zumindest eine Eigenschaft des Problems zur Optimierung ausnutzen, also zum Beispiel bereits erreichte Zustände erkennen, die Zustände normalisieren usw. Eine wiederholte Anwendung der LLL-Berechnung ist zwar nicht nötig, aber akzeptabel.
- $[-1]$ **Ausgabe schlecht nachvollziehbar**
  Die Ergebnisse, insbesondere die Umfüllfolgen bei Aufgabenteil 1, sollten leicht nachvollziehbar und überprüfbar sein.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Für Aufgabenteil 1 soll mindestens eine Umfüllfolge gezeigt werden. Das Ergebnis für die Eingabe 10 (also die Werte $L(1),\ldots,L(10)$) soll vollständig, die Werte $L(11),\ldots,L(100)$ sollen mindestens teilweise angegeben sein.

Aufgabe 4: Urlaubsfahrt

4.1 Lösungsidee

Wenn wir die Zahl an Tankstellenstopps nach oben begrenzen, können wir vom Startpunkt aus mit aufsteigender Zahl an Maximalstopps versuchen, an das Ziel zu gelangen.

Wir stellen eine rekursive Funktion auf, die für einen gegebenen Kilometer, verbleibende erlaubte Tankstopps und eine Tankfüllung die günstigst-mögliche Variante ausgibt, das Ziel zu erreichen. Sobald nur noch ein Stopp verbleibt (der an der Tankstelle, an der wir gerade stehen), wird evaluiert ob das Ziel ohne weiteren Stopp erreichbar ist.

- Ist das Ziel im Rahmen des Tankvolumens erreichbar, fülle den Tank soweit auf, dass das Ziel direkt erreicht wird.
- Ist das Ziel nicht erreichbar, markiere die Kosten für dieses Verhalten als ∞.

Bei mehr verfügbaren Stopps werden alle erreichbaren Stopps angefahren und rekursiv mit einem Stopp weniger und der verbleibenden Tankfüllung evaluiert. Dabei wird wie folgt an der aktuellen Station getankt:

- Ist die Zieltankstelle günstiger oder gleich teuer wie die aktuelle Tankstelle, fülle den Tank genau auf um diese zu erreichen. (Dies setzt voraus, dass die Tankstelle nicht auch ohne Auftankung erreicht hätte werden können.)
- Ist die Zieltankstelle teurer als die aktuelle Tankstelle, fülle den Tank vollkommen auf und fahre bis zu der Tankstelle.

Damit wird eine Laufzeit von $O(mn^3)$ erreicht, bei einer Maximalzahl an Stopps m und n Tankstellen. Im Wesentlichen müssen nämlich alle Kombinationen an möglichen verbleibenden Halten und möglichen Tankfüllungen für jede Tankstelle getestet werden. Diese sind durch die Zahl der Tankstellen nach oben beschränkt (siehe folgenden Absatz).

4.2 Optimierung durch DP und Umsortierung

Das obige Verfahren kann durch dynamische Programmierung optimiert werden. Dazu identifizieren wir die drei Variablen, nach denen tabellarisch Zwischenergebnisse gespeichert werden können. Jedes Zwischenergebnis von Interesse kann durch die Tankstelle (ihren Index oder Kilometer), die Anzahl verbleibender erlaubter Stopps und den restlichen, bei Ankunft im Tank vorhandenen Kraftstoff identifiziert werden.

Damit nicht ein Zwischenergebnis für beliebige Tankfüllungen berechnet werden muss, werden die obigen Tankregeln (Abschnitt 4.1) beachtet. Diese bedingen, dass für jede Tankstelle nur solche Restfüllungen möglich sind, die der Rest von der Anfahrt einer teureren vorhergehenden Tankstelle sind (durch Regel 2). Ansonsten kann der Tank noch leer sein. Damit könnte die Tabelle der Zwischenergebnisse wieder in $O(mn^3)$ naiv befüllt werden (statt in pseudopolynomieller Zeit, was durch eine Diskretisierung der Tankfüllungen hätte erreicht werden können.)

