



Lösungshinweise und Bewertungskriterien

Allgemeines

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

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

Dokumentation

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

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

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

Zu einer Einsendung gehören als zweiter Teil der Implementierung *Programme*, die möglichst eigenständig lauffähig sind. Für die gängigsten Skript-Sprachen stehen Interpreter zur Verfügung. Einige Entwicklungsumgebungen (z. B. BlueJ), bei denen die erstellten Programme ohne Weiteres nur in der Umgebung selbst laufen, stehen bei der Bewertung zur Verfügung, aber sicher nicht alle. Kompilierung von Quellcode ist während der Bewertung nicht möglich. Deshalb können Programme nur dann mit unterschiedlichen Eingaben getestet werden, wenn diese

vom Programm eingelesen oder über eine (nicht zwingend grafische) Schnittstelle eingegeben werden können.

Auch die folgenden eher inhaltlichen Dinge sind zu beachten:

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

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

Bewertung

Nun einige Erläuterungen zur Bewertung:

- Pro Aufgabe werden maximal fünf Punkte vergeben. Zu jeder Aufgabe gibt es eine Reihe von Bewertungskriterien. Sie sind in der Bewertung, die man im PMS einsehen kann, aufgelistet. In der ersten Runde gibt es in der Regel nur Punktabzüge; deshalb sind die Kriterien meist negativ formuliert. Hat man bei einem Kriterium 0 Punkte, ist die Einsendung in Bezug auf dieses Kriterium einwandfrei. Wenn das Kriterium nicht erfüllt ist, gibt es in der Regel einen Punkt Abzug (-1), manchmal auch zwei. Ist die Bearbeitung insgesamt unzureichend, wird ein spezielles Kriterium angewandt, bei dem es bis zu fünf Punkte Abzug geben kann. Im schlechtesten Fall wird die Aufgabenbearbeitung mit null Punkten bewertet.
- In der Hauptliga sind für die Gesamtpunktzahl die drei am besten bewerteten Aufgabenbearbeitungen maßgeblich. Es lassen sich also maximal 15 Punkte erreichen. Einen 1. Preis erreicht man mit 14 oder 15 Punkten, einen 2. Preis mit 12 oder 13 Punkten und einen 3. Preis mit 9 bis 11 Punkten. Mit einem 1. oder 2. Preis ist man für die zweite Runde qualifiziert.
- In der Juniorliga wird ein 1. Preis für 9 oder 10 Punkte, ein 2. Preis für 7 oder 8 Punkte und ein 3. Preis für 5 oder 6 Punkte vergeben. Leider gibt es in der Juniorliga (noch) keine zweite Runde.
- Leider wurde in einigen Fällen die Regelung zur Bearbeitung von Junioraufgaben nicht beachtet. Zitat aus dem Mantelbogen des Aufgabenblatts: „Ausgeschlossen von der Bearbeitung der Junioraufgaben sind Schülerinnen und Schüler aus der Qualifikationsphase der gymnasialen Oberstufe sowie aus der Berufsschule.“ Nur wenn ein Team mindestens ein Mitglied hat, das Junioraufgaben bearbeiten darf, darf dessen Einsendung auch Bearbeitungen von Junioraufgaben enthalten.

- Eine Einsendung wird in der Juniorliga gewertet, wenn alle Gruppenmitglieder die Bedingung für Junioraufgaben erfüllen (damit kommt die Einsendung für die Juniorliga in Frage) und in der Einsendung Junioraufgaben bearbeitet wurden. Eine solche Einsendung wird zusätzlich in der Hauptliga gewertet, wenn auch „normale“ Aufgaben bearbeitet wurden.
- Grundsätzlich kann die Dokumentation als „schlecht / nicht vollständig“ bewertet werden, wenn Teile wie Umsetzung, Beispiele oder Quellcode(auszüge) fehlen. Außerdem wird in der Regel gefordert, dass die gewählten Verfahren gut nachvollziehbar beschrieben sind und dass begründet wird, warum sie das gegebene Problem lösen.
- Leider ließ sich nicht verhindern, dass etliche Teilnehmer nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden. Das ist ziemlich riskant, da Fehler sich leicht einschleichen.
- Es ist verständlich, wenn jemand, der nicht weitergekommen ist, über eine Reklamation nachdenkt. Kritische Fälle, insbesondere die mit 11 Punkten, haben wir allerdings schon sehr gründlich und mit viel Wohlwollen geprüft.

Danksagung

An der Erstellung der Lösungsideen haben mitgewirkt: Friedrich Hübner und Thekla Hamm (Junioraufgabe 1), Martin Thoma (Junioraufgabe 2), Nikolai Wyderka (Aufgabe 1), Karl Schrader (Aufgabe 2), Philip Wellnitz (Aufgabe 3), Florian Behrens (Aufgabe 4) und Lukas Michel (Aufgabe 5).

Die Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt, und zwar aus Vorschlägen von Torben Hagerup (Junioraufgabe 1 und Aufgabe 3), Holger Schlingloff (Junioraufgabe 2, Aufgabe 1 und Aufgabe 2), Wolfgang Pohl (Aufgabe 4) und Jens Gallenbacher (Aufgabe 5).

J1 Landnahme

J1.1 Festlegungen zur Eingabe und deren Verarbeitung

Für die Lösung dieser Aufgabe gibt es verschiedene Strategien. Welche man verwenden kann, hängt auch davon ab, mit welchen Eingaben man zurecht kommen muss, und zu welcher Zeit man auf einzelne Eingaben reagiert. Zunächst zur Eingabe: Die besteht laut Aufgabenstellung aus einer Liste von Rechtecken. Ein Rechteck ist durch vier Zahlen gegeben: die Koordinaten zweier sich diagonal gegenüberliegender Ecken. Bezüglich dieser Zahlen sind zwei Punkte nicht genau festgelegt, nämlich das Zahlenformat und der Zahlenbereich, so dass man hier mehrere Möglichkeiten hat:

Zahlenformat Auch wenn die Beispieleingabe nur natürliche Zahlen enthält, schließt die Aufgabenstellung nicht ausdrücklich aus, dass die Koordinaten der Punkte auch negative (ganze) Zahlen oder Fließkommazahlen sein können. Man muss also selbst einen sinnvollen Datentyp festlegen. Je nachdem verwendet man entweder einen Datentyp für ganze Zahlen, wie *int* oder *long*, oder einen für Fließkommazahlen, wie *float* oder *double*.¹

Zahlenbereich Wichtig ist auch, dass man eine Obergrenze für die Größe der Zahlen festlegt. Verwendet man z.B. eine 32-Bit-Ganzzahl (*int*), so dürfen die Koordinaten nicht größer sein als $2^{31} - 1 = 2147483647$. Erwartet man auch größere Koordinaten, so sollte man auf 64-Bit-Ganzzahlen (*long*) zurückgreifen. Man kann sich aber überlegen, dass auch im wilden Westen Entfernungen nicht in Millimeter angegeben wurden, oder dass niemand mehrere 1000km in 24 Stunden zu Fuß schaffen kann. Einem alten Commodore reicht also auch ein 16-Bit-Ganzzahl, die immerhin Koordinaten bis $2^{15} - 1 = 32768$ speichern kann.

Online oder nicht Ein dritter offener Punkt ist, wie der Commodore auf die Eingabe reagieren muss. Es ist nicht angegeben, ob er sofort nach der Umrundung des Rechteckes entscheiden muss, ob es genehmigt ist, oder ob er erst entscheidet, wenn er alle Rechtecke kennt. Wählt man eine Strategie, mit welcher der Commodore sofort nach jedem Rechteck entscheidet, so spricht man von einem Online-Algorithmus. Für andere Lösungsstrategien wiederum ist es wichtig, dass man alle Rechtecke schon zu Beginn kennt, da das Vorberechnungen ermöglicht. Online oder nicht – beides ist vertretbar.

J1.2 Lösungsstrategien

Prinzipiell muss man die Rechtecke in der Reihenfolge abarbeiten, in der sie in der Eingabe stehen. Man kann nur entscheiden, ob ein Rechteck genehmigt wird oder nicht, wenn diese Frage schon für alle Rechtecke davor entschieden wurde. Daher ist die grundlegende Idee, sich die Menge der schon genehmigten Rechtecke zu speichern und dann immer das nächste Rechteck zu überprüfen. Algorithmus 1 drückt diese grundlegende Verfahrensweise in Pseudocode aus.

¹Die Namen von Datentypen variieren je nach Programmiersprache. In einigen Programmiersprachen, wie z.B. Python, muss kein Datentyp angegeben werden.

Algorithmus 1 Arbeitsanweisung für den alten Commodore

function LANDNAHME
 $M \leftarrow \emptyset$

▷ Menge der genehmigten Rechtecke; zu Beginn leer

for Rechteck r **do** **if** r überschneidet sich mit einem Rechteck aus M **then**

PRINT('abgelehnt')

else

PRINT('genehmigt')

 füge r zu M hinzu **end if****end for****end function**

Interessant ist nun, wie man die Menge M speichert und wie man überprüft, ob sich ein Rechteck mit einem anderen überschneidet. Dazu werden im Folgenden zwei Varianten vorgestellt.

Variante 1: Online-Algorithmus für beliebige Zahlen

Man speichert die Rechtecke mit ihren Koordinaten in einem Array, in einer verketteten Liste oder einer ähnlichen Datenstruktur. Aus den gegebenen Koordinaten eines Rechtecks gehen die minimale und maximale x-Koordinate x_{min} und x_{max} bzw. die minimale und die maximale y-Koordinate y_{min} und y_{max} des Rechtecks direkt hervor. Damit lassen sich die Koordinaten aller Eckpunkte eines Rechtecks problemlos konstruieren.

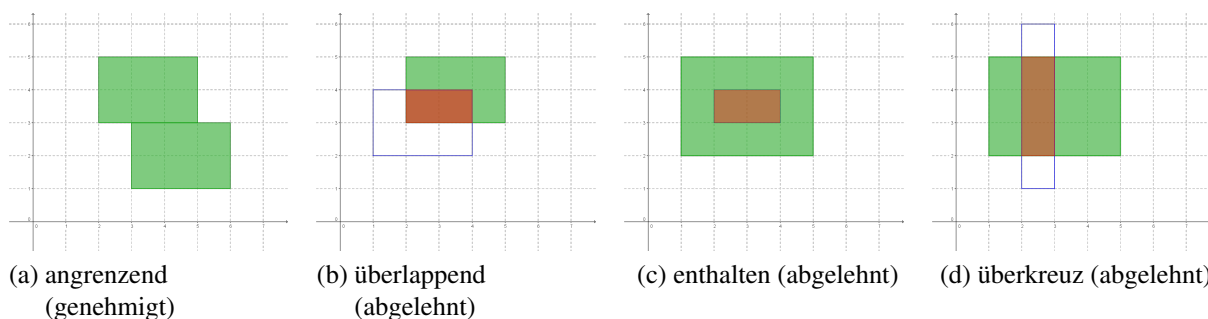


Abbildung 1: Wichtige Fälle bei der Überschneidungsprüfung

Um zu ermitteln, ob ein neu beantragtes Rechteck sich mit einem schon genehmigten überschneidet, überprüft man für jedes bereits genehmigte Rechteck einzeln, ob das beantragte sich mit ihm überschneidet. Eine solche Prüfung muss alle Fälle korrekt behandeln, insbesondere auch die in Abb. 1 gezeigten. Eine naheliegende Idee, nämlich die Überprüfung, ob mindestens ein Eckpunkt des beantragten Rechtecks im bereits genehmigten Rechteck enthalten ist, scheidet am Fall „überkreuz“ und am Umkehrfall von „enthalten“ (wenn das genehmigte Rechteck im beantragten enthalten ist).

Ein sicheres Kriterium ist hingegen das folgende: Ein Rechteck r_1 überschneidet sich dann nicht mit einem Rechteck r_2 , wenn es links, rechts, oben oder unten von r_2 liegt. Das bedeutet z.B., dass die rechte Seite von r_1 links von der linken Seite von r_2 liegen könnte, also: $r_1.x_{max} <$

$r_2 \cdot x_{min}$. Betrachtet man alle Seiten, dann kommt man darauf, dass sich zwei Rechtecke genau dann nicht überschneiden, wenn gilt (\vee ist das logische „oder“):

$$(r_1 \cdot x_{max} \leq r_2 \cdot x_{min}) \vee (r_1 \cdot x_{min} \geq r_2 \cdot x_{max}) \vee (r_1 \cdot y_{max} \leq r_2 \cdot y_{min}) \vee (r_1 \cdot y_{min} \geq r_2 \cdot y_{max}) \quad (1)$$

Indem man das Ergebnis dann noch logisch negiert, kann man ermitteln, ob sich die beiden Rechtecke schneiden. Statt mit den Koordinaten der Eckpunkte kann man auch mit Mittelpunkten und Seitenlängen rechnen, um zu klären, ob zwei Rechtecke ohne Überlappung neben- bzw. übereinander liegen.

Der Algorithmus ist online, da man keine Informationen über spätere Rechtecke benötigt. Weiterhin ist es egal, welches Zahlenformat man verwendet, da man nur Zahlen vergleichen muss. Zur Laufzeit kann man sich Folgendes überlegen: Im schlimmstem Fall muss man jedes Rechteck mit jedem anderen vergleichen, also bei n Rechtecken $0 + 1 + 2 + \dots + (n - 1) = \frac{n(n - 1)}{2} \in O(n^2)$ Vergleiche ausführen². Bei einer Siedlerschwemme könnte der alte Commodore mit diesem Verfahren also durchaus ins Schwitzen kommen: Bei 1000 Rechtecken stehen etwa 1 Million Vergleiche, bei 1 Million Rechtecke etwa 1 Billion Vergleiche an.

Da man die Koordinaten nur miteinander vergleicht, ist es für die Betrachtung der Laufzeit egal, welche Werte die Koordinaten haben. Diese Lösung eignet sich also für eine nicht allzu große Zahl von Rechtecken, deren Koordinaten dafür beliebig geartet und beliebig groß sein dürfen.

Variante 2: Für kleinere, ganzzahlige Koordinaten

Wenn man davon ausgeht, dass die Koordinaten nur natürliche Zahlen sind, so kann auch folgende Variante sehr effektiv sein. An Stelle einer Liste von Rechtecken speichert man eine Landkarte. Diese Landkarte unterteilt man in Quadrate der Größe 1×1 und speichert für jedes Quadrat, ob es Teil eines Rechteckes ist. Das geht nur, wenn die Eingabe nur aus natürlichen Zahlen besteht³. Die Landkarte kann man dann zum Beispiel als zweidimensionales Array A speichern. Dabei ist der Eintrag $A[x][y]$ gleich 1, wenn das Einheitsquadrat an der Position (x, y) schon in einem Rechteck enthalten ist, und 0 sonst. Abb. 2 zeigt ein Beispiel für diese Umsetzung.

Da man vor dem Anlegen eines Arrays schon die benötigte Größe wissen muss, kann man diesen Algorithmus nur verwenden, wenn man weiß, welche Koordinaten in der Eingabe auftreten werden. Dazu müsste man theoretisch alle Rechtecke von Beginn an kennen, so dass der Algorithmus nicht online ist. Man kann aber spezielle Datenstrukturen wie dynamische Arrays verwenden, die man bei Bedarf in ihrer Größe verändern kann, um aus dem Algorithmus doch wieder einen Online-Algorithmus zu machen. Es ist aber – gerade für eine Juniorkaufgabe – auch in Ordnung, selbst eine Obergrenze für die Koordinaten festzulegen; dann funktioniert der Algorithmus auch bei einem Array fester Größe online.

Will man ermitteln, ob ein neues Rechteck genehmigt wird oder nicht, geht man alle Einheitsquadrate durch, die in dem neuen Rechteck liegen. Ist der Eintrag eines dieser Einheitsquadrate gleich 1, also das Quadrat schon belegt, so wird das Rechteck nicht genehmigt. Ansonsten wird

²Die sogenannte O-Notation $O(f(n))$ gibt das charakteristische Verhalten der Funktion $f(n)$ an.

³Durch eine sogenannte Koordinatenkompression kann man auch Fließkommazahlen verwenden. Dabei verwendet man anstatt von Einheitsquadraten Rechtecke unterschiedlicher Größe. Allerdings ist dies deutlich komplizierter zu implementieren.