Diese Anordnung erlaubt jedoch, bei der Berechnung der Zwischenschritte eine Optimierung vorzunehmen. Damit können wir für eine Tankstelle und Anzahl verbleibender Stopps alle Einträge für alle Restfüllungen generieren. Die von einer Tankstelle aus als nächstes angestrebte

Tankstelle ist größtenteils unabhängig von der aktuellen Resttankfüllung. Der Kostenterm abhängig von der Restfüllung ist für alle Zieltankstellen gleich. Daher bestimmen wir lediglich einmal die möglichen Kostenterme unabhängig von der Restfüllung und wählen den minimalen daraus aus um auf ihn die von der Restfüllung abhängigen Terme aufzuaddieren (so erhalten wir die tatsächlichen Kosten).

Dabei müssen wir lediglich die Einschränkung beachten, dass die Restfüllung nicht ausreichen darf, um die nächste Tankstelle zu erreichen (siehe Einschränkung bei Abschnitt 4.1). Ist dies der Fall, verwerfen wir den zugehörigen Term und suchen in der sortierten Liste den nächstgrößeren gültigen unabhängigen Term. Dieser Schritt wird durch die Sortierung der Terme dominiert und ermöglicht eine Gesamtlaufzeit von $O(mn^2 \log n)$

Details und zugehörige Formeln sind in [5] unter Abschnitt 2.1 “The gas station problem using δ stops” zu finden, auf dem dieser Ansatz basiert.

In unserem Problem ist der DP-Ansatz jedoch im Feldversuch deutlich zeit- und speicherintensiver als ein naiver rekursiver Ansatz. Dies weist darauf hin, dass ein großer Teil der möglichen Tabelleneinträge nie genutzt wird.

4.3 Optimierung durch Pruning

Deutlich beschleunigt wird das Programm durch das Überspringen von Zwischenstopps, die sicher nicht mit der Zahl an verbleibenden Stopps bis zum Ziel führen. Dies kann bestimmt werden, indem berechnet wird, wie oft auf Basis des Verbrauchs und Tankvolumens mindestens angehalten werden müsste, um von diesem Kilometer aus an das Ziel zu gelangen. Die Formel für die Mindestanzahl an weiteren Stopps bis zum Ziel ist

$$Stopps \geq \frac{Zieldistanz * Verbrauch}{Tankvolumen} \tag{4.1}$$

Dies lässt sich schnell umformen zu einer Mindestdistanz zum Ziel, wenn nur noch eine gewisse Zahl an Stopps verbleiben.

$$Zieldistanz \leq Stopps * \frac{Tankvolumen}{Verbrauch} \tag{4.2}$$

Zieht man diese Distanz zum Ziel von der Länge der Fahrt (also der Gesamtdistanz des Ziels) ab, so erhält man einen Mindeststreckenkilometer, ab welchem die nächste Tankstelle stehen darf. Da hier eigentlich noch die Tankfüllung bei Ankunft an diesem Kilometer berücksichtigt werden müsste, kann eine untere Abschätzung für den Mindestkilometer durch einen imaginären zusätzlichen Stopp am Start erreicht werden.

Diese Optimierung beschleunigt den Algorithmus in den meisten Fällen signifikant. Für ein Problem der Größe von 100000km /120 Stopps sank die Rechenzeit von 48s auf 3s. Ein Problem der Größe 1000000km / 1154 Stopps, das unoptimiert etwa ein Terabyte Hauptspeicher belegen würde, konnte so in 267s mit lediglich 4 GB Speicherbelegung berechnet werden.

[5] Khuller, Samir, Azarakhsh Malekian, and Julián Mestre. “To fill or not to fill: The gas station problem.” ACM Transactions on Algorithms (TALG) 7.3 (2011): 36.

4.4 Reduktion der Durchläufe

Mithilfe des Fasty-Algorithmus aus Abschnitt 4.5 können wir direkt die Mindestzahl an Stopps berechnen, die benötigt werden, um das Ziel zu erreichen. Sämtliche Versuche mit weniger Stopps das Ziel zu erreichen, können also übersprungen werden. Damit sinkt die benötigte Laufzeit auf $O(n^3)$.

4.5 Alternative Lösungsansätze

Greedy und Fasty Lösung

Grundsätzlich existieren relativ einfache Ansätze, das Problem möglichst kostengünstig oder mit möglichst wenig Stopps zu lösen. Für eine kostengünstige Lösung werden die Regeln aus Abschnitt 4.1 angewandt, jedoch nicht auf alle erreichbaren, sondern lediglich auf die kostengünstigste billigere Tankstelle oder, falls keine billigere Tankstelle erreichbar ist, die nächste Tankstelle (in der Hoffnung von dort aus eine günstigere Tankstelle erreichen zu können). Details dazu sind in [6] zu finden. Für die Fahrt mit den wenigsten Stops wird stets die am weitesten entfernte Tankstelle angefahren. Hier kann auch das Tankverhalten mithilfe der obigen Regeln optimiert werden. Im Allgemeinen sind diese Ansätze jedoch nicht gefragt und gehen an der expliziten Frage nach der Auswahl geeigneterer günstigerer Tankstellen vorbei.

Greedy mit Restriktionen

Eine einfache Erweiterung des Greedy-Algorithmus, die der Aufgabenstellung schon deutlich näher kommt, ist die Einschränkung, dass der obige Greedy-Algorithmus lediglich Tankstellen in Betracht ziehen darf, bei denen der Tank bereits (nahezu) vollständig entleert ist. Hier ist eine geschickte Definition von "nahezu vollständiger Entleerung"wünschenswert. Neben festen Werten von 10-30% der Tankfüllung ist es auch aufgrund der sehr geringen (linearen) Laufzeit gut möglich, den Algorithmus automatisch verschiedene Prozentwerte oder Absolutgrenzen austesten zu lassen, um eigenständig ein Optimum zu finden. Die Ergebnisse dieses Ansatzes können jedoch von der Optimallösung stark abweichen.

4.6 Beispiele

Für eines der vorgegebenen Beispiele könnte das eben verbesserte Verfahren so ablaufen:

fahrt2.txt

```
KM0: Versuche mit 273 L Diesel und 11 Stopps das Ziel zu erreichen
KM0: Mindestkilometer um das Ziel in 11 Stopps zu erreichen: -2558.333333333334
KM0: Fahre KM24 von start an, nichts Tanken
KM24: Versuche mit 267.24 L Diesel und 10 Stopps das Ziel zu erreichen
KM24: Mindestkilometer um das Ziel in 10 Stopps zu erreichen: -1416.666666666666
```