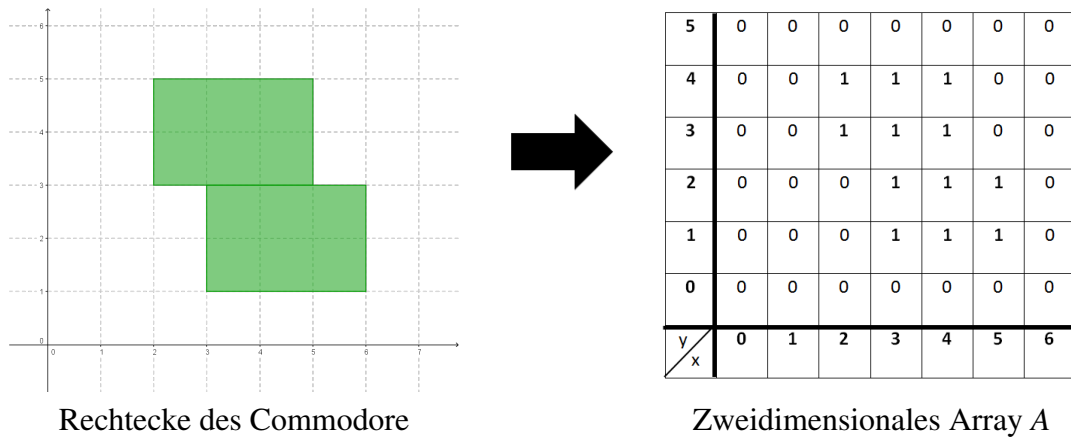


Abbildung 2: Repräsentation von Rechtecken in einem Array

das Rechteck genehmigt, und die Einträge der dadurch (neu) belegten Einheitsquadrate werden auf 1 gesetzt.

Das Problem bei dieser Variante ist, dass die Laufzeit abhängig ist von der Größe der Rechtecke. Um zu ermitteln, ob ein Rechteck mit Breite w und Höhe h genehmigt wird oder nicht, muss man $w \cdot h$ Einheitsquadrate überprüfen. Die Laufzeit ist also nur schwer abschätzbar und kann bei sehr großen Rechtecken sehr hoch sein.⁴

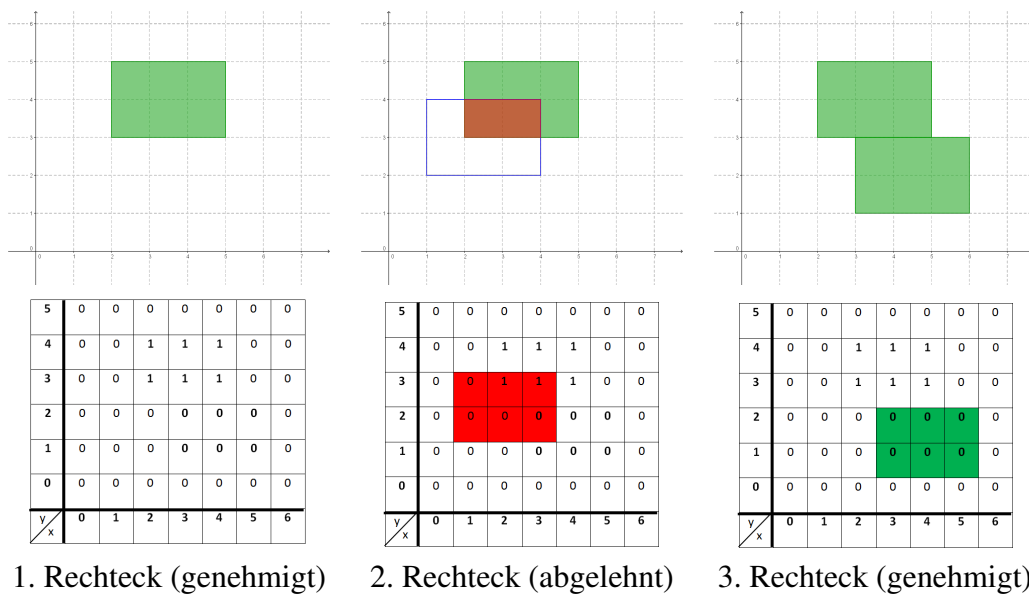


Abbildung 3: Abarbeitung des Beispiels aus der Aufgabenstellung

⁴Es gibt (äußerst komplizierte) Datenstrukturen, mit denen das Überprüfen und Hinzufügen eines Rechteckes in logarithmischer Laufzeit $O(\log n)$ möglich ist, unabhängig von der Größe der Rechtecke. Eine solche Datenstruktur ist z.B. ein Segmentbaum. Damit kann man insgesamt eine Laufzeit von $O(n \log n)$ erreichen, die sogar besser ist als Variante 1 mit $O(n^2)$.

J1.3 Beispiel

Im Folgenden wird das Vorgehen der beiden vorgestellten Varianten anhand der in der Aufgabenstellung angegebenen Beispieleingabe erläutert (vgl. Abb. 3). Zu Beginn gibt es kein auf Überschneidung zu prüfendes Rechteck, bzw. keine 1-er-Einträge für Einheitsquadrate auf der Landkarte. Daher wird in beiden Varianten das erste Rechteck genehmigt.

Sei r_1 das erste und r_2 das zweite Rechteck, mit $r_1.x_{min} = 2, r_1.x_{max} = 5, r_1.y_{min} = 3, r_1.y_{max} = 5$ und $r_2.x_{min} = 1, r_2.x_{max} = 4, r_2.y_{min} = 2, r_2.y_{max} = 4$. Wenn man in Variante 1 nach Formel 1 überprüft, ob sich beide Rechtecke nicht schneiden, dann erhält man:

$$(5 \leq 1) \vee (2 \geq 4) \vee (5 \leq 2) \vee (3 \geq 4) = \text{false}$$

Demzufolge schneiden sich beide Rechtecke. Analog sind bei Variante 2 die beiden Einträge A[2][3] und A[3][3] schon 1. Daher wird in beiden Varianten das zweite Rechteck abgelehnt.

Das dritte beantragte Rechteck muss in Variante 1 mit dem ersten Rechteck abgeglichen werden. Formel 1 ergibt für die Koordinaten dieser beiden Rechtecke

$$(6 \leq 2) \vee (3 \geq 5) \vee (3 \leq 3) \vee (1 \geq 5) = \text{true}$$

Die Rechtecke überschneiden sich also nicht. In Variante 2 wiederum sind alle Einheitsquadrate des dritten Rechtecks noch frei. Das dritte Rechteck wird also in beiden Varianten genehmigt.

Die Ausgabe zu diesem Beispiel sollte dann, wie in der Aufgabenstellung angegeben, folgendermaßen aussehen:

```
2 3 5 5 genehmigt
1 2 4 4 abgelehnt
3 1 6 3 genehmigt
```

J1.4 Bewertungskriterien

- Ausdrückliche Überlegungen zur Eingabe und zur Frage nach der Online-Abarbeitung der Eingabe, wie sie in diesen Lösungshinweisen angestellt werden, werden in der Bearbeitung einer Junioraufgabe nicht erwartet.
- Die Entscheidung über Genehmigung oder Ablehnung eines Grundstücks darf nur auf der Grundlage vorher genehmigter Grundstücke fallen; bereits abgelehnte Grundstücke oder in der Eingabe erst folgende Grundstücke dürfen keine Rolle spielen.
- Der Kern der Lösung ist die Prüfung zweier Rechtecke auf Überschneidung bzw. Nicht-Überschneidung. Sie soll grundsätzlich korrekt funktionieren, auch für speziellere Fälle. Falsche Ergebnisse in Einzelfällen, z.B. auf Grund von Fehlern in der Implementierung, sollen ebenfalls nicht vorkommen.
- Bewusst wurden bei dieser Aufgabe keine Pflichtbeispiele vorgegeben, um bei der Betrachtung der möglichen Fälle nicht zu viele Hinweise zu geben. Es ist aber eine grundsätzliche Anforderung an jede Dokumentation, dass das Funktionieren des Verfahrens anhand einiger geeigneter Beispiele dokumentiert wird.
- Falls die freiwillige Zusatzaufgabe (Visualisierung der Landnahme) einigermaßen brauchbar bearbeitet wurde, soll das in der Bewertung vermerkt sein, gibt aber keine Pluspunkte.

J2 Kassiopeia

Die Schildkröte Kassiopeia will wissen, ob sie in Quadraten jedes weiße Feld erreichen kann.

Dazu sollte zuerst bemerkt werden, dass für die Antwort unerheblich ist, wo die Schildkröte sich befindet. In dieser Aufgabe darf sie sowohl jedes Feld als auch jeden Feldübergang beliebig häufig nutzen. Wenn sie also von ihrem Feld K aus ein Feld F nicht erreichen kann, dann könnte sie von Feld F aus auch nicht Feld K erreichen. Im Folgenden ignorieren wir also die Information, wo K ist. Dieses Feld wird wie jedes andere weiße Feld behandelt.

J2.1 Der direkte Weg

Um diese Aufgabe zu lösen, muss zunächst die in der Eingabe enthaltene Landkarte von Quadraten gespeichert werden. Dafür kann man einen Array nehmen, dessen Größe in der ersten Eingabezeile vorgegeben wird. In die Zeilen des Arrays speichert man dann die Inhalte der Eingabezeilen, und zwar ein Eingabezeichen pro Array-Element. Auch andere Datenstrukturen sind möglich. Wichtig ist, dass man für ein Landkartenfeld leicht die Elemente in der Datenstruktur bestimmen kann, die den oben, unten, rechts bzw. links gelegenen Feldern entsprechen.

Außerdem benötigt man eine Liste – oder irgendeine andere Datenstruktur, in der mehrere Elemente gespeichert werden können, z. B. Mengen, Stacks oder Warteschlangen (englisch *queue*). Wir verwenden eine Liste und speichern darin, von wo aus Kassiopeia noch weitergehen kann. Diese Liste nennen wir nun L_W . Zu Beginn enthält L_W nur das Startfeld.

Von jedem Feld aus kann Kassiopeia theoretisch einen Schritt nach oben, unten, rechts oder links machen. Praktisch geht das natürlich nur, wenn das jeweilige Nachbarfeld weiß ist. Wenn ein Nachbarfeld eines aktuell untersuchten Feldes also weiß ist, wird es in L_W gesteckt. Wenn wir die vier Nachbarfelder überprüft haben, dann sind wir mit dem aktuellen Feld fertig. Damit wir das Feld nicht nochmals besuchen, färben wir es ein. Natürlich müssen wir das aktuell untersuchte Feld auch aus L_W entfernen, da wir ja schon alle Möglichkeiten betrachtet haben, wie man von dort aus weitergehen kann. Dann untersuchen wir das nächste Feld in L_W , und so weiter.

Wenn L_W leer ist, hat Kassiopeia alle Felder besucht, die sie besuchen kann. Nun müssen wir überprüfen, ob das tatsächlich alle Felder sind. Dazu schauen wir uns alle Felder an. Wenn ein Feld noch weiß ist, dann kann Kassiopeia dieses Feld nicht erreichen.

J2.2 Repräsentation der Landkarte als Graph

Die Beispieldaten liegen in dem Format vor, das in Abb. 4a dargestellt ist. Dabei steht # für ein schwarzes Feld, und \mathbb{K} sowie Leerzeichen stehen für weiße Felder. Die erste Zeile der Eingabe gibt die Höhe bzw. Breite von Quadraten an.

Wie oben beschrieben, kann die analoge Repräsentation dieser Landkarte in einem Array durchaus verwendet werden, um die Aufgabe zu lösen. Allerdings ist die Repräsentation als *Graph* üblicher und erlaubt die Lösung mit bekannten Methoden. *Graphen* sind mathematische Objekte, die aus einer *Knotenmenge* V und einer *Kantenmenge* E bestehen.⁵ Jede Kante stellt eine

⁵Auf Englisch heißen Knoten „vertices“ und Kanten „edges“.

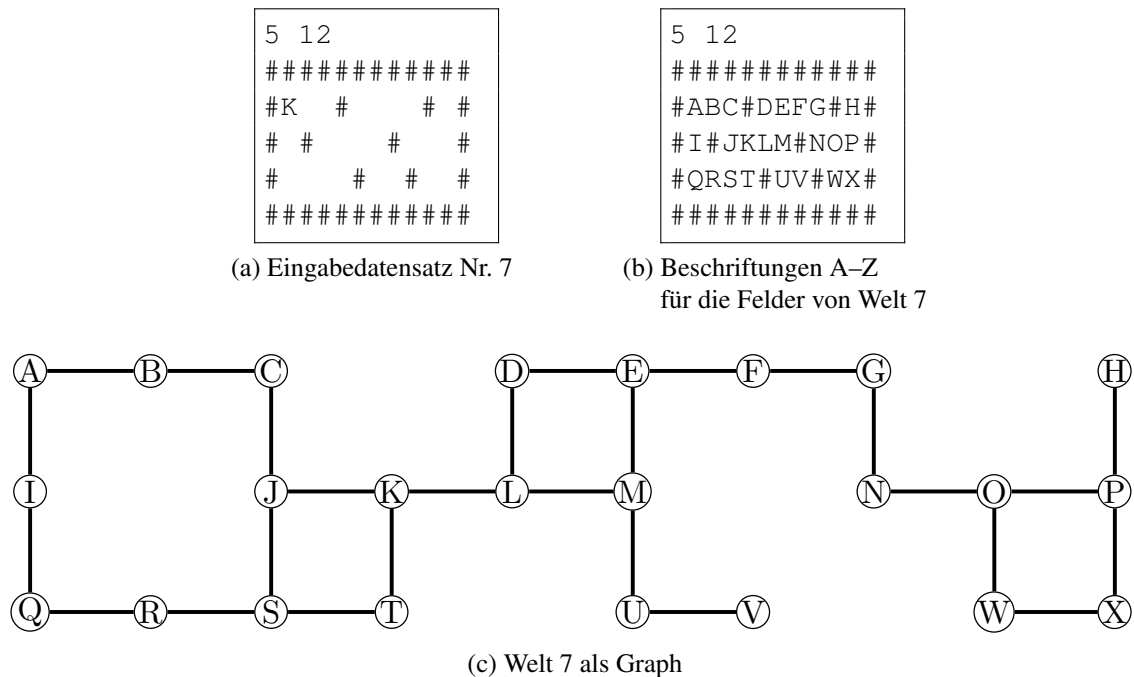


Abbildung 4: Repräsentation einer Landkarte als Graph

Beziehung zwischen zwei Knoten aus V her. In unserem Beispiel kann man jedes weiße Feld als einen Knoten sehen. Dann sind zwei Knoten durch eine Kante verbunden, wenn sie benachbarten Feldern entsprechen. Wir tun also so, als ob schwarze Felder einfach gar nicht vorhanden wären. Eine typische Art einen Graphen bildlich darzustellen ist in Abb. 4c zu sehen.

Einen Graph nennt man *zusammenhängend*, wenn man von jedem Knoten aus jeden anderen über Kanten und andere Knoten erreichen kann. Kassiopia kann also ein Feld nicht erreichen, wenn der Graph nicht zusammenhängend ist.

Doch bevor wir das algorithmisch bestimmen, müssen wir den Graphen als Datenstruktur repräsentieren. Dazu gibt es zwei verbreitete Möglichkeiten, die beide ihre Stärken haben: Die *Adjazenzmatrix* und die *Adjazenzliste*. Eine Adjazenzmatrix ist eine Tabelle aus Nullen und Einsen. Eine Eins in Zeile i und Spalte j bedeutet, dass man von Knoten i zu Knoten j kommt. Da es auf Quadraten keine Einbahnstraßen gibt, muss diese Tabelle (bzw. *Matrix*) symmetrisch sein. Wenn ich also von i nach j komme, dann auch von j nach i .

Um die Matrix für den Graphen der weißen Felder aufzubauen, kann man zuerst den kompletten Gittergraphen als Matrix abspeichern. Es wird also ein Graph mit (Breite von Quadraten) · (Höhe von Quadraten) Knoten erstellt, wobei die inneren Knoten mit jeweils vier Nachbarn verbunden sind und die äußeren mit jeweils zwei Nachbarn. Sobald man diesen Graphen hat, löscht man die Knoten heraus, die schwarze Felder repräsentieren. Dies macht man, indem die jeweilige Spalte und Zeile des Knotens aus der Matrix entfernt werden. Um Fehler bei der Verschiebung der Indizes zu vermeiden empfiehlt es sich, die zu löschenden Felder einmal zu berechnen, nach ihrem Index absteigend zu sortieren und dann in dieser Reihenfolge zu entfernen.

Eine Adjazenzliste ist eine Liste von Listen. Jede Teilliste steht für einen Knoten. Die Liste des Knotens i beinhaltet die Namen aller Knoten, zu denen man von i aus kommen kann.

Beispiel	0	1	2	3	4	5	6	7
Zusammenhängend	Ja	Nein	Ja	Ja	Ja	Ja	Ja	Ja

Tabelle 1: Ergebnisse für die Beispieldaten

J2.3 Prüfung auf Zusammenhang

Um zu überprüfen, ob ein Graph zusammenhängend ist, gibt es zahlreiche Algorithmen. Die beiden wohl einfachsten und nützlichsten sind die *Breitensuche* bzw. *Tiefensuche*. Die Idee ist dabei, den Graphen zielgerichtet abzulaufen (bzw. zu *traversieren*, wie man in der Graphentheorie sagt).