[6] Lin, Shieu Hong, Nate Gertsch, and Jennifer R. Russell. “A linear-time algorithm for finding optimal vehicle refueling policies.” Operations Research Letters 35.3 (2007): 290-296.

```
KM24: Fahre KM157 an, teurer also volltanken, Kosten: 6.759999999999991 L * 119
    EUR/L
KM157: Versuche mit 242.07999999999998 L Diesel und 9 Stopps das Ziel zu
    erreichen
KM157: Mindestkilometer um das Ziel in 9 Stopps zu erreichen: -275.0
KM157: Fahre KM256 an, teurer also volltanken, Kosten: 31.920000000000016 L *
    140 EUR/L
KM256: Versuche mit 250.24 L Diesel und 8 Stopps das Ziel zu erreichen
KM256: Mindestkilometer um das Ziel in 8 Stopps zu erreichen: 866.6666666666661
KM256: Fahre KM1375 an, billiger also nur wenig Tanken, Kosten:
    18.319999999999993 L * 141 EUR/L
KM1375: Versuche mit 0 L Diesel und 7 Stopps das Ziel zu erreichen
KM1375: Mindestkilometer um das Ziel in 7 Stopps zu erreichen: 2008.333333333333
KM1375: Fahre KM2041 an, teurer also volltanken, Kosten: 274 L * 130 EUR/L
KM2041: Versuche mit 114.16 L Diesel und 6 Stopps das Ziel zu erreichen
KM2041: Mindestkilometer um das Ziel in 6 Stopps zu erreichen: 3150.0
KM2041: Fahre KM3162 an, billiger also nur wenig Tanken, Kosten:
    154.88000000000002 L * 141 EUR/L
KM3162: Versuche mit 0 L Diesel und 5 Stopps das Ziel zu erreichen
...
KM6651: Fahre KM7737 an, teurer also volltanken, Kosten: 268.08 L * 119 EUR/L
KM7737: Versuche mit 13.360000000000014 L Diesel und 1 Stopps das Ziel zu
    erreichen
KM7737: 1 Stopp -> Ziel nicht direkt erreichbar, Kosten: inf
KM7737: Finale Entscheidung fällt auf KM10000, mit Kosten inf EUR
KM6651: Finale Entscheidung fällt auf KM7737, mit Kosten inf EUR
...
KM7589: Mit 48.879999999999995 L Diesel und 3 Stopps Ergebnis bekannt, Kosten
    63013.2 EUR
KM6651: Fahre KM7653 an, billiger also nur wenig Tanken, Kosten: 240.48 L * 119
    EUR/L
KM7653: Mit 0 L Diesel und 3 Stopps Ergebnis bekannt, Kosten 66247.92 EUR
KM6651: Fahre KM7737 an, teurer also volltanken, Kosten: 274 L * 119 EUR/L
KM7737: Mit 13.360000000000014 L Diesel und 3 Stopps Ergebnis bekannt, Kosten
    62849.59999999999 EUR
KM6651: Finale Entscheidung fällt auf KM7049, mit Kosten 94575.12 EUR
KM5581: Finale Entscheidung fällt auf KM6452, mit Kosten 125827.2 EUR
KM4455: Finale Entscheidung fällt auf KM5352, mit Kosten 158441.92 EUR
KM3346: Finale Entscheidung fällt auf KM4289, mit Kosten 187529.2 EUR
KM2240: Finale Entscheidung fällt auf KM3346, mit Kosten 225333.6 EUR
KM1118: Finale Entscheidung fällt auf KM1922, mit Kosten 253032.91999999998 EUR
KM0: Finale Entscheidung fällt auf KM1118, mit Kosten 253032.91999999998 EUR
processing time: 0.1919269561767578 s
```

Übersicht

In der folgenden Tabelle sind die optimalen Kosten (in Euro) und Anzahlen Stopps für alle Beispiele und ein paar alternative Lösungsansätze zu finden.

#	Optimum	R. Greedy	Stopps	Treibstoff [l][7]	Greedy[8]	(Stopps)	Fasty[9]
1	81.70	81.70	2	58	80.26	(3)	116.50
2	2530.33	2567.90	9	2127	2477.06	(13)	2694.63
3	99.20	99.20	1	80	95.70	(7)	131.56
4	236.88	236.96	2	189	230.21	(10)	270.16
5	2494.32	2518.54	9	2058	2387.89	(16)	2731.42

4.7 Bewertungskriterien

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

- $[-2]$ **Minimierung der Tankvorgänge unzureichend**
  $[-1]$ Es wird nicht immer genau die geringste Anzahl an Tankvorgängen (Stopps) gewählt, aber eine Optimierung zu einer geringen Anzahl von Stopps hin ist erkennbar.
  $[-2]$ Es wird gar keine Optimierung bzgl. der Anzahl an Stopps vorgenommen.
- $[-2]$ **Minimierung der Kosten unzureichend**
  $[-1]$ Es wird nicht die optimale Lösung berechnet (zum Beispiel durch einen eingeschränkten Greedy-Algorithmus) oder lediglich ein zu den Tankregeln ähnliches Verfahren angewandt (Fasty). Auch wenn immer vollgetankt wird, werden die Kosten nicht immer minimiert.
  $[-2]$ Es wird gar keine Optimierung bezüglich der Tankkosten vorgenommen.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Nur besonders umständliche Verfahren führen zu Punktabzug. Bei bekannter minimaler Anzahl der Tankvorgänge ist ein Brute-Force-Ansatz zur Ermittlung der Variante mit minimalen Kosten ausreichend.
- $[-1]$ **Ausgabe schlecht nachvollziehbar**
  Das Programm muss übersichtlich auflisten, an welchen Tankstellen wie viel Diesel getankt wird. Schön ist, wenn auch die Gesamtkosten angegeben werden.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Ergebnisse zu den vorgegebenen Beispielen sollen mehrheitlich (mindestens drei von fünf) dokumentiert und korrekt sein. Rundungsfehler werden akzeptiert.

[7] Jede korrekte Lösung sollte diese Werte berechnen – exklusive der Tankfüllung bei Kilometer 0.
[8] Geringer können die Kosten nicht sein.
[9] Größer sollten die Kosten nicht sein.

Aufgabe 5: Rominos

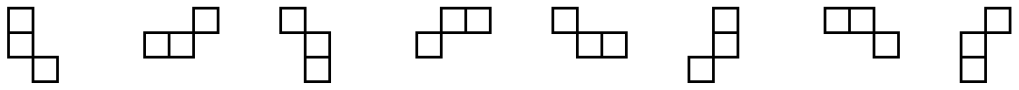
5.1 Lösungsidee

Um die Zusammensetzung eines n-Rominos zu modellieren, verwenden wir ein Array der Größe $n \times n$, in dem nach und nach n Felder mit einem Wert belegt werden, der einem Quadrat entspricht. Als Lösungsverfahren nutzen wir einen Backtracking-Ansatz: Beginnend mit der Position $(0,0)$ (oder einem beliebigen anderen Startfeld) wird entschieden, ob die aktuelle Position Teil des n-Rominos ist, und der Algorithmus rekursiv für beide Möglichkeiten aufgerufen. Wir speichern eine Liste von Nachbarfeldern des aktuellen n-Rominos (also derjenigen Felder, die schon als Teil des n-Rominos festgelegt wurden). Dies sind die Felder, die noch betrachtet werden müssen. Wir betrachten niemals Felder, die nicht Nachbarn des aktuellen n-Rominos sind, oder für die schon entschieden wurde, ob sie Teil des n-Rominos sein sollen.

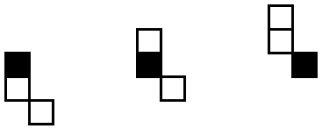
Sobald ein n-Romino die Größe n erreicht hat, werden keine weiteren Felder hinzugefügt und es wird geprüft, ob das Romino gültig ist. Das ist es genau dann, wenn mindestens zwei Quadrate wie in der Aufgabenstellung beschrieben nur an einer Ecke aneinander liegen. Falls es gültig ist, wird es zu einer vorläufigen Ergebnisliste hinzugefügt.

5.2 Entfernen von Duplikaten

Nach dem Ausführen des obigen Algorithmus enthält diese Ergebnisliste bis zu 8 verschiedene Rotationen und Spiegelungen jedes n-Rominos. Durch drei Rotationen um 90 Grad, eine Spiegelung und weitere drei Rotationen können diese alle gefunden werden.



Außerdem wird ein n-Romino mit fixer Rotation jeweils genau n mal erzeugt, wobei jeweils ein anderes Feld das Startfeld war.



Um für zwei gegebene n-Rominos herauszufinden, ob sie gleich sind, müssen wir zum einen alle 8 Rotationen/Spiegelungen ausprobieren, und zum anderen die Koordinaten normalisieren. Das heißt zum Beispiel, dass wir das n-Romino so verschieben, dass die niedrigste verwendete x- und y-Koordinate jeweils 0 ist.

Um nicht alle Paare von n-Rominos in der vorläufigen Ergebnisliste vergleichen zu müssen, bietet es sich an, die normalisierten n-Rominos in einer Hashmap zu speichern.