Algorithmus 2 Graphen traversieren

```

function ISTZUSAMMENHAENGEND(Graph  $G = (V, E)$ )
   $ZuLaufen \leftarrow$  LISTE
   $start \leftarrow x \in V$  ▷ wähle zufällig einen Knoten aus
  HINZUFUEGEN( $ZuLaufen, start$ )
   $Gelaufen \leftarrow \{ start \}$ 
  while  $|ZuLaufen| > 0$  do
     $x \leftarrow$  ZIEHE( $ZuLaufen$ ) ▷ das erste Element, also von Vorne
     $Gelaufen \leftarrow Gelaufen \cup x$ 
    for Nachbar  $x_j \in$  NACHBARN( $x$ ) do
      if  $x_j \notin Gelaufen$  then
        HINZUFUEGEN( $ZuLaufen, x_j$ )
      end if
    end for
  end while
  return  $V \setminus Gelaufen = \emptyset$ 
end function

```

Sowohl bei der Breitensuche als auch bei der Tiefensuche hat man (a) eine Liste von Knoten, die man noch betrachten will, (b) eine Liste von Knoten, die bereits betrachtet wurden, sowie (c) den Graphen selbst. Algorithmus 2 beschreibt mit Pseudocode die grundsätzliche Vorgehensweise beim Traversieren eines Graphen. Je nachdem, wie man dabei die Funktion HINZUFUEGEN implementiert, wird daraus eine Breiten- oder Tiefensuche. Fügt man die Elemente vorne in die Liste ein, so ist es eine Tiefensuche; fügt man die Elemente hinten in die Liste ein, ist es eine Breitensuche. In beiden Varianten geht man alle Knoten durch, die man erreichen kann.

J2.4 Bewertungskriterien

- Bei einer Junioraufgabe ist eine Lösung auf Basis von Graphen natürlich nicht gefordert. Alle funktionierenden Vorgehensweisen sind akzeptabel.
- Das Verfahren sollte aber nicht unnötig aufwändig sein. Insbesondere ist entscheidend, die bereits besuchten Felder sauber zu kennzeichnen, um nicht unnötig viele Möglichkeiten prüfen zu müssen.

- Der Startpunkt von Kassiopeia ist für die Berechnung der Erreichbarkeit irrelevant. Diese Einsicht könnte man haben – aber das wird nicht erwartet. Für das Verfahren ist es in Ordnung, immer vom Startpunkt auszugehen.
- Die in der Aufgabenstellung vorgegebenen Bedingungen müssen beachtet sein:
 - Kassiopeia betritt nur weiße Felder.
 - Sie bewegt sich in einem Schritt auf ein horizontal oder vertikal (nicht diagonal!) benachbartes Feld.
 - Das mit κ markierte Feld ist auch ein weißes Feld.
- Das Verfahren und seine Implementierung sollten korrekte Ergebnisse liefern. Insbesondere sollen die Ergebnisse für die 7 Beispieldatensätze korrekt sein (vgl. Tabelle 1).
- Es sollen wenigstens so viele Beispiele (und die zugehörigen Ergebnisse) auch in der Dokumentation behandelt sein, dass das korrekte Funktionieren der Lösung klar wird.

Aufgabe 1: Kassiopias Weg

1.1 Einleitung

Nachdem in der zweiten Junioraufgabe herausgefunden wurde, ob die weißen Felder in Kassiopias Welt zusammenhängend sind, wollen wir nun bestimmen, ob es einen Weg gibt, der alle weißen Felder genau einmal besucht. Ganz neu ist diese Problemstellung nicht: In der Informatik ist diese Aufgabe als Hamilton-Pfad-Problem bekannt. Um das Problem zu formalisieren, fassen wir zunächst die Welt so als Graph auf, wie es in der Lösung zur Junioraufgabe 2 beschrieben ist. Jedem weißen Feld ordnen wir also einen Knoten zu. Zwei Knoten sind genau dann mit einer Kante verbunden, wenn ihre Felder benachbart sind. Den Knoten, auf dem Kassiopia startet, nennen wir v_0 .

Angenommen, es gäbe n weiße Felder; dann gibt es auch n Knoten. Ein Hamilton-Pfad ist ein Weg durch einen Graphen, der jeden Knoten genau einmal passiert – also gerade das, was wir für die weißen Felder von Quadraten suchen. Beim Hamilton-Pfad-Problem muss entschieden werden, ob es für einen Graphen einen Hamilton-Pfad gibt: ob ausgehend von v_0 ein Pfad der Länge n existiert (= eine Folge von Knoten $(v_0, v_1, \dots, v_{n-1})$, wobei $v_i \neq v_j$ für $0 \leq i < j < n$ und v_i und v_{i+1} für $0 \leq i < n - 1$ benachbart sind). Um diese umständliche Beschreibung etwas anschaulicher zu machen, betrachten wir die Beispiele in Abb. 5.

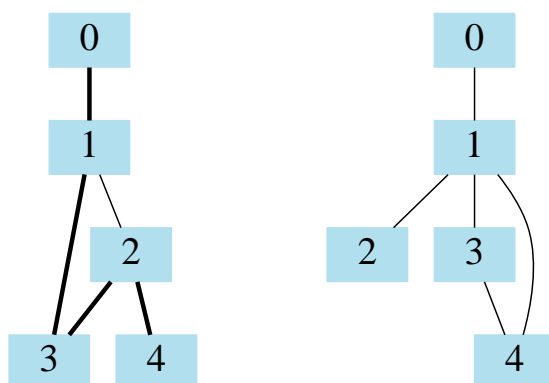


Abbildung 5: Hamilton-Pfad: zwei einfache Beispiele

Bei beiden Graphen sei der Startknoten $v_0 = 0$. Im linken Graphen gibt es einen Hamilton-Pfad, gegeben durch die Knotenfolge $(0, 1, 3, 2, 4)$. Im rechten Graphen gibt es keinen Hamilton-Pfad, weil in jedem Weg durch alle Knoten der Knoten 1 doppelt besucht werden müsste.

In der Welt Kassiopias haben wir es mit einem Spezialfall des allgemeinen Hamilton-Pfad-Problems zu tun, weil unser Graph von einer flachen, zweidimensionalen Welt herrührt, bei der jedes Feld maximal vier Nachbarfelder hat. Nichtsdestotrotz gehört es zu den schwierigsten Problemen der Informatik, da es ein sogenanntes NP-vollständiges Problem ist. Das heißt, dass es höchstwahrscheinlich keinen wesentlich besseren Algorithmus gibt als das Ausprobieren aller möglichen Wege, also sogenanntes Brute Force. Eine systematische Variante des Brute Force ist das Backtracking, bei dem ausgehend vom Startknoten rekursiv alle Wege durchprobiert werden. Auch dieses Verfahren ist laufzeittechnisch sehr aufwändig.

Ein bisschen besser geht es aber dennoch. Zunächst schauen wir uns jedoch die Backtracking-Lösung an.

1.2 Backtracking

Mit der folgenden rekursiven Methode kann man feststellen, ob ein Hamilton-Pfad von dem Knoten v_0 aus existiert: Starte bei Knoten v_0 und markiere ihn als besucht (also sozusagen als schwarz). Probiere nacheinander für jeden Nachbarknoten v_1 von v_0 aus, ob ein Hamilton-Pfad von v_1 aus alle noch nicht besuchten weißen Felder erreichen kann. Gibt es einen Nachbarn, bei dem das möglich ist, so gib „wahr“ zurück, wenn nicht, dann „falsch“. Sonderfall: Entspricht die Länge des so erzeugten Pfades der Anzahl von weißen Knoten, also n , so gib „wahr“ zurück. Als Parameter benötigt diese rekursive Methode den Graphen, den aktuellen Knoten und die Rekursionstiefe, mit deren Hilfe festgestellt werden kann, ob der Pfad aus n Knoten besteht. In Python-Pseudocode sieht das so aus:

```

1 def hasHamiltonPath(graph, node, depth):
2     if depth == graph.n:
3         return True
4
5     node.visited = True
6     for v in node.neighbours:
7         if not v.visited:
8             if hasHamiltonPath(graph, v, depth+1):
9                 return True
10
11     node.visited = False
12
13     return False

```

Natürlich möchte man nicht nur wissen, ob ein Pfad existiert, sondern auch, wie der Pfad aussieht. Dazu könnte man die Methode so abändern, dass sie eine Liste mit den Knoten zurück gibt, die zum Pfad gehören, oder None bzw. Null, wenn es keinen Pfad gibt.

```

1 def hasHamiltonPath(graph, node, depth):
2     if depth == graph.n:
3         return [node]
4
5     node.visited = True
6     for v in node.neighbours:
7         if not v.visited:
8             path = hasHamiltonPath(graph, v, depth+1)
9             if path != None:
10                return [node] + path
11
12     node.visited = False
13
14     return None

```

Man kann sich den Ablauf des Algorithmus mithilfe eines Aufrufbaumes verdeutlichen. Dabei wird die Wurzel mit v_0 markiert, und für jeden Unteraufruf mit Knoten v wird ein Kind erzeugt, das mit dem entsprechenden Knoten beschriftet wird. Die Laufzeit des Algorithmus ist durch $O(3^n)$ beschränkt, weil für jedes Feld (mit Ausnahme des ersten Feldes) maximal für drei Nachbarn ein rekursiver Unteraufruf gestartet wird.

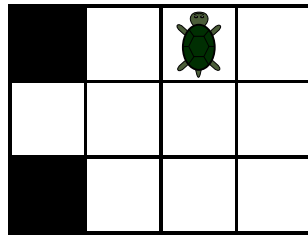


Abbildung 6: Einfaches Beispiel

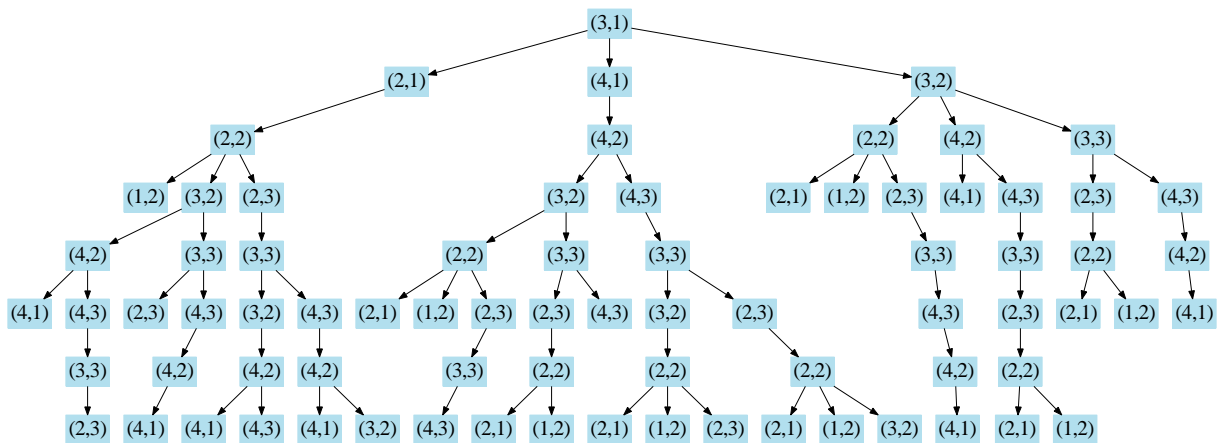


Abbildung 7: Aufrufbaum des Backtracking-Algorithmus.

Man betrachte zum Beispiel die Landkarte in Abb. 6. Benennen wir die weißen Felder anhand der Koordinaten, starten also bei Knoten $(3,1)$, so ergibt sich der umfangreiche Aufrufbaum in Abb. 7. Wie man sieht, gibt es keinen Hamilton-Pfad. Wir können nun einige Überlegungen anstellen, wie der Algorithmus schneller zu der Erkenntnis gelangen kann, dass es hier keine Lösung gibt.

1.3 Pruning

Eine Möglichkeit zur Verbesserung der Backtracking-Variante besteht darin, möglichst früh herauszufinden, dass ein Ast des Aufrufbaumes nicht zu einem Hamilton-Pfad führen kann. In der Informatik nennt man dies *pruning* (= Zurechtstutzen). Ein sehr einfacher Test besteht zum Beispiel darin, zu Beginn wie in Juniorkaufgabe 2 zu prüfen, ob die weißen Felder zusammenhängend sind. Wenn nicht, so kann kein Hamilton-Pfad existieren. Aber es gibt noch viele andere Möglichkeiten:

Zusammenhang prüfen Man kann nach jedem Schritt überprüfen, ob die noch nicht besuchten Felder zusammenhängend sind: Wenn nicht, so können nicht mehr alle Felder erreicht werden. Der Test kann allerdings recht aufwändig sein.

Zwingende Endpunkte Ein Feld, das nur einen weißen Nachbarn hat, muss das Ende des Pfades bilden (wie Feld $(1,2)$ im obigen Beispiel). Gibt es mehrere solche Felder, oder erzeugt man während der Suche mehrere solche Felder, kann kein Hamilton-Pfad mehr existieren und die Suche in diesem Zweig kann abgebrochen werden.

Parität zum Endpunkt Man kann die Welt von Kassiopeia wie ein Schachbrett einfärben. Gibt es einen zwingenden Endpunkt und sind (mit diesem) noch eine gerade Zahl von Feldern zu besuchen, so müssen der Ausgangspunkt und der Endpunkt die gleiche Schachbrettfarbe haben, bei einer ungeraden Anzahl eine ungleiche; denn bei jedem Schritt in Kassiopeias Welt ändert sich die Farbe des Feldes, auf dem man steht. Die Idee wird weiter unten in Abb. 9 noch ausführlicher beschrieben.

Gelenkpunkte Gibt es ein Feld, das zwei Zusammenhangskomponenten der weißen Felder verbindet, d.h. dass bei Wegnahme des Feldes die weißen Felder nicht mehr zusammenhängend wären, dann nennt man dieses Feld einen Gelenkpunkt. Dann kann man das Gesamtproblem in zwei (oder bei mehreren Gelenkpunkten in entsprechend viele) Unterprobleme aufteilen und jeweils Hamilton-Pfade von Gelenkpunkt zu Gelenkpunkt suchen. Dadurch erhält man viele zwingende Endpunkte und erhöht die Wahrscheinlichkeit, dass eine der obigen Pruning-Maßnahmen greift.

Wir schauen uns nun die einzelnen Möglichkeiten genauer auf Implementierung und Nutzen an.

Zusammenhang prüfen

Der Test auf Zusammenhang kann zum Beispiel wie in Junioraufgabe 2 gelöst werden, ist dann aber zu zeitaufwändig, wenn er nach jedem Rekursionsschritt durchgeführt werden soll. Stattdessen kann ein Konturen-Test durchgeführt werden: Dazu färbt man den gerade besuchten Knoten schwarz und sucht sich einen weißen Nachbarn aus. Dieser liegt am Rand des weißen Gebiets. Läuft man entlang des Randes um das Gebiet herum und erreicht dabei alle weißen Nachbarn des gerade schwarz gefärbten Knotens, so ist das weiße Gebiet nach wie vor zusammenhängend. Der Vorteil dieser Methode ist ihr mit durchschnittlich $O(\sqrt{n})$ bei n Knoten gutes Laufzeitverhalten, ein Nachteil die recht aufwändige Implementierung.

Als Beispiel betrachte man wieder die Landkarte in Abb. 6. Startete man mit einem Schritt nach unten und wollte anschließend weiter nach unten gehen, so teilte man das weiße Gebiet dadurch in zwei Hälften. Der Konturen-Test kann daher den rechts liegenden Knoten vom links liegenden aus nicht erreichen; der Schritt kann gleich wieder verworfen werden. Dadurch wird der Aufrufbaum deutlich kleiner, wie in Abb. 8 zu sehen.

Zwingende Endpunkte

Die Suche nach zwingenden Endpunkten kann zu zwei unterschiedlichen Zeitpunkten erfolgen: vor der eigentlichen Suche und nach jedem Rekursionsschritt. Die Suche zu Beginn kann erfolgen, indem für jeden weißen Knoten geprüft wird, ob er genau einen Nachbarn hat und nicht der Startknoten ist. In diesem Fall ist er ein zwingender Endknoten. Gibt es zwei oder mehr davon, kann direkt abgebrochen werden.

Die Suche in jedem Rekursionsschritt kann effizienter gestaltet werden. Dazu sollte ein eventuell zwingender Endpunkt per Parameter mit übergeben werden. Betritt man dann ein Feld, so prüft man alle noch nicht besuchten Nachbarn darauf, ob diese jetzt nur noch einen unbesuchten Nachbarn haben, und erzeugt eine Liste der neuen Endpunktkandidaten. Anschließend müssen drei verschiedene Fälle unterschieden werden:

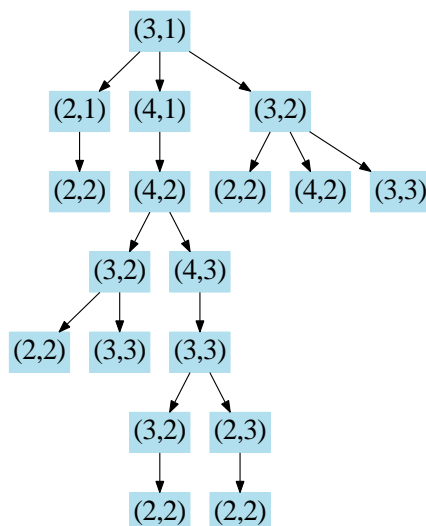


Abbildung 8: Reduzierter Aufrufbaum des Backtracking-Algorithmus mit Konturen-Test.