5.3 Optimierungen

Bisher erzeugt der Algorithmus jede Rotation/Spiegelung jedes n-Rominos n mal. Das lässt sich vermeiden, indem man festlegt, dass das Startfeld eines der linkesten Felder des n-Rominos und von den linkesten Feldern das oberste sein soll. Felder links von bzw. direkt über dem Startfeld werden nicht hinzugefügt. Dadurch beschleunigt sich das Programm ca. um den Faktor n.

Abgesehen von der Laufzeit kann auch der Speicherverbrauch bei größeren n ein Problem werden. Dieser lässt sich zumindest um einen Faktor 8 reduzieren, wenn man schon vor dem hinzufügen eines n-Rominos zur Ergebnisliste (oder Ergebnishashmap) prüft, ob bereits ein bis auf Rotationen und Spiegelungen gleiches n-Romino gefunden wurde und es dann gegebenenfalls nicht hinzufügt.

5.4 Beispiele

Die folgende Tabelle zeigt die Anzahl n-Rominos bis $n = 11$:

n	**Anzahl n-Rominos**	n	**Anzahl n-Rominos**
2	1	7	2 924
3	3	8	18 406
4	17	9	116 883
5	82	10	753 905
6	489	11	4 899 488

Es folgen alle 4- und 5-Rominos. Schon ab $n = 6$ ist die Anzahl zu groß, um sie alle hier abzudrucken.

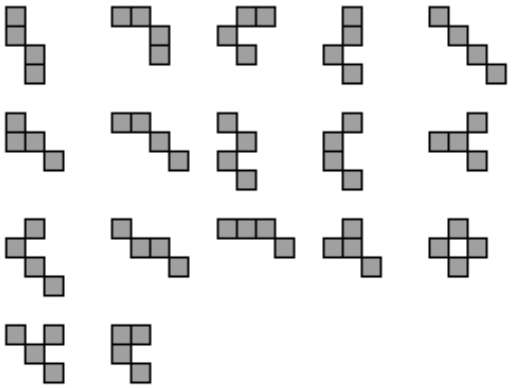


Abbildung 5.1: 4-Rominos

5.5 Alternative Lösungswege

Für Polyominos (die Verallgemeinerung der Pentominos aus der Aufgabenstellung) sind viele Algorithmen dokumentiert[10], um sie zu generieren oder auch nur ihre Anzahl zu zählen. Diese lassen sich meist leicht angepasst auch auf die Rominos anwenden.

[10] Eine Übersicht findet sich unter anderem auf Wikipedia:
https://en.wikipedia.org/wiki/Polyomino#Algorithms_for_enumeration_of_fixed_polyominoes

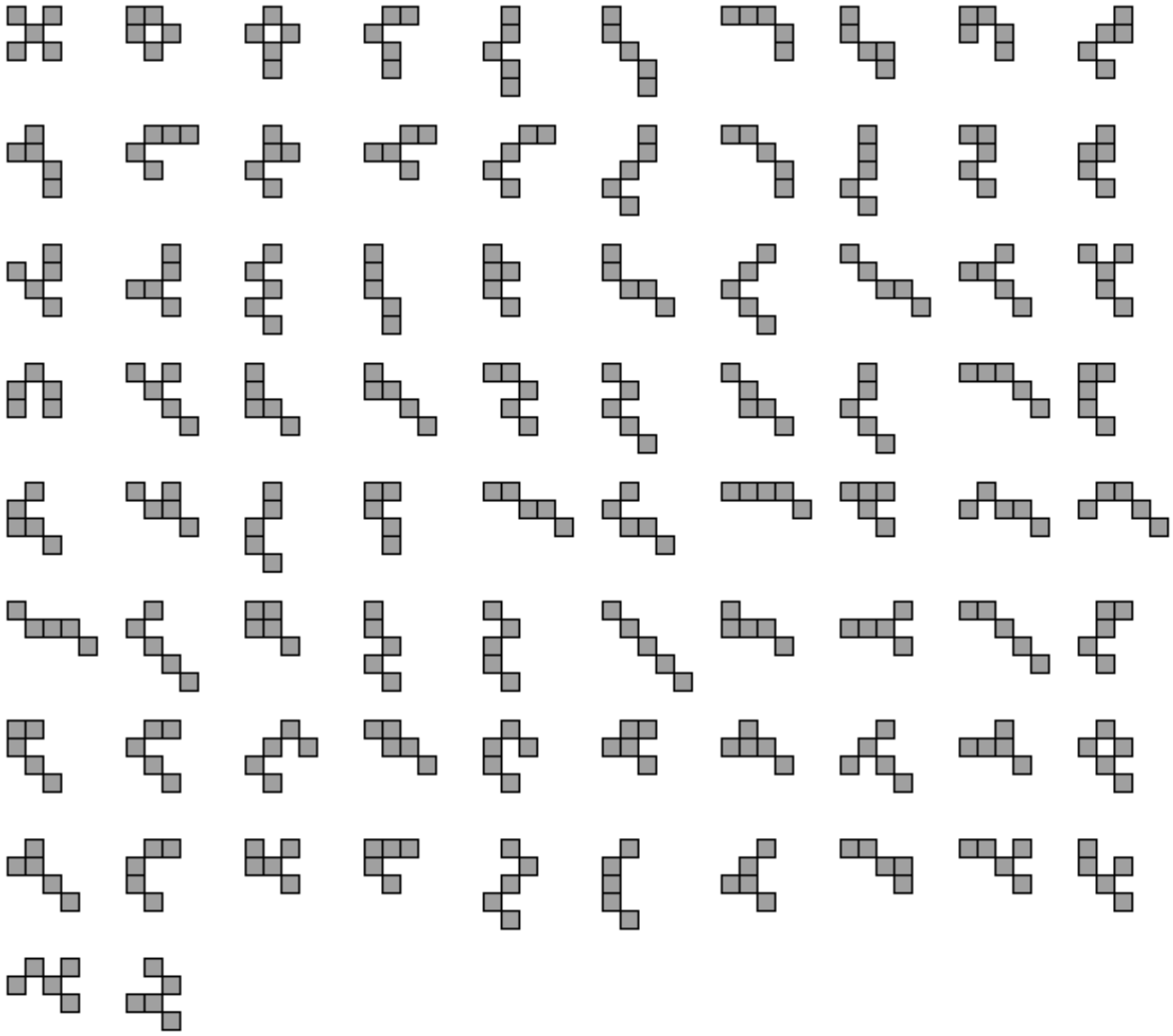


Abbildung 5.2: 5-Rominos

Alternativ zu dem bisher beschriebenen Ansatz ist es zum Beispiel möglich, n-Rominos aus $(n-1)$-Rominos zu generieren, indem an jede mögliche Position am Rand des $(n-1)$-Rominos ein Stein angefügt wird. Diese Methode ist etwas langsamer als die hier beschriebene, aber bei guter Implementierung ausreichend schnell, um alle Beispiele zu bearbeiten.

Es gibt noch deutlich schnellere Algorithmen, die nur die Anzahl von n-Polyominos berechnen. Die Aufgabenstellung verlangt allerdings, die Rominos auch zu generieren und auszugeben.

Der Bruteforce-Ansatz, für alle Kästchen in einem hinreichend großen (also $n \times n$) Feld jeweils zu entscheiden, ob sie Teil des n-Rominos sind, und erst dann zu prüfen, ob es sich insgesamt um ein n-Romino handelt, funktioniert höchstens bis $n = 5$ und ist daher zu langsam.