- Sind mehr als zwei der Nachbarn Endpunktkandidaten, oder genau zwei und gab es schon einen (als Parameter übergebenen) Endpunkt, der nicht in der Liste steht, so kann die Suche in diesem Zweig abgebrochen werden.
- Gibt es zwei Kandidaten und noch keinen Endpunkt oder einen Kandidaten und einen Endpunkt, so muss der Pfad entlang dieses Kandidaten weitergehen und im anderen Kandidaten enden (neuer Endpunkt) bzw. im bereits vorhandenen Endpunkt.
- Gibt es einen Kandidaten und noch keinen Endpunkt und betritt man einen anderen Knoten als den Kandidaten, so ist dies von nun an der neue Endpunkt.

Im Beispiel aus Abb. 6 gibt es den zwingenden Endpunkt $(1,2)$. Aber schon, wenn *Kassiopeia* startet, gibt es die zwei Nachbarn $(2,1)$ und $(4,1)$, die auch potentiell zwingende Endpunkte sind: Ginge man einen Schritt nach unten, hätte man genau diese drei zwingenden Endpunkte und die Suche muss erfolglos enden. Ginge man nach links oder rechts, blieben noch zwei zwingende Endpunkte, und eine weitere Suche muss ebenso erfolglos bleiben. Der Aufrufbaum besteht in diesem Fall also nur aus der Wurzel $(3,1)$ und die Suche wird praktisch sofort abgebrochen.

Paritätstest

Der Paritätstest ist sehr einfach zu implementieren: Wir sagen, dass ein Feld mit Koordinaten (x,y) von heller Schachbrettfarbe ist, wenn $c = x + y \equiv 0 \pmod{2}$, und von dunkler Farbe, wenn $c = x + y \equiv 1 \pmod{2}$ (nicht zu verwechseln mit der tatsächlichen Farbe des Feldes!). Das Zielfeld ist erreichbar, falls noch n Felder zu betreten sind und für die Farben c_S und c_T von Startpunkt S und Zielpunkt T gilt

$$c_S + n \equiv c_T \pmod{2}.$$

Im obigen Beispiel hilft der Paritätstest nicht weiter, da dort diese Gleichung erfüllt ist. Das vorgegebene Beispiel *kassiopeia2* jedoch ist anfällig für diesen Test. Zunächst färben wir das Feld wie ein Schachbrett, was in Abbildung 9 dargestellt ist.

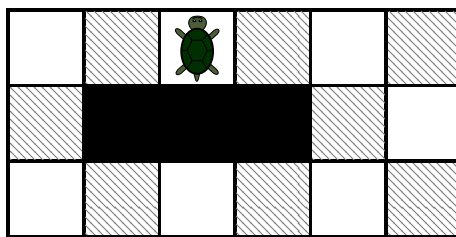


Abbildung 9: Veranschaulichung des Paritätstests. Egal, ob wir zuerst nach rechts oder nach links laufen, müssen wir anschließend noch 13 Schritte gehen, um schließlich das Endfeld links bzw. rechts vom Startfeld zu erreichen. Weil unser Standort und das Endfeld die gleiche Farbe haben, wir aber eine ungerade Anzahl von Schritten gehen müssen, kann es keinen korrekten Pfad geben.

Zwar gibt es zu Beginn noch kein zwingendes Endfeld, aber egal ob man rechts oder links herum startet, ergibt sich dieses sofort links bzw. rechts vom Startfeld und ist von dunkler Schachbrettfarbe, also $c_T = 1$. Wir stehen dann auch auf einem dunklen Schachbrettfeld ($c_S = 1$), es müssen aber noch $n = 13$ Felder betreten werden. Wir testen also

$$\begin{aligned} c_S + n &= 1 + 13 \\ &\equiv 0 \pmod{2}. \\ &\not\equiv c_T \pmod{2}. \end{aligned}$$

Das Zielfeld ist also nicht erreichbar und wir brechen sofort ab.

Auf den Gelenkpunkte-Test wollen wir hier nicht näher eingehen, da er bei den vorgegebenen Beispielen keinen großen Nutzen bringt. Er lässt sich aber ebenfalls effizient mithilfe der Konturensuche implementieren.

Kombiniert man die drei besprochenen Pruning-Methoden, erhält man folgenden, ziemlich umfangreichen Python-Pseudocode:

```

1 def hasHamiltonPath(graph, node, depth, target):
2     node.visited = True
3
4     # Alle Knoten erreicht?
5     if depth == graph.n:
6         return [node]
7
8     # Paritätstest:
9     if target != None:
10        color = (node.x + node.y) % 2
11        target_color = (target.x + target.y) % 2
12        nodes_left = graph.n - depth - 1
13        if (color + target_color + nodes_left) % 2 == 1:
14            # Falsche Parität
15            node.visited = False
16            return None
17
18        new_targets = []
19        # Endknotentest
20        # Finde alle Nachbarn, die jetzt nur noch einen

```

```

21     # Nachbarn haben
22     for v in node.neighbours:
23         if v != target and len(v.unvisited_neighbours) == 1:
24             new_targets.append(v)
25
26     t = len(new_targets)
27     # Fall 1:
28     if t > 2 or (t == 2 and target != None):
29         node.visited = False
30         return None
31     # Fall 2:
32     elif (t == 2 and target == None)
33         or (t == 1 and target != None):
34         path = hasHamiltonPath(graph, new_targets[0], depth+1,
35                               new_targets[1] if t == 2 else target)
36         if path != None:
37             return [node] + path
38         else:
39             node.visited = False
40             return None
41     # Fall 3: siehe unten
42
43     # Konturen-Test, wie auch immer er implementiert ist
44     if not contour_connects(graph, node):
45         node.visited = False
46         return None
47
48
49     # Eigentliche Rekursion
50     for v in node.neighbours:
51         if not v.visited:
52             # Fall 3 (versteckt)
53             if (t == 1 and v != new_targets[0]):
54                 the_target = new_targets[0]
55             else:
56                 the_target = target
57
58             path = hasHamiltonPath(graph, v, depth+1, the_target)
59             if path != None:
60                 return [node] + path
61
62     node.visited = False
63
64     return None

```

1.4 Dynamische Programmierung

Eine oft vielversprechende Möglichkeit, NP-vollständige Probleme in etwas besserer Laufzeit zu lösen, ist die dynamische Programmierung. Dabei kann auf einen Teil der Berechnungen auf Kosten von zusätzlichem Speicherverbrauch verzichtet werden. Sei wie oben der Startknoten

von Kassiopeia mit v_0 bezeichnet und die Menge aller Knoten mit V . Die Idee ist nun folgende: Man berechnet für jede Teilmenge $U \subset V \setminus \{v_0\}$ und einen Knoten $v \in U$, ob ein Pfad von v_0 über alle Knoten in U existiert, der in v endet. Die Funktion, die dies entscheidet, nennen wir $\text{isPossible}(U, v)$. Natürlich gilt für einen beliebigen Knoten v $\text{isPossible}(\{v\}, v)$ genau dann, wenn v ein Nachbar von v_0 ist, und allgemein lässt sich $\text{isPossible}(U, v)$ rekursiv berechnen: $\text{isPossible}(U, v)$ ist wahr genau dann, wenn es ein $u \in U$ gibt, sodass $\text{isPossible}(U \setminus \{v\}, u)$ wahr ist und u und v benachbart sind. Bildlich gesprochen: Es gibt einen solchen Pfad, der in v endet, wenn es einen um eins kürzeren Pfad gibt, der in einem Nachbarn von v endet.

Auf diese Weise kann man nacheinander alle einelementigen, dann die zweielementigen, usw. Teilmengen der Knotenmenge durchlaufen und die Funktion isPossible für alle Elemente der Menge ausrechnen. Hat man sich so zu $U = V \setminus \{v_0\}$ vorgearbeitet, muss man lediglich prüfen, ob es ein Element v gibt, für das $\text{isPossible}(V \setminus \{v_0\}, v)$ wahr ist, denn dann existiert ein Hamilton-Pfad. Möchte man anschließend den Pfad rekonstruieren, speichert man statt wahr oder falsch einfach den Knoten u , anhand dessen positiv über einen existenten Pfad beschieden wurde, bzw. None oder Null, falls es den nicht gibt.

Die Laufzeit dieser Methode liegt mit $O(n^2 2^n)$ tatsächlich etwas niedriger als die der reinen Backtracking-Variante. Der Speicherverbrauch ist aber mit $O(n 2^n)$ ähnlich groß und tatsächlich schon bei kleineren Feldern problematisch. Dank der Pruning-Methoden kann er auch laufzeit-technisch nicht mithalten, was hauptsächlich daran liegt, dass wir es hier mit planaren Graphen zu tun haben, die den Backtracking-Ansatz begünstigen.

1.5 Andere Algorithmen

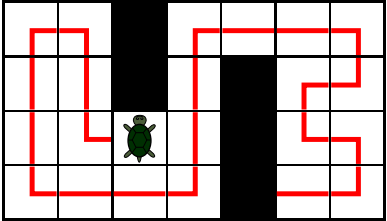
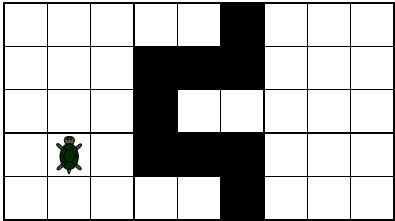
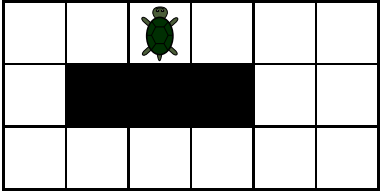
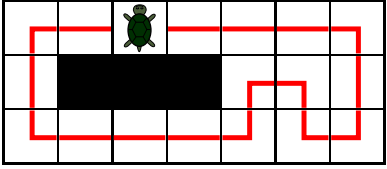
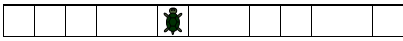

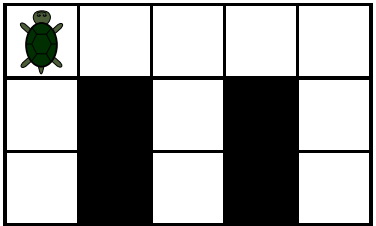
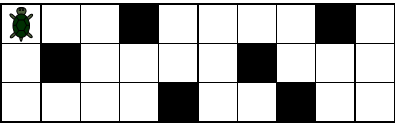
Es gibt noch eine Vielzahl anderer Algorithmen, die teilweise auf Heuristiken basieren, um möglichst schnell einen möglicherweise vorhandenen Pfad aufzuspüren. Zum Beispiel scheint es oft günstig zu sein, zuerst die Nachbarn zu besuchen, die selbst möglichst wenige Nachbarn haben. Es gibt außerdem viele Algorithmen, welche Heuristiken nutzen, um schnell Pfade zu finden, dabei aber unter Umständen valide Pfade übersehen. Findet ein solcher Algorithmus keine Lösung, muss also anschließend doch die aufwändige Suche genutzt werden. Viele dieser Algorithmen suchen Pfade im Graphen und versuchen diese dann zu verlängern.

Es ist sehr spannend, sich unterschiedliche Algorithmen anzuschauen. Die vorgegebenen Beispiele in dieser Aufgabe lassen sich allerdings auch mit der einfachen Backtracking-Methode in kurzer Zeit lösen, sodass der Druck, effizientere Algorithmen zu finden, ziemlich gering ist. Daher wird hier auch nicht näher darauf eingegangen.

1.6 Beispiele

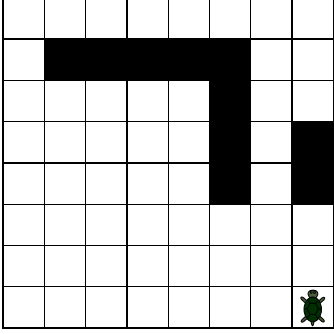
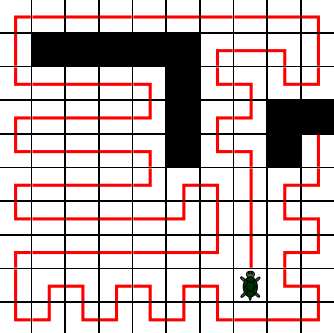
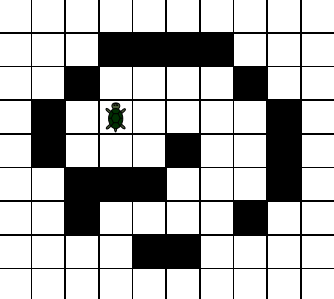
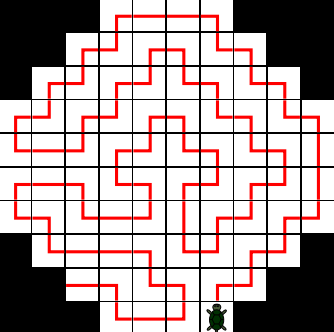
Zu guter Letzt geben wir die Lösungen zu den vorgegebenen Beispielen und einigen eigenen Beispielen an. Bei mehreren möglichen Pfaden sind jeweils alle Möglichkeiten alphabetisch sortiert angegeben.

Vorgegebene Beispiele

Nr.	Welt	Ausgabe
0		<pre> WNNWSSSSOOONNNNOOSSSWNN WNNWSSSSOOONNNNOOSWSOSW WNNWSSSSOOONNNNOOSWSSON WNNWSSSSOOONNNNOOSSONNN </pre>
1		Kein Pfad
2		Kein Pfad
3		<pre> OOOSSWNWSWWWNNO OSONOSSWWWWWNNO WSSOOOONOSONNWWW WSSOOOOOONNWSWNW </pre>
4		Kein Pfad
5		WWWWWWWWWWWW
6		Kein Pfad
7		Kein Pfad

Eigene Beispiele

Hier sind etwas größere Welten abgebildet, die sich nicht mit reinen Backtracking-Ansätzen lösen lassen. Insbesondere Nr. 1 ist an der Grenze des mit Pruning Machbaren.

Nr.	Welt	Ausgabe
0		Kein Pfad
1		NNNNWNONWNOO...ONWNON Viele weitere Pfade...
2		Kein Pfad
3		NONONONNNWNW...OSWWNW Viele weitere Pfade...

1.7 Bewertungskriterien

- Junioraufgabe 2 (Erreichbarkeit) muss als erster Teil dieser Aufgabe bearbeitet sein. In der Dokumentation muss angesprochen werden, auf welche Weise das erledigt wurde. Die Bearbeitung soll außerdem korrekt sein, also u.a. für die vorgegebenen Beispiele richtige Ergebnisse liefern (vgl. Tabelle 1).
- Funktionsweise und Korrektheit des Verfahrens (für den Hauptteil der Aufgabe) müssen nachvollziehbar erläutert bzw. begründet werden.
- Das benutzte Verfahren soll korrekt sein, es darf also nicht nur auf Heuristiken vertrauen. Allzu einfache Lösungen können nicht richtig sein.
- Es wird erwartet, dass die vorgegebenen Beispiele in brauchbarer Zeit verarbeitet und gelöst werden können. Ein normaler Backtracking-Ansatz bekommt das hin und ist deswegen grundsätzlich in Ordnung. Schlechter (bezüglich Laufzeit oder evtl. auch Speicherbedarf) darf es aber nicht sein. Wie bei Junioraufgabe 2 kann es zu unnötig hohem Aufwand führen, wenn für einzelne Felder nicht sauber verwaltet wird, ob sie bereits komplett abgearbeitet sind.
- In Ordnung ist natürlich, wenn – über die Aufgabenstellung hinaus – nicht nur ein Hamilton-Pfad, sondern alle Hamilton-Pfade ermittelt und ausgegeben werden.
- Wenn reines Backtracking genutzt wird, soll dessen Nachteil bei größeren Beispielen zumindest erwähnt sein. Diese Forderung ist auch dann erfüllt, wenn einfache Ideen zur Verbesserung angesprochen werden.⁶ Diskussion und Umsetzung wirksamer Pruning-Strategien sind natürlich lobenswert, werden aber nicht erwartet.
- Die Lösungen der vorgegebenen Beispiele müssen dokumentiert sein.

⁶Dieses Bewertungskriterium wurde häufig kritisiert, weil eine Betrachtung der Laufzeit in der Aufgabenstellung nicht ausdrücklich gefordert und die Bearbeitung aller vorgegebenen Beispiele mit einer einfachen Backtracking-Lösung problemlos möglich war. In einem Wettbewerb sollen aber Begabungen erkannt werden. Zu einer Begabung für Informatik gehört es, die eigene Arbeit immer auch kritisch bzw. analytisch zu betrachten; ohne diese Eigenschaft ist man nicht in der Lage, optimale und fehlerfreie Lösungen zu finden bzw. Programme zu schreiben. Die kritische Betrachtung einer Backtracking-Lösung führt unmittelbar zu der Erkenntnis, dass deren Laufzeit für größere Beispiele problematisch ist (und da die Aufgabenstellung „mindestens“ die Bearbeitung der vorgegebenen Beispiele fordert, ist es nicht abwegig, über weitere Beispiele nachzudenken). Es hätte gereicht, dies zu erwähnen, um einen Punktabzug zu vermeiden. Letztlich hat dieses Bewertungskriterium geholfen, gute Bearbeitungen von perfekten Bearbeitungen zu unterscheiden – das Kriterium wurde von vielen Bearbeitungen durchaus erfüllt. Ohne dieses Kriterium wäre die durchschnittliche Bewertung der Aufgabe übrigens ähnlich hoch ausgefallen wie die einer leichten Junioraufgabe.

Aufgabe 2: Ameisenfutter

2.1 Aufgabenstellung

Durch eine Simulation soll das Verhalten einer Menge von Ameisen auf einem Feld analysiert werden. Dabei soll der Einfluss der folgenden Parameter auf die Simulation ermittelt werden:

- Anzahl der Ameisen
- Position des Nests
- Anzahl der Futterquellen
- Verdunstungszeit (oder -rate o.Ä.) der Pheromone

Folgende Parameter sind fest und werden nicht verändert:

- Futter pro Futterquelle: 50
- Größe des Feldes: 500×500

Da die Aufgabenstellung an einigen Stellen Raum für verschiedene Interpretationen lässt, sollte zuerst geklärt werden, welche Interpretation der Aufgabe gewählt wird. Für diese Beispiellösung sollen folgende Dinge gelten:

- Mehrere Ameisen dürfen auf dem gleichen Feld stehen.
- Feld a grenzt direkt an Feld b bedeutet, dass a und b eine gemeinsame Kante besitzen. Daraus folgt unter anderem:
 - Ameisen können sich nur auf oben, unten, links oder rechts benachbarte Felder bewegen, nicht auf diagonal benachbarte.
 - Ameisen riechen die Pheromonkonzentration nur dieser direkt angrenzenden Felder.
- Ameisen bemerken eine Futterquelle genau dann, wenn sie direkt darauf oder auf einem angrenzenden Feld stehen.
- Die Verdunstung erfolgt nach und nach: Innerhalb der Verdunstungszeit sinkt die Pheromonkonzentration eines Duftpunktes gemäß einer bestimmten Verdunstungsrate. (Alternativ könnte die Konzentration eines Duftpunktes mit dem Ablauf der Verdunstungszeit abrupt von 100% auf 0% sinken.)
- Alle Ameisen beginnen bei Simulationsschritt 0 am Nest — wie vorgegeben.

Um das Verhalten der Ameisen zu beschreiben, benötigt man erst noch einige Schreibweisen:

- (x, y) (mit $x, y \in \mathbb{N}_0, 0 \leq x < 500, 0 \leq y < 500$) beschreibt einen Punkt auf dem Feld.
- $d(P_1, P_2) = d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$ beschreibt die Distanz zwischen zwei Punkten des Feldes, oder auch zwischen einer Ameise und dem Nest.
- $phero(P) = phero((x, y))$ beschreibt die Menge der Pheromone am Punkt P .

Dann sei für eine Ameise an der Position (x, y) die Funktion $next((x, y))$ wie folgt definiert:

$$\text{Kandidaten} = \{(a, b) \mid d((a, b), (x, y)) = 1 \wedge d(\text{Nest}, (a, b)) > d(\text{Nest}, (x, y))\}$$

$$next((x, y)) = (a, b) \in \text{Kandidaten, so dass } phero((a, b)) = \max_{phero}(\text{Kandidaten})$$

Anschaulich ist $next((x, y))$ das Feld, das an die Ameise grenzt, weiter als das Feld der Ameise vom Nest entfernt ist und von allen an die Ameise grenzenden Feldern die maximale Menge an Pheromonen hat. Bei gleicher Pheromonmenge wird ein Feld zufällig gewählt.

Jede Ameise an der Position (x, y) verhält sich damit nach dem in Abb. 10 dargestellten Flowchart.

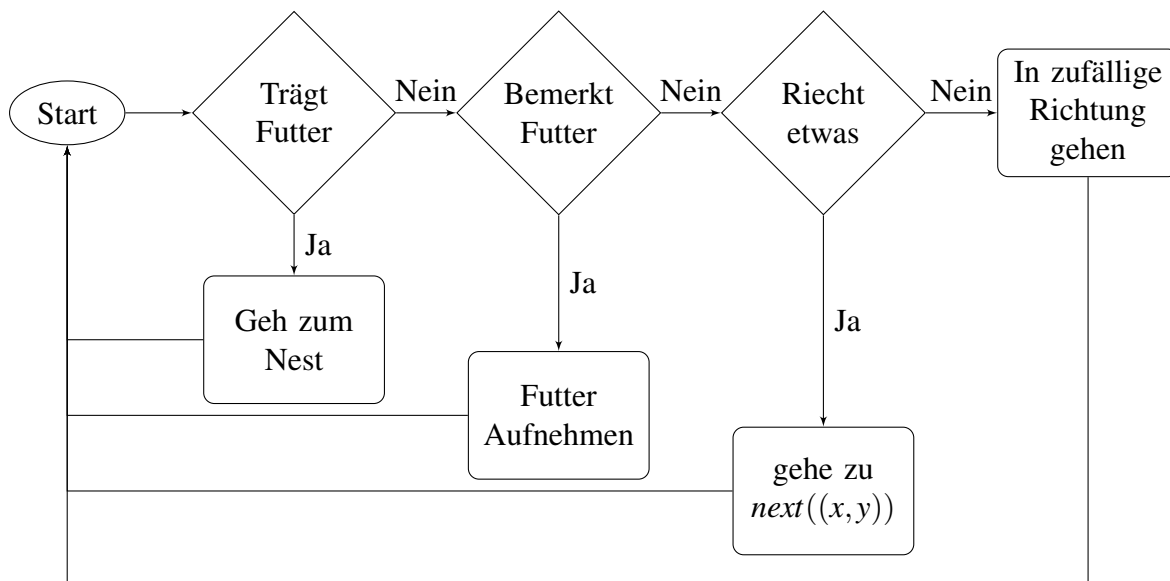


Abbildung 10: Verhalten der Ameisen in der Simulation

Terminologie

Zum präziseren Beschreiben der Vorgänge seien folgende Begriffe festgelegt:

- Tick: ein Simulationsschritt
- Verdunstungsrate: Eine Ameise platziert eine Dufteinheit auf einem Feld, wenn sie Futter trägt. Der Anteil der Menge, der pro Tick verdunstet, ist die Verdunstungsrate. Zum Beispiel bedeutet eine Verdunstungsrate von 0,5%, dass eine Einheit nach 200 Ticks verdunstet, 2 Einheiten nach 400 usw.
- Laufzeit: Anzahl der Ticks, bis alle Futtereinheiten eingesammelt wurden

2.2 Lösungsidee

Um die Effekte der verschiedenen Parameter auf die Simulation zu ermitteln, muss zuerst eine effiziente Simulation entwickelt werden.

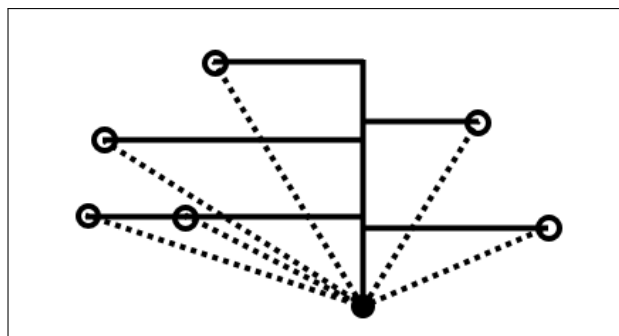


Abbildung 11: Zwei Arten sinnvoller Wege

Datenstrukturen

Dafür sollte man sich überlegen, welche Abfragen man wie oft durchführen muss und darauf basierend sich passende Datenstrukturen suchen. Folgende Abfragen müssen mindestens möglich sein (# steht für „Anzahl“):

Anfrage	Zugriffszahl pro Tick
Ameise → aktuelle Position	#(Ameisen)
Ameise → trägt Futter?	#(Ameisen)
Feld → Futter(-menge)	$5 \cdot \text{\#(Ameisen)}$
Feld → Pheromonmenge	$4 \cdot \text{\#(Ameisen)} + \text{\#(Felder mit Pheromonen)}$
Menge der Felder mit Pheromonen	1

Die ersten vier Anfragen lassen sich einfach über Arrays umsetzen. Die Menge der Felder mit Pheromonen wird benötigt, um bei jedem Tick die Verdunstung zu simulieren. Sie lässt sich erstellen und aktualisieren, indem gespeichert wird, auf welchen Feldern Ameisen Pheromone platzieren.

Wegfindung

Auch sollte die Bedeutung der Wegfindung der Ameisen auf dem Weg zurück zum Nest bedacht werden. Da sich die Ameisen auf einem Gitter bewegen, ist jeder Weg sinnvoll, der sich in jedem Schritt dem Nest annähert. Hierbei können verschiedene Varianten gewählt werden. Zwei Beispiele dafür sieht man in Abb. 11. Sowohl die gepunkteten als auch die durchgezogenen Linien stellen sinnvolle Wege dar. Bewegen sich die Ameisen auf den durchgezogenen Linien, überdecken sich die Duftspuren, und die Pheromonkonzentration auf den gemeinsamen Stücken nimmt enorm zu. Bei den einer Geraden angenäherten, gepunkteten Wegen überlagern sich die Pfade nur in der Nähe des Nestes. Dies sorgt für ein anderes Verhalten der Ameisen beim Folgen der Duftspuren und letzten Endes auch für verschiedene Laufzeiten.

Keine der Varianten ist falsch, solange man sich des Effektes bewusst ist. In dieser Beispiellösung nutzen die Ameisen die Approximation einer Geraden (gepunktete Linien).

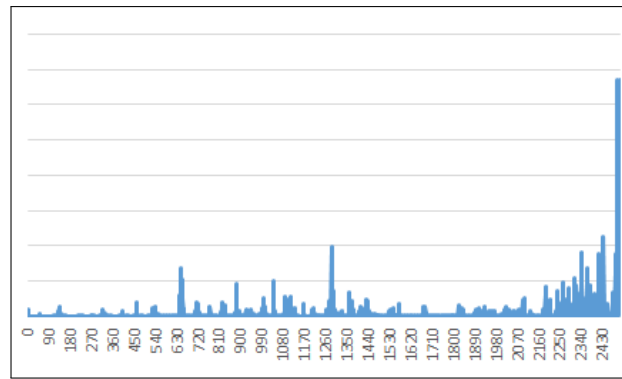


Abbildung 12: Diagramm der Sammelrate

Komponenten

Das erstellte Programm sollte die folgenden Komponenten enthalten:

- Simulation: Berechnet aus einem Zustand den nächsten.
- Grafik: Berechnet aus einem Zustand ein aussagekräftiges Bild.
- Statistik: Sammelt verschiedene Daten während der Simulation zur späteren Auswertung.

Wenn in den Vorüberlegungen die Datenstrukturen sorgfältig gewählt wurden, sollte die Umsetzung keine bösen Überraschungen bereithalten. Mit den beschriebenen Ansätzen sollten sich über 30.000 Ticks pro Sekunde erreichen lassen.

Potentielle Fallstricke

Da sich alle Ameisen gleichzeitig bewegen, dürfen neue Duftmarken erst platziert werden, nachdem die Aktionen aller Ameisen ermittelt wurden. Dies ist aufwändiger, als die Pheromonänderungen sofort auszuführen, ist jedoch notwendig, um die Simulation korrekt zu halten. Die anstehenden Änderungen lassen sich zum Beispiel als eine Liste von Punkten speichern. Nachdem sich alle Ameisen bewegt haben, wird bei den in der Liste gespeicherten Punkten die Pheromonmenge erhöht.

Des Weiteren ist es sicherlich verlockend, die Menge der Felder mit Pheromonen nicht zu berechnen und stattdessen jeden Tick über alle Felder zu iterieren und darüber die Verdunstung durchzuführen. Jedoch bedeutet das bei 500×500 Feldern 250.000 Zugriffe pro Tick gegenüber den im Normalfall weniger als 1000 Zugriffen. In der Praxis ist die zweite Variante ca. 40-60 mal schneller.

2.3 Beobachtungen

Parameterunabhängige Beobachtungen

Die folgenden Beobachtungen sollten allgemein festzustellen sein, unabhängig von den hier getroffenen Annahmen über die Fähigkeiten der Ameisen:

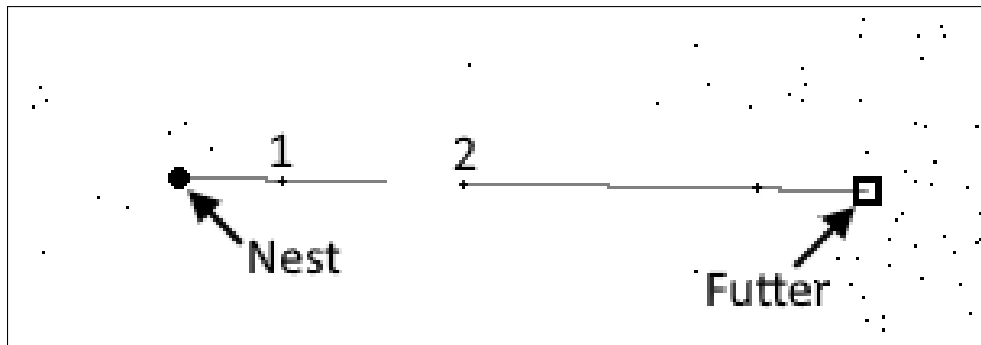


Abbildung 13: Beispiel zum Verlieren einer Duftspur

- Die Rate, mit der Futter gefunden wird, nimmt mit der Zeit ab. Das Diagramm in Abb. 12 stellt die Intervalle zwischen dem Einsammeln von Futter in Abhängigkeit zu der Anzahl der bereits gesammelten Futtereinheiten dar. Auf eine Skala wurde verzichtet, da diese stark von Parameterwerten abhängt, die generelle Form des Diagramms aber nicht.
- Futterquellen, die näher an der Mitte liegen, werden im Schnitt eher gefunden.

Parameterabhängige Beobachtungen

Die folgenden Effekte lassen sich jeweils nicht auf einen einzelnen Parameter zurückführen, sondern nur auf eine Kombination von Parametern. Aus diesem Grund können keine festen Parameterwerte angegeben werden, ab denen die Effekte auftreten. Stattdessen soll sich mit einem Beispiel begnügt werden.

Verlieren einer Duftspur

Dieser Effekt tritt auf, wenn eine oder mehrere der folgenden Aussagen gilt:

- Das Nest liegt dezentral (0, 0).
- Die Verdunstungsrate ist groß (2%).
- Es gibt wenige Ameisen (10).
- Es gibt weit auseinanderliegende Futterquellen.

Am in Abb. 13 dargestellten Beispiel soll dieser Effekt erklärt werden. Ameise 1 hatte eine Duftspur vom Futter zum Nest gezogen und will der gleichen Duftspur jetzt zurück zum Futter folgen. Jedoch endet diese jetzt, bevor das Futter erreicht ist. Dadurch „verliert“ die Ameise die Futterquelle wieder und muss zufällig herumirren, bis sie diesen oder einen anderen Haufen findet. In dem gezeigten Bild hat durch Zufall auch Ameise 2 das Futter entdeckt und zieht die Duftspur neu.

Dieser Effekt führt zu einer erhöhten Laufzeit, wenn die Ameisen weit vom Nest entfernte Futterquellen wiederholt verlieren.

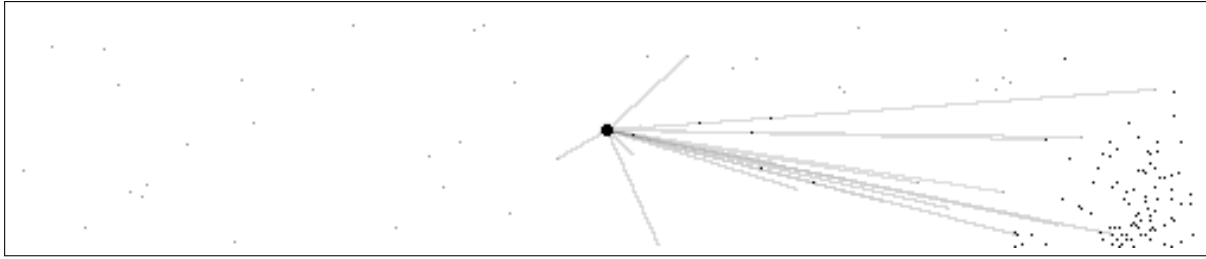


Abbildung 14: Beispiel einer toten Duftspur

Tote Duftspuren

Dieser Effekt tritt auf, wenn eine oder mehrere der folgenden Aussagen gilt:

- Das Nest liegt zentral (250, 250).
- Die Verdunstungsrate ist klein (0,1%).
- Es gibt viele Ameisen (1000).
- Es gibt geclusterte Futterquellen.