5.6 Bewertungskriterien

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

- $[-1]$ **Lösungsverfahren findet nicht alle Rominos**
  Es müssen alle n-Rominos gefunden werden.
- $[-1]$ **Lösungsverfahren zeigt / zählt Duplikate**
  Es sollen keine Duplikate (m. a. W. Rotationen oder Spiegelungen von anderen n-Rominos) ausgegeben werden.
- $[-1]$ **Lösungsverfahren anderweitig fehlerhaft**
  Die ausgegebenen n-Rominos müssen regelkonform sein: Sie müssen also zusammenhängend sein, und es muss mindestens ein Paar von Quadraten geben, die aneinander nur mit einer Ecke angrenzen und die nicht beide eine gemeinsame Kante mit ein und demselben dritten Quadrat haben.
- $[-1]$ **Verfahren unnötig aufwendig / ineffizient**
  Das Programm sollte für $n \leq 10$ die n-Rominos in angemessener Zeit bestimmen können. Für $n > 8$ dürfen das aber durchaus einige Minuten sein. Eine vollständige Suche ist ausreichend, wenn der Suchraum nicht unnötig groß ist – was z. B. durch eine Normalisierung erreicht werden kann.
- $[-1]$ **Ausgabe der n-Rominos unzureichend**
  Es muss eine Form von grafischer Ausgabe geben. Ein GUI ist schön, aber nicht zwingend notwendig; eine Ausgabe in eine Bilddatei oder per übersichtlich angeordneter Textzeichen (ASCII Art) ist auch akzeptabel. Die Ausgabe eines n-Rominos als Liste von Koordinaten der Quadrate ist nicht ausreichend.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Für $n = 4$ müssen alle, für $n = 5$ soll ein Teil der n-Rominos dokumentiert werden. Für $n \in \{4, \ldots, 10\}$ muss außerdem die Anzahl der n-Rominos angegeben werden.

Anhang: Perlen der Informatik aus den Einsendungen

Allgemeines

Hier sind die Lösungen des Programms der Aufgaben, die auf der Website zu finden sind. *Die restliche Seite ist leer.* **Schlusswort**

Es bietet sich an, diese Aufgabe objektorientiert zu lösen. Also hatte ich zuerst die Objekte [...] erstellt, ohne einen Lösungsansatz zu haben.

Leider ist das relativ schwer zu erklären, das ist bei rekursiven Methoden fast immer so.

[...], worauf danach die praktische Implementierung und Exekution folgt.

Aus den Quellcodes:

```
public class BruteForce extends Blumenbeet {
```

```
Out-of-bounce Error
```

```
case ‘‘grün’’: [...];
case ‘‘gruen’’: [...];
case ‘‘grã¼n’’: [...]
```

Schleifenzähler: `DerWieVielteDurchlaufDerForSchleife` *– (Wir hoffen, die IDE hatte eine Autovervollständigung für Variablennamen.)*

Teamnamen:

Stackoverflow Masters

Junioraufgabe 1: Parallelen

Eigentlich ganz einfach, könnte man meinen [...]

Danach wird in der Methode geDicht() das Gedicht eingelesen [...]

Aufgabenstellung: Zähle die Anzahl *n* der Buchstaben in diesem Wort [...]. *Teilnehmer: zählt die Anzahl der „n“s in den Wörtern.*

Junioraufgabe 2: Kacheln

[...] die sechs vorgegebenen Beispiele, die ich auf Grund von Platzmangel nicht auflisten werde. *– (Es folgen 13 Seiten Quellcode.)*

Diese Felder bestimmen wir, um in unserer Klimakrise mehr Land zu retten, natürlich als festen Boden.

Aufgabe 1: Blumenbeet

4.5 Sekunden [Laufzeit] sind nicht optimal, aber in der Zeit kann der Blumenverkäufer ja noch ein paar nette Worte mit dem Kunden wechseln :)

Aufgabe 3: Telepaartie

Eingabeaufforderung: Wenn sie die LLL möchten drücken sie die 1; falls sie L(n) möchten drücken sie die 2.

Aufgabe 4: Urlaubsfahrt

Verrauch des Fahrzeugs

die Tanggröße

Das Ziel ist eine kostenlose Tankstelle [...]

Ausgabe zur Lösung nach der Ausgabe der Kosten und unabhängig davon, wie hoch diese sind: Also fahren Sie lieber mit dem Zug. Spart Geld und schont das Klima.

Bei der Urlaubsfahrt handelt es sich um eine Europareise, die durch eine lange Autofahrt gebrandmarkt ist.

Aufgabe 5: Rominos

Ermittlung der möglichen zusammenhängenden „Rombinationen“ aus *n* Blöcken

[...] dass die Figur korrekt und ein Unikat ist.