In Abb. 14 sieht man, wie Ameisen einem Pfad gefolgt sind, der schon nicht mehr zu Futter führt. Jetzt sind sie alle in der unteren rechten Ecke des Feldes gesammelt, wodurch es erheblich länger dauert, bis die Futterstellen auf der linken Seite gefunden werden.

Letzten Endes hängt die Laufzeit von allen Parametern ab: Wenn das Nest in der Ecke sitzt, ist die (bezüglich der Laufzeit) optimale Verdunstungsrate anders als wenn das Nest in der Mitte sitzt.

2.4 Simulationsphasen: ein Beispiel

An dem in Abb. 15 dargestellten Beispiel werden die verschiedenen Phasen, die bei den meisten typischen Parametern auftreten, illustriert. Es wurden 50 Futterquellen mit je 50 Futtereinheiten, 100 Ameisen, eine Verdunstungsrate von 0,4% und das Nest in der Mitte als Parameter verwendet.

Startphase: Diese findet in den ersten 10.000 Ticks statt. Die Ameisen verbreiten sich in alle Richtungen, die Futterquellen nahe am Nest werden gefunden und geleert.

Clustering: Es kristallisiert sich eine Richtung heraus, in der besonders viele Futterquellen gefunden werden. In diese bewegt sich ein Großteil der Ameisen. Dadurch bildet sich eine Ecke oder Kante heraus, bei der sich nahezu alle Ameisen befinden. In diesem Beispiel ist es die obere linke Ecke. Diese Phase findet bis etwa Tick 50.000 statt.

Verteilen: Die Region, in der sich die Ameisen konzentriert hatten, ist frei von Futter und die Duftspuren verschwinden. Jetzt beginnen die Ameisen, sich von dieser Zone weg auszubreiten. Dabei werden die gefundenen Futterquellen zügig geleert. Diese Phase fand bis ca. Tick 400.000 statt.

Reste sammeln: Die Ameisen sind gleichmäßig auf dem Feld verteilt und sammeln die letzten Futterreste ein. Diese Phase kann unter Umständen enorm lange dauern, wenn die Futterquellen ungünstig liegen oder der Zufall nicht mitspielt. Im Beispiel endete diese Phase bei 516.473 Ticks.

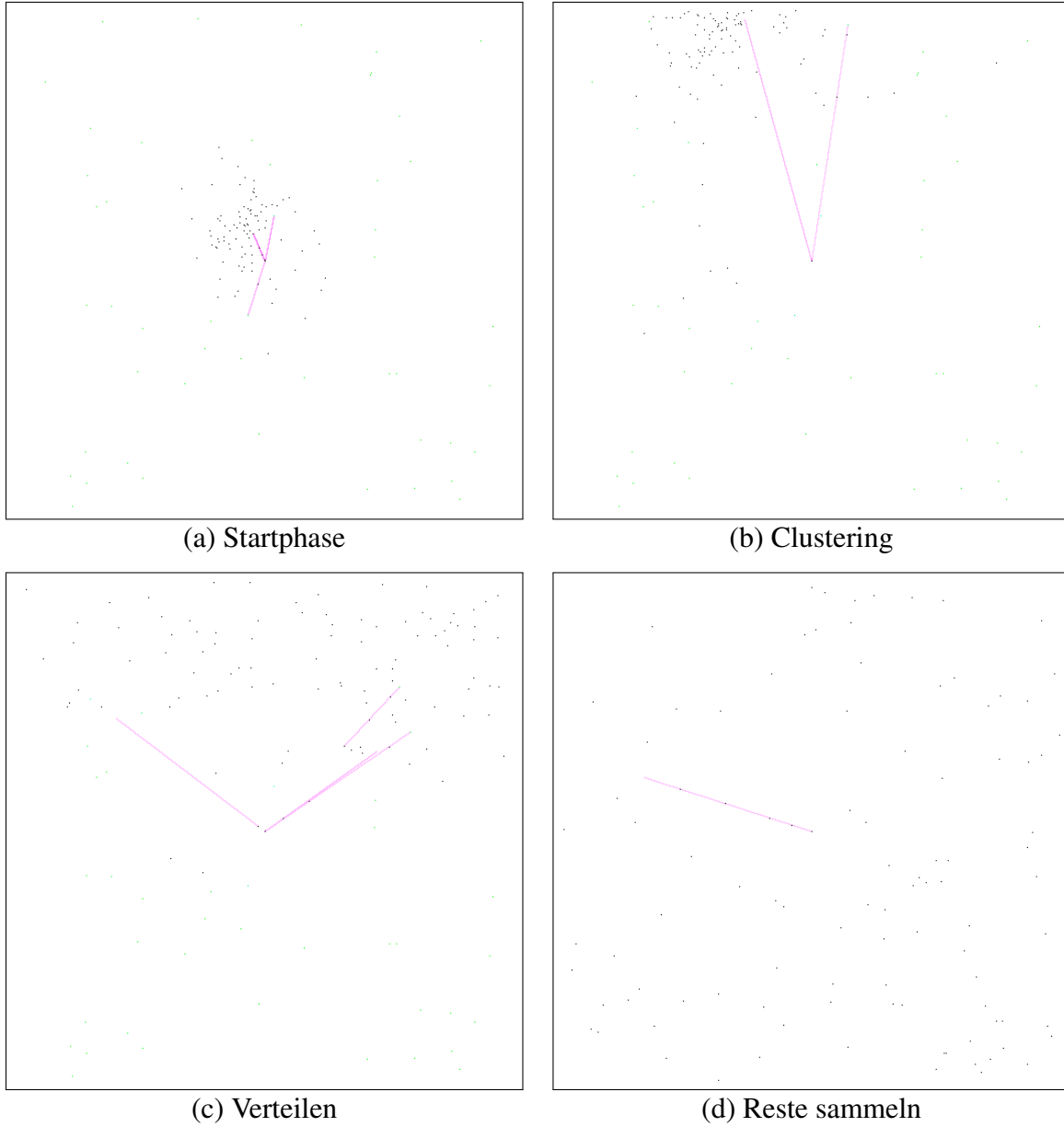


Abbildung 15: Verschiedene Phasen der Simulation

2.5 Bewertungskriterien

- Das Verhalten der Ameisen bzw. des gesamten Simulationsprozesses muss gut nachvollziehbar beschrieben sein. Eine formalisierte Beschreibung durch ein Flowchart o.Ä. ist lobenswert, wird aber nicht erwartet.
- Die Aufgabenstellung ist nicht in allen Punkten absolut präzise – eine ganz normale Spezifikation eben. Einiges ist also Auslegungssache: Welche Felder werden als „direkt angrenzend“ behandelt: nur oben, unten, links und rechts oder auch diagonal? Wie verdunsten die Pheromone: graduell oder sprunghaft? Wie wirkt die Verdunstungszeit bzw. -rate: auf jeden Duftpunkt einzeln oder auf die gesamte Duftmenge eines Feldes? Schön ist, wenn eine Bearbeitung zu diesen Punkten klar Stellung bezieht; zwingend erwartet wird das aber nicht. Nur Festlegungen, die *im Gegensatz* zur Aufgabenstellung stehen oder scheinbar wenig sinnvoll sind (z.B. wenn man die Ameisen auch ohne Futter Richtung Nest laufen lässt), müssen klar erwähnt und begründet sein.
- Die genauen Abläufe der Simulation sind natürlich von der oben erwähnten Auslegung der Aufgabenstellung abhängig, müssen aber damit stimmig sein. In der Simulation sollte die Verarbeitung der Duftpunkte und die Bewegung der Ameisen auseinander gehalten werden, damit die Ergebnisse nicht verfälscht werden. Insbesondere darf die Verdunstung der Duftpunkte erst dann berechnet werden, wenn alle Bewegungen vollzogen sind. Toleriert wird, wenn das Ablegen der Duftpunkte mit der Bewegung vermischt ist.
- Die in der Aufgabenstellung genannten Simulationsparameter (Anzahl der Ameisen, Position des Nests, Anzahl der Futterquellen und Verdunstungszeit) müssen für jeden Simulationslauf einstellbar sein. Ob das über Aufrufparameter in der Kommandozeile, eine Einstellungsdatei oder eine GUI ermöglicht wird, ist unerheblich. Nicht akzeptabel ist, wenn die Änderung der Parameter im Quellcode erfolgen muss; in diesem Fall kann man nicht mehr davon sprechen, dass das Programm „für verschiedene Werte ... verwendet werden kann“ (s. Aufgabenstellung). Eine über die Anforderungen der Aufgabenstellung hinausgehende Parametrisierung der Simulation – ein weiterer Parameter kann z.B. die Größe des Simulationsareals sein – ist natürlich kein Mangel, sondern begrüßenswert.
- Die grafische Darstellung muss kein Kunstwerk sein, jedoch sollten die Ameisen, die Futterquelle, das Nest und die Pheromonkonzentration erkennbar sein.
- Es sollen beobachtete Effekte beschrieben und mit Daten und Beispielen illustriert werden. Insgesamt sollte an um die drei aussagekräftigen Beispielen gezeigt werden, wie sich die Ameisen verhalten. Zu einem Beispiel sollten nicht die 3141592 Einzelbilder einer Simulation gezeigt werden, es reichen einige wenige aussagekräftige.

Aufgabe 3: Flaschenzug

3.1 Lösungsideen

Bei dieser Aufgabe wird die Anzahl an verschiedenen Möglichkeiten $A_F(N; k)$ gesucht, N nicht unterscheidbare Flaschen auf k unterscheidbare Behälter b_1, b_2, \dots, b_k zu verteilen. Dabei passen in den Behälter b_i maximal $F(b_i)$ Flaschen.

Speziell geht es also darum, die Funktion A_F an dem Punkt $(N; k)$ auszuwerten. Dabei sind die Fassungsvermögen F der Behälter als Nebenbedingungen zu berücksichtigen.

Für manche N und k lässt sich die Funktion A_F sehr einfach auswerten. So gibt es nur eine Möglichkeit, keine Flaschen auf Behälter zu verteilen. Für alle k gilt also:

$$A_F(0; k) = 1.$$

Einfach auswertbar wird A_F auch, wenn es überhaupt keine Behälter gibt. Dann ist es unmöglich die Flaschen zu verteilen. Sind allgemeiner mehr Flaschen zu verteilen, als alle Behälter zusammen fassen können, so gilt:

$$A_F(N; k) = 0 \text{ für alle } N > \sum_{i=1}^k F(b_i).$$

Für den etwas theoretisch wirkenden Fall, dass weniger als 0 Flaschen verteilt werden sollen, existiert ebenfalls keine mögliche Verteilung der Flaschen.

Brute Force

Für allgemeine N, k und F soll $A_F(N; k)$ zunächst mithilfe einer *Brute-Force-Strategie* berechnet werden. Es sollen also alle Möglichkeiten generiert und dabei gezählt werden. Bei der Funktion A_F bietet es sich an, diese Generierung und Zählung *rekursiv* durchzuführen: Zuerst werden $0 \leq m \leq F(b_1)$ Flaschen in den ersten Behälter b_1 gelegt. Dann wird berechnet, wie viele Möglichkeiten es gibt, die verbleibenden $N - m$ Flaschen auf die verbleibenden Behälter b_2, \dots, b_k zu verteilen. Speziell wird dann also $A_{F'}(N - m, k - 1)$ berechnet, wobei $F'(b_i) = F(b_{i+1})$ gilt.

Da ja *alle* Möglichkeiten gezählt werden sollen, müssen zur korrekten Berechnung von $A_F(N, k)$ alle Möglichkeiten für m berücksichtigt werden. Es ergibt sich für $A_F(N; k)$ also folgende *Rekursionsgleichung*:

$$A_F(N; k) = \sum_{m=0}^{F(b_1)} A_{F'}(N - m; k - 1),$$

wobei $F'(b_i) = F(b_{i+1})$ für alle positiven $i < k$ gilt. Zusammen mit den Basisfällen für $N \leq 0$ und $k = 0$ ergibt sich Algorithmus 3.

Wie man an einem einfachen Beispiel erahnen kann, ist diese Methode der Berechnung sehr langsam. So müssen zur Lösung des Beispiels der Aufgabenstellung

$$A_{\{(b_1;3);(b_2;5)\}}(7, 2)$$

Algorithmus 3 Brute ForceEingabe: N , k , Array an Fassungsvermögen $F[0..(k-1)]$ Ausgabe: $AF(N, k)$

```

ANZAHL( N, k, F[0..(k-1)] )
  if N = 0 then return 1
  if N < 0 then return 0
  if k = 0 then return 0

  result := 0
  for m := 0..F[0] do
    result := result + ANZAHL( N-m, k-1, F[1..(k-1)] )
  return result

```

folgende Berechnungen durchgeführt werden:

$$\begin{aligned}
A_{\{(b_1;3);(b_2;5)\}}(7;2) &= \sum_{m=0}^3 A_{\{(b_1;5)\}}(7-m;2-1) \\
&= A_{\{(b_1;5)\}}(7;1) + A_{\{(b_1;5)\}}(6;1) + A_{\{(b_1;5)\}}(5;1) + A_{\{(b_1;5)\}}(4;1) \\
&= \sum_{m=0}^5 A_{\{\}}(7-m;1-1) + \sum_{m=0}^5 A_{\{\}}(6-m;1-1) \\
&\quad + \sum_{m=0}^5 A_{\{\}}(5-m;1-1) + \sum_{m=0}^5 A_{\{\}}(4-m;1-1) \\
&= A_{\{\}}(7;0) + A_{\{\}}(6;0) + A_{\{\}}(5;0) + A_{\{\}}(4;0) + A_{\{\}}(3;0) + A_{\{\}}(2;0) \\
&\quad + A_{\{\}}(6;0) + A_{\{\}}(5;0) + A_{\{\}}(4;0) + A_{\{\}}(3;0) + A_{\{\}}(2;0) + A_{\{\}}(1;0) \\
&\quad + A_{\{\}}(5;0) + A_{\{\}}(4;0) + A_{\{\}}(3;0) + A_{\{\}}(2;0) + A_{\{\}}(1;0) + A_{\{\}}(0;0) \\
&\quad + A_{\{\}}(4;0) + A_{\{\}}(3;0) + A_{\{\}}(2;0) + A_{\{\}}(1;0) + A_{\{\}}(0;0) + A_{\{\}}(-1;0) \\
&= 0+0+0+0+0+0 \\
&\quad +0+0+0+0+0+0 \\
&\quad +0+0+0+0+0+1 \\
&\quad +0+0+0+0+1+0 \\
&= 2
\end{aligned}$$

Die Zahl 2 kann also auch sehr kompliziert dargestellt werden. Auch allgemein wird der Algorithmus jede Lösung letztendlich nur aus dem Aufaddieren von Einsen und Nullen gewinnen können. Schließlich sind dies die einzigen Werte, die nach dem Erreichen einer Abbruchbedingung direkt zurückgegeben werden können.

Durch eine geschickte Implementierung kann noch verhindert werden, dass nie die Anzahl der Möglichkeiten für eine negative Flaschenanzahl berechnet wird. Doch selbst dann werden bei jeder Berechnung (schlimmstenfalls) $N+1$ Teilprobleme berechnet. Bei diesen Teilproblemen ist das Argument k jedoch nur um 1 geringer als dasjenige des ursprünglichen Problems. Insgesamt muss das Ergebnis also schlimmstenfalls aus $(N+1)^k$ Nullen und Einsen zusammgebaut werden.

Das obige Beispiel verdeutlicht diese Rechnung. So müssen für das eigentliche Problem

$$A_{\{(b_1;3);(b_2;5)\}}(7,2)$$

zwar nicht 8, aber immerhin 4 Teilprobleme gelöst werden. Um jedes dieser Teilprobleme an sich jedoch zu lösen, werden je 6 Berechnungen notwendig. Insgesamt müssen also schon $4 \cdot 6 = 24 \leq (7+1)^2$ Teilprobleme gelöst werden. Glücklicherweise ist bei all diesen Teilproblemen $k=0$ oder $N=0$, sodass für diese Teilprobleme dann direkt die Lösung zurückgegeben werden kann (also je entweder eine 0 oder 1). Hätten in die beiden Behälter dagegen je 7 Flaschen gepasst, so hätten auch tatsächlich $(7+1)^2 = 64$ Berechnungen angestellt werden müssen.

Dieser Algorithmus berechnet zwar stets das korrekte Ergebnis, jedoch (sehr) langsam. Für etwas größere N oder k kann dieser Algorithmus also nicht verwendet werden.

Vermeiden doppelter Berechnungen

Es wird also ein Algorithmus benötigt, der (sehr viel) schneller die Lösung berechnet als der Brute-Force-Algorithmus. Um eine dazu hilfreiche Idee zu entwickeln, soll zunächst auf ein Beispiel aus der Mathematik zurückgegriffen werden, die *Fibonacci-Zahlen*. Die n -te Fibonacci-Zahl $Fib(n)$ ist definiert durch:

$$Fib(n) := \begin{cases} 1, & \text{für } n \leq 1 \\ Fib(n-1) + Fib(n-2), & \text{sonst} \end{cases}$$

Versucht man die n -te Fibonacci-Zahl direkt nach dieser Definition auszurechnen, so stößt man auf das gleiche Problem wie bei der Berechnung von $A_F(N;k)$:

$$\begin{aligned} Fib(5) &= && Fib(4) && + && Fib(3) \\ &= && Fib(3) &+ & Fib(2) &+ & Fib(2) &+ & Fib(1) \\ &= & Fib(2) &+ & Fib(1) &+ & Fib(1) &+ & Fib(0) &+ & Fib(1) &+ & Fib(0) &+ & 1 \\ &= & Fib(1) &+ & Fib(0) &+ & 1 &+ & 1 &+ & 1 &+ & 1 &+ & 1 &+ & 1 &+ & 1 \\ &= & 1 &+ & 1 &+ & 1 &+ & 1 &+ & 1 &+ & 1 &+ & 1 &+ & 1 &+ & 1 \\ &= & 8 \end{aligned}$$

Die n -te Fibonacci-Zahl wird durch Addition (sehr) vieler Einsen gewonnen. Bei dieser Berechnung fällt jedoch auf, dass Fibonacci-Zahlen *mehrfach* berechnet werden. So wird in dem Beispiel dreimal die 2. und zweimal die 3. Fibonacci-Zahl berechnet. Intuitiv scheint es wenig sinnvoll zu sein, genau das selbe mehr als einmal zu berechnen. Speichert man sich die schon berechneten Fibonacci-Zahlen, so vereinfacht sich die Rechnung in dem Beispiel erheblich:

$$\begin{aligned} Fib(5) &= && Fib(4) && + && Fib(3) \\ &= && Fib(3) &+ & Fib(2) &+ & 3 \\ &= & Fib(2) &+ & Fib(1) &+ & 2 &+ & 3 \\ &= & Fib(1) &+ & Fib(0) &+ & 1 &+ & 2 &+ & 3 \\ &= & 1 &+ & 1 &+ & 1 &+ & 2 &+ & 3 \\ &= & 8 \end{aligned}$$

Zur Erklärung: Es wird immer zunächst der linke Term komplett berechnet, *bevor* der rechte Term berechnet wird. Deshalb ist auch in der ersten Zeile das Ergebnis von $Fib(3)$ bekannt und kann direkt eingesetzt werden.

Schaut man sich nun erneut die beispielhafte Berechnung von $A_{\{(b_1;3);(b_2;5)\}}(7,2)$ an, so werden auch dort viele Werte für A mehrfach berechnet. Zwar werden bei diesem Beispiel nur Fälle mehrfach aufgerufen, von welchen sofort das Ergebnis bestimmt werden kann. Es ist jedoch leicht zu sehen, dass in anderen Beispielen durchaus auch komplexere Berechnungen mehrfach durchgeführt werden. Folglich soll der Brute-Force-Algorithmus nun so abgeändert werden, dass er keine Werte mehrfach berechnet.

Es muss also eine Möglichkeit gefunden werden, für zwei Aufrufe feststellen zu können, wann sie dasselbe Ergebnis berechnen. Bei den Fibonacci-Zahlen berechnen zwei Aufrufe $Fib(a)$ und $Fib(b)$ dasselbe, wenn $a = b$ gilt.

Es stellt sich heraus, dass mit dem Paar (N, k) ein Aufruf von $A_{F'}$ schon eindeutig bestimmt wird. Schließlich wird aus den Fassungsvermögen F' immer nur genau dann das erste Fassungsvermögen entfernt, wenn auch k um 1 sinkt. Aus einem k und den ursprünglichen Fassungsvermögen F können für einen Aufruf also direkt die korrekten Fassungsvermögen F' berechnet werden.

Für jedes im Verlauf der Berechnungen auftretende Paar (N, k) kann nun einfach der Wert von $A_F(N, k)$ gespeichert werden. Soll A_F für ein Paar (N', k') berechnet werden, für welches schon ein Wert gespeichert ist, so wird anstelle einer Berechnung der gespeicherte Wert zurückgegeben. Umgesetzt wird diese Idee in Algorithmus 4.

Algorithmus 4 Vermeiden doppelter Berechnungen

Eingabe: N , k , Array an Fassungsvermögen $F[0..(k-1)]$

Ausgabe: $AF(N, k)$

```

sp[ 0..startN ][ 0..startk ] //Zu Beginn mit einem
    //besonderen Wert (z.B. -1) gefüllt, der anzeigt,
    //dass der Wert an dieser Stelle noch berechnet werden muss
ANZAHL2( N, k, F[ 0..(k-1) ] )
    if N = 0 then return 1
    if N < 0 then return 0
    if k = 0 then return 0

    if sp[ N ][ k ] != -1 then
        //Gesuchter Wert wurde bereits berechnet
        return sp[ N ][ k ]

    sp[ N ][ k ] := 0
    for m := 0..F[0] do
        if N - m < 0 then break //break, um Zugriffsfehler
                                //bei Arrays zu vermeiden
        sp[ N ][ k ] := sp[ N ][ k ] + ANZAHL2(N-m, k-1, F[1..(k-1)])
    return sp[ N ][ k ]

```

Da A_F eine Funktion ist, gibt es für jedes Paar (N, k) genau einen Wert $A_F(N; k)$. Folglich entspricht ein gespeicherter Wert stets demjenigen, den eine erneute Berechnung ergeben würde. Algorithmus 4 liefert also auch das korrekte Ergebnis.

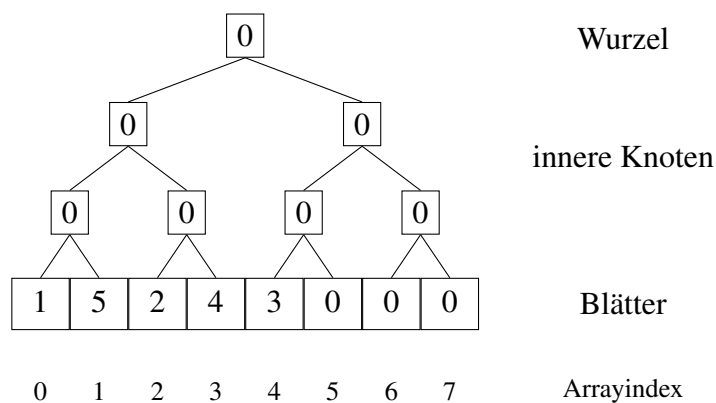
Dieser Algorithmus wird schlimmstenfalls $O(N^2k)$ Operationen durchführen, nämlich wenn das gesamte Array `sp` befüllt werden muss. Um den Wert für ein Paar (N, k) zu bestimmen, werden dabei $O(N)$ Operationen benötigt. Dieser Algorithmus reicht also aus, um alle Beispiele der Aufgabenstellung in wenigen Sekunden berechnen zu können.

Binärbäume

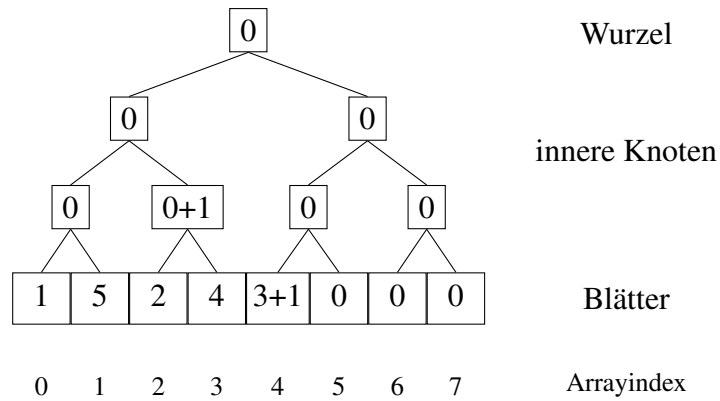
Für größere Eingabewerte kann reines Speichern von schon berechneten Werten immer noch zu langsam sein. Daher soll der bereits gefundene Algorithmus noch beschleunigt werden.

Es wird sich herausstellen, dass dazu eine Datenstruktur benötigt wird, mit deren Hilfe man *schnell* einen konstanten Wert c zu jedem Wert in einem Teil eines Arrays $a[1..n]$ addieren kann. Eine solche Datenstruktur ist ein *vollständiger Binärbaum*. Dazu werden in dessen Blättern die Werte des Arrays gespeichert. In der Wurzel und in den inneren Knoten werden die Werte gespeichert, die zu allen Blättern unter diesem Knoten aufaddiert werden sollen. Um dann einen konkreten Wert zu berechnen, werden alle Werte auf dem Pfad von dem entsprechenden Blatt zur Wurzel aufaddiert.

Das Array $[1, 5, 2, 4, 3]$ soll als Beispiel dienen. Zunächst wird ein Binärbaum über diesem Array konstruiert, dessen Knoten (bis auf die im Array enthaltenen Blätter) zu Beginn alle den Wert 0 haben:

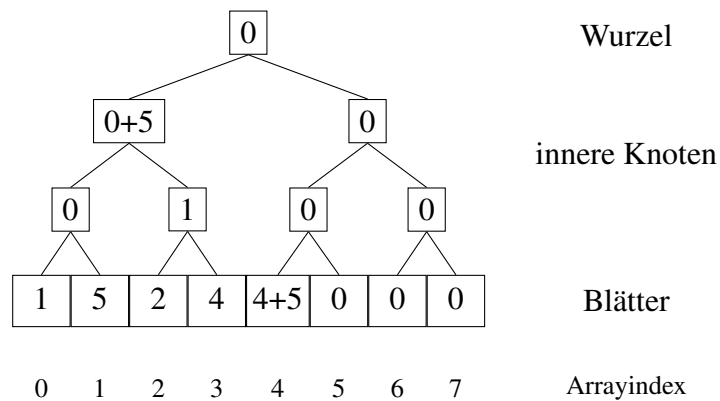


Angenommen, auf die Werte mit den Indizes 2 bis 4 soll die Zahl 1 addiert werden. Es wird nun die minimal mögliche Menge an inneren Knoten und Blättern gesucht, sodass auf den Pfaden von genau den entsprechenden Blättern zur Wurzel je genau eine 1 addiert wird. Speziell sieht der Baum nach dieser Operation `add` also wie folgt aus:



Auf den Pfaden von Blatt 2, Blatt 3 und Blatt 4 zur Wurzel wurde je genau eine 1 hinzuaddiert.

Jetzt sollen auf die Werte mit den Indizes 1 bis 5 der Wert 5 addiert werden. Nach diesem add sieht der Baum wie folgt aus:



Soll nun der Wert w mit dem Index 2 berechnet werden, so werden alle Werte auf dem Pfad von dem entsprechenden Blatt zur Wurzel aufaddiert:

$$w = 0 + 5 + 1 + 2 = 8$$

Ohne Pseudocode anzuführen ist es leicht einzusehen, dass zum Berechnen eines Wertes genau ein Pfad entlang gegangen werden muss. Auch müssen nur konstant viele Pfade betrachtet werden, soll eine Konstante auf ein Teilarray aufaddiert werden. Da ein Pfad in einem vollständigen Binärbaum $O(\log n)$ lang ist, benötigen das Abfragen eines Wertes und das Addieren einer Konstante also je $O(\log n)$ Operationen.

Die gesuchte Datenstruktur ist also gefunden. Jetzt muss der Algorithmus noch so umformuliert werden, dass auch Binärbäume eine Anwendung finden können. Dazu soll noch einmal die Rekursionsgleichung genauer betrachtet werden, auf der der bisherige Algorithmus basiert:

$$A_F(N; k) = \sum_{m=0}^{F(b_1)} A_{F'}(N-m; k-1)$$

Anstatt ein größeres Problem auf kleinere zurückzuführen, sollen nun aus gelösten Teilproblemen größere „zusammgebaut“ werden. Anstatt also mit der Berechnung von $A_F(N; k)$ zu

beginnen, soll nun mit der Berechnung von $A_{\{\}}(0;0)$ begonnen werden.

Aus der Rekursionsgleichung folgt, dass ein Term $A'_F(N,k)$ als Summand in den Termen

$$A_{\{b_1;F(b_1)\} \cup F'}(N+m, k+1), 0 \leq m \leq F(b_1)$$

auftaucht. Aus allen für ein festes k auftretenden Termen $A'_F(N,k)$ können also offenbar alle für $k+1$ auftretende Terme „zusammengebaut“ werden. Dies wird in Algorithmus 5 umgesetzt. Diese Art der Berechnung wird *Dynamische Programmierung* genannt (vgl. Aufgabe 1), kurz: DP. (Analog können auch die Fibonacci-Zahlen mit Dynamischer Programmierung berechnet werden.)

Algorithmus 5 Dynamische Programmierung

Eingabe: N , k , Array an Fassungsvermögen $F[0..(k-1)]$
 Ausgabe: $AF(N,k)$

```
ANZAHL3( N, k, F )
  dp[ 0..N ][ 0..k ] := 0
  dp[ 0 ][ 0 ] := 1
  for i := 0..(k-1) do
    for j := 0..N do
      for m := 0..min( N-j, F[ i ] ) do
        dp[ i+1 ][ j+m ] := dp[ i+1 ][ j+m ]+dp[ i ][ j ]
  return dp[ N ][ k ]
```

Die Laufzeit hat sich durch diese andere Art der Berechnung nicht geändert. Auf alle Elemente des Arrays $dp[i+1]$ zwischen j und $j + \min(N-j, F[i])$ wird jedoch ein konstanter Wert $dp[i][j]$ aufaddiert. Dies ist genau diejenige Operation *add*, für die Binärbäume besonders geeignet sind. Verwendung von Binärbäumen führt zu Algorithmus 6.

Algorithmus 6 Dynamische Programmierung mit Binärbäumen

Eingabe: N , k , Array an Fassungsvermögen $F[0..(k-1)]$
 Ausgabe: $AF(N,k)$

```
ANZAHL4( N, k, F[ 0..(k-1) ] )
  dp[ 0..N ] //Array von N+1 Binärbäumen
  dp[ 0 ][ 0 ] := 1
  for i := 0..(k-1) do
    for j := 0..N do
      dp[ i+1 ].add( dp[ i ][ j ], j, min( N, F[i] + j ) )
  return dp[ N ][ k ]
```

Für die Laufzeit dieses neuen Algorithmus ergibt sich $O(k \cdot n \log n)$. Damit lässt sich die Funktion A für Werte von $N \cdot k \leq 2 \cdot 10^6$ auswerten. Dieser Algorithmus ist für die zu lösenden Beispiele also definitiv schnell genug.⁷

Berechnet man die Ergebnisse für die letzten beiden Beispiele, so stellt man fest, dass diese Ergebnisse größer als eine 64-Bit Zahl sind; das letzte ist sogar größer als eine 128-Bit Zahl.

⁷Mit einigen geschickten Umformungen kann sogar noch eine Laufzeit von $O(k \cdot n)$ erreicht werden, bei der keine Binärbäume benötigt werden.

Bei der Implementierung muss dies berücksichtigt werden. In manchen Programmiersprachen sind Operationen (speziell die Addition) auf und mit beliebig großen Zahlen bereits standardmäßig vorhanden, z.B. in Java mit der Klasse `BigInteger`. In anderen Sprachen wie C oder C++ müssen diese Operationen selbst implementiert werden. Aufgrund dieser Tatsache sind auch die asymptotischen Laufzeiten der Algorithmen mit Vorsicht zu genießen, da bei deren Ermittlung davon ausgegangen wurde, dass das Ergebnis in eine einzige Variable passt.

3.2 Beispiele

Die Ergebnisse für die vorgegebenen Beispiele werden in Tabelle 2 vorgestellt. Um dabei gleich einen Eindruck zu vermitteln, für welche Eingaben welcher Algorithmus noch „gut genug“ ist, wird jeweils der einfachste Algorithmus angegeben, mit dem die Ergebnisse in angemessener Zeit berechnet werden können. Um einen Eindruck für die Größe „schnell“ bzw. „gut genug“ zu wecken, sind auch bei jedem Beispiel die Laufzeiten auf einem konkreten Rechner⁸ angegeben.

DP mit Binärbäumen

Die Lösung der nun folgenden Beispiele ist nicht gefordert gewesen. Die Laufzeiten beziehen sich bei diesen Beispielen auf eine Implementierung mit Binärbäumen, im Gegensatz zu den vorgegebenen Beispielen.

Beispiel *1

Eingabe	Ausgabe
100000 20	5913068692201615816980737340486279922
1000 2000 3000 4000 5000 6000	6645376960764091870332553077346193226
7000 8000 9000 10000 11000	Used 4.819s.
12000 13000 14000 15000	
16000 17000 18000 19000 20000	

Beispiel *2

Eingabe	Ausgabe
50000 50	1
1000 1000 1000 1000 1000 1000	Used 5.743s.
1000 1000 1000 1000 1000	
1000 1000 1000 1000 1000	
1000 1000 1000 1000 1000	
1000 1000 1000 1000 1000	
1000 1000 1000 1000 1000	
1000 1000 1000 1000 1000	
1000 1000 1000 1000 1000	
1000 1000 1000 1000	

⁸Intel Core i5-6200U, 8GB RAM

Beispiel	Algorithmus	Eingabe	Ausgabe
Aufgabe (1)	Brute Force	7 2 3 5	2 Used 0.000s.
Aufgabe (2)	Brute Force	7 3 2 3 4	6 Used 0.000s.
1	Brute Force	5 3 2 4 4	13 Used 0.000s.
2	Brute Force	10 3 5 8 10	48 Used 0.000s.
3	Vermeidung doppelter Berechnungen / Dynamisches Programmieren	30 20 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	6209623185136 Used 0.000s.
4	Vermeidung doppelter Berechnungen / Dynamisches Programmieren	3000 11 5 100 100 200 500 500 600 800 800 1000 1000	743587168174197919278525 Used 0.686s.
5	Vermeidung doppelter Berechnungen / Dynamisches Programmieren	2000 18 100 110 130 170 180 200 220 220 230 250 280 300 340 350 350 370 380 390	423761833216813064373 4395335220863408628 Used 0.490s.

Tabelle 2: Ergebnisse für die vorgegebenen Beispiele

Beispiel *3

Eingabe	Ausgabe
50000 50	2990009562009114418295515038428558
10000 10000 10000 10000 10000	6934305696949854368543646815132907
10000 10000 10000 10000	3018227140053249774635772948937116
10000 10000 10000 10000	9285298139762242009664229888161917
10000 10000 10000 10000	67737123633427920557425997455126
10000 10000 10000 10000	Used 9.730s.
10000 10000 10000 10000	
10000 10000 10000 10000	
10000 10000 10000 10000	
10000 10000 10000 10000	
10000 10000 10000 10000	
10000 10000 10000 10000	
10000 10000 10000 10000	
10000 10000 10000 10000	
10000 10000 10000 10000 10000	

3.3 Bewertungskriterien

- Funktionsweise und Korrektheit des Verfahrens müssen nachvollziehbar erläutert bzw. begründet werden. Eine formale Begründung wird aber nicht erwartet.
- Das verwendete Verfahren darf nicht allzu aufwändig sein: Es sollte alle vorgegebenen Beispiele (korrekt) in wenigen Sekunden berechnen können. Ein reiner Brute-Force-Ansatz schafft in der Regel nur die Beispiele 1 und 2; in diesem Fall werden zwei Punkte abgezogen. Schafft ein ineffizientes Verfahren durch Optimierungen mehr als die Vorgabebeispiele 1 und 2 (aber nicht alle fünf Beispiele), wird nur ein Punkt abgezogen. Sollten schlechtere Laufzeiten aufgrund langsamer Operationen auf großen Zahlen entstehen, so ist dies akzeptabel; insbesondere wenn für die Implementierung eine Sprache gewählt wurde, die solche Operationen nicht standardmäßig unterstützt.
- Das Verfahren bzw. seine Implementierung soll korrekte Ergebnisse liefern. Aus der Aufgabenstellung wird klar, dass Behälter auch leer bleiben dürfen; das muss berücksichtigt sein.
- Bei dieser Aufgabe ist die Laufzeit des Verfahrens offensichtlich kritisch. Das Thema Laufzeit sollte explizit (theoretische Abschätzung oder Angabe konkreter Laufzeiten) oder implizit (Versuch von Optimierungen oder Realisierung eines ausreichend schnellen Verfahrens) behandelt sein.
- Die Lösungen zu allen vorgegebenen Beispielen 1 bis 5 sollten angegeben sein. Die Beispiele aus der Aufgabenstellung sind nicht gefordert.
- Zusätzliche Beispiele wären schön, insbesondere wenn sie Schwächen des verwendeten Verfahrens demonstrieren – speziell bei Optimierungen, die nur für manche Eingaben Laufzeitvorteile bieten oder für manche Eingaben falsche Ergebnisse liefern. Erwartet werden zusätzliche Beispiele aber nicht.

Aufgabe 4: Schlüssellöcher

4.1 Anzahl verdrehsicherer Schlüssel

Zunächst wollen wir festhalten, wie die Anzahl aller möglichen Schlüssel von der Anzahl der verfügbaren Codepunkte abhängt.

Wir müssen uns bewusst werden, dass wir für jeden Codepunkt genau zwei verschiedene Zustände kennen: ein Codepunkt kann ein Loch oder eben kein Loch sein. Aus dieser Tatsache kann man nun einfach herleiten, wie viele unterschiedliche Schlüssel es für eine gegebene Anzahl n von Codepunkten gibt.

Es gibt genau zwei verschiedene Schlüssel mit nur einem Codepunkt: den mit Loch und den ohne. Für jeden weiteren verfügbaren Codepunkt verdoppelt sich die Anzahl der möglichen Schlüssel, da es die Anzahl der vorher verfügbaren Schlüssel nun zweimal gibt: einmal, sodass bei allen vorherigen Schlüsseln der neu hinzugefügte Codepunkt ein Loch ist, und einmal, sodass der neue Codepunkt kein Loch ist.

Wir erkennen also: die Anzahl aller möglichen unterschiedlichen Schlüssel hängt wie folgt von der Anzahl n der verfügbaren Codepunkte ab:

$$\underbrace{2 \cdot 2 \cdot \dots \cdot 2 \cdot 2}_{n\text{-mal}} = 2^n \quad (2)$$

Für unsere 5×5 Codepunkte ist die Anzahl der möglichen Schlüssel also $2^{25} = 33.554.432$.

Verdrehsicherheit durch Festsetzen zweier Eckpunkte

Durch das Festsetzen der beiden oberen Eckpunkte erhält man zwar verdrehsichere Schlüssel, man schränkt jedoch auch die Menge der verfügbaren Schlüssel ein. Dadurch, dass der Zustand zweier Codepunkte bereits festgelegt ist, kann man nur noch bei 23 Codepunkten deren Zustand wählen. Man erhält somit nur noch $2^{23} = \frac{2^{25}}{4} = 8.388.608$ Schlüssel.

Verdrehsicherheit durch Symmetrie

Noch weiter schränkt man die Anzahl der verfügbaren Schlüssel ein, indem man an der mittleren Spalte eine Achsenspiegelung vornimmt, so dass es egal ist, mit welcher Seite nach oben ein Schlüssel eingelegt wird.

Durch die Spiegelung sind nur noch 15 Codepunkte frei wählbar: dies erkennt man, wenn man sich bewusst macht, dass man nur die 3 linken Spalten frei wählen kann: die 4. Spalte von links wird bereits durch die 2. Spalte von links festgelegt und die 5. Spalte von links dementsprechend von der 1. Spalte. Somit bleiben $3 \cdot 5 = 15$ Codepunkte frei zu wählen. Damit ergibt sich die Anzahl aller verfügbaren Schlüssel zu $2^{15} = 32.768$.

4.2 Algorithmisches Erzeugen möglichst unterschiedlicher Schlüssel

Unterschiedlichkeit zweier Schlüssel

Als Maß für die Unterschiedlichkeit zweier Schlüssel wollen wir die Anzahl der unterschiedlichen Codepunkte verwenden. Je unterschiedlicher zwei Schlüssel zueinander sind, desto größer wird diese Maßzahl. Die Unterschiedlichkeit liegt immer im Intervall $[0; 25] \subset \mathbb{N}_0$. Wir schreiben $|x - y|$ für die Unterschiedlichkeit zweier Schlüssel x und y .

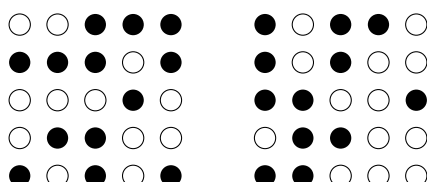


Abbildung 16: Zwei Schlüssel mit Unterschiedlichkeit 11

Einen Schlüssel kann man auch als Binärzahl bzw. Bitfolge auffassen: man beginnt in der unteren rechten Ecke des Schlüssels und schreibt jeden Codepunkt entweder als 0, wenn es sich um ein Loch handelt oder als 1, wenn es sich nicht um ein Loch handelt. Dann fährt man mit dem Codepunkt an der zweiten Stelle von rechts unten fort usw., bis man eine 25-stellige Binärzahl erhält. Das oben beschriebene Maß für die Unterschiedlichkeit zweier Schlüssel entspricht dann genau der *Hamming-Distanz* zwischen den beiden den Schlüsseln entsprechenden Binärzahlen, die als Anzahl der unterschiedlichen Bit-Stellen definiert ist. Hier sind die beiden Schlüssel aus Abb. 16 als Binärzahlen untereinander geschrieben (zuerst der linke Schlüssel); die unterschiedlichen Bits sind markiert:

```

1010100110010001011111100
0001100110100110010101101

```

Bewertung der Unterschiedlichkeit einer Menge von Schlüsseln

Unser Ziel ist es, $n \in \mathbb{N}$ möglichst unterschiedliche Schlüssel zu generieren. Wir haben zwar oben bereits ein Maß für die Unterschiedlichkeit zweier Schlüssel definiert, jedoch ist noch nicht klar, was wir unter der Unterschiedlichkeit einer Menge von Schlüsseln verstehen.

Es gibt verschiedene Möglichkeiten, die Unterschiedlichkeit einer Menge von Schlüsseln zu definieren. Intuitiv bietet sich an, das arithmetische Mittel über die Unterschiedlichkeiten aller möglichen Schlüsselpaare (davon gibt es $|S|(|S| - 1)/2$) aus der betrachteten Schlüsselmenge zu betrachten:

$$U(S) = \frac{2}{|S|^2 - |S|} \sum_{x,y \in S, x \neq y} |x - y| \quad (3)$$

Beide Schlüsselmengen in Abb. 17 haben nach obiger Definition die durchschnittliche Unterschiedlichkeit 6. Jedoch wollen wir Schlüsselmengen wie die in Abb. 17b vermeiden, da es zwei Schlüssel gibt, die einander stark ähnlich sind. Dazu können wir zusätzlich die durchschnittliche Abweichung der Unterschiedlichkeiten der Schlüsselpaare vom Mittelwert betrachten. In Abb. 17a ist diese 0, da keine Paar-Unterschiedlichkeit vom Mittelwert abweicht. In Abb. 17b



Abbildung 17: Zwei Schlüsselmengen mit durchschnittlicher Unterschiedlichkeit 6.

hingegen ist die durchschnittliche Abweichung $\frac{8}{3} = 2,\bar{6}$. Eine gute Schlüsselmenge hat also eine große durchschnittliche Unterschiedlichkeit, aber eine geringe Abweichung der Unterschiedlichkeiten vom Mittelwert.

$$A(S) = \frac{2}{|S|^2 - |S|} \sum_{x,y \in S, x \neq y} ||x - y| - U(S)| \quad (4)$$

Da wir nun zwei Werte $U(S)$ und $A(S)$ zur Verfügung haben, um zu bestimmen, welche zweier Schlüsselmengen die bessere ist, müssen wir noch eine Verknüpfung der beiden Werte herstellen. Wir benutzen

$$B(S) = \frac{A(S)}{U(S)} \quad (5)$$

um die beiden Maße zu einer einzigen Zahl zu verschmelzen. Eine Schlüsselmenge S_1 ist genau dann besser als eine andere Schlüsselmenge S_2 , wenn $B(S_1) < B(S_2)$.

Ein alternatives Maß für Schlüsselmengen S ist die kleinste Unterschiedlichkeit aller Schlüsselpaare, also $\min_{x,y \in S, x \neq y} |x - y|$; je größer dieser Wert, desto größer sind eben auch die paarweisen Unterschiede.

Einfacher Algorithmus

Es ist nicht schwer, einen Algorithmus anzugeben, der n möglichst unterschiedliche Schlüssel berechnet. Algorithmus 7 geht wie folgt vor: wenn n Schlüssel gefunden werden sollen, die zueinander möglichst unterschiedlich sein sollen, dann betrachtet er einfach jede n -elementige Teilmenge der Menge aller Schlüssel, bewertet diese nach Gleichung 5, vergleicht dies mit der bisherigen besten Lösung und behält die neue Lösung, falls diese besser ist. Am Ende erhält man so eine der möglicherweise mehreren besten Lösungen.

Algorithmus 7 hat allerdings ein Problem. Betrachtet man die Anzahl der Schleifendurchläufe, die die innersten Schleifen machen, wird schnell klar, dass dieser Algorithmus nicht dazu geeignet ist, vor dem Ende des Universums zu einer Lösung zu kommen (außer bei sehr wenigen Schlüsseln mit sehr wenigen Codepunkten). Die Anzahl der inneren Schleifendurchläufe ergibt sich zu:

$$x(n, c) = \binom{2^c}{n} * \binom{n}{2} = \frac{(2^c)!}{(2^c - n)! * (n - 2)! * 2} \quad (6)$$

Besser verstehen lässt sich diese Formel in folgender Form:

$$x(n, c) = \left\{ \prod_{i=0}^{n-1} (2^c - i) \right\} * \frac{1}{2 * (n - 2)!} \quad (7)$$

Dabei ist c die Anzahl der Codepunkte, die zur Verfügung stehen, und n die Anzahl der zu berechnenden Schlüssel.

Algorithmus 7 Ein Algorithmus, der garantiert eine beste Lösung findetEingabe: n , die Anzahl der zu generierenden SchlüsselAusgabe: n Schlüssel mit möglichst großer Unterschiedlichkeit $besteLoesung \leftarrow null$ $besteBewertung \leftarrow null$ **for** jede n -elementige Teilmenge der Menge aller Schlüssel **do** $distanz \leftarrow 0$ **for** jede 2-elementige Teilmenge $\{x, y\} | x \neq y$ der ausgewählten Untermenge **do** $distanz \leftarrow distanz + |x - y|$ **end for** $mittel \leftarrow \frac{distanz}{n}$ $abweichung \leftarrow 0$ **for** jede 2-elementige Teilmenge $\{x, y\} | x \neq y$ der ausgewählten Untermenge **do** $abweichung \leftarrow abweichung + ||x - y| - mittel|$ **end for** $mittlereAbweichung \leftarrow \frac{abweichung}{n}$ $bewertung \leftarrow (mittel, mittlereAbweichung)$ **if** $besteLoesung = null$ **oder** $bewertung$ ist besser als $besteBewertung$ **then** $besteLoesung \leftarrow$ ausgewählte Untermenge $besteBewertung \leftarrow bewertung$ **end if****end for****return** $besteLoesung$

Verlangt man bei einem 3×3 -Feld (also $c = 9$) nur $n = 5$ Schlüssel, so müsste dieser Algorithmus bereits

$$1.457.703.663.037.440 = 1,45770366303744 \cdot 10^{15}$$

Schleifendurchläufe machen. Geht man davon aus, dass ein moderner Prozessor 1 Million dieser Schleifendurchläufe pro Sekunde schafft, dann benötigt der Computer schon über 46 Jahre, um die beste Lösung für $c = 9$ und $n = 5$ zu finden. Aus der Formel kann man außerdem sehen, dass die Situation für größere c und größere n nur immer schlimmer wird.

Kann man überhaupt einen schnellen optimalen Algorithmus finden?

Wir wollen uns jetzt der Frage widmen, ob man überhaupt einen effizienten Algorithmus angeben kann, der garantiert eine der möglichst unterschiedlichen Schlüsselmengen der gegebenen Mächtigkeit n findet. Wir haben bereits einen Algorithmus angegeben, der garantiert das beste Ergebnis findet. Allerdings mussten wir feststellen, dass dessen Laufzeit viel zu groß ist; selbst für sehr kleine Eingaben liefert der Algorithmus bereits länger, als wir jemals warten können.

Sieht man sich ein wenig in der Literatur um, so findet man unser Problem, etwas anders aufgefasst, in der Kodierungstheorie wieder. Es wurde ja schon beschrieben, dass unsere Schlüssel Binärzahlen entsprechen. Für uns interessant sind insbesondere die Erkenntnisse, die bereits über die kodierungstheoretische Funktion $A_2(n, d)$ gewonnen wurden. $A_2(n, d)$ gibt die Anzahl

der Binärzahlen der festen Länge n an, die sich zueinander alle um genau d Stellen unterscheiden. Die genauen Werte von $A_2(n, d)$ wurden für viele Parameter n, d noch nicht herausgefunden. Es ist also eine offene Frage, ob es einen effizienten Algorithmus gibt, der diese Werte berechnen kann. In Tabelle 3 sind alle bekannten Werte für $A_2(25, d)$ aufgelistet, teilweise nur mit oberen und unteren Grenzen für den Wert der Funktion (Quelle: <http://www.win.tue.nl/~aeb/codes/binary-1.html>). Man kann klar erkennen, dass die Anzahl der verfügbaren Schlüssel mit der geforderten Unterschiedlichkeit stark abnimmt. So gibt es schon nur noch 4 Schlüssel, die sich alle zueinander um 15 Stellen unterscheiden.

Unterschiedlichkeit	Anzahl der Schlüssel		
3	2^{20} (= 1048576) bis 1198368	9	384 bis 836
4	2^{19} (= 524288) bis 599184	10	192 bis 466
5	32768 bis 84260	11	64 bis 96
6	16384 bis 47538	12	52 bis 55
7	4104 bis 9275	13	14
8	4096 bis 5421	14	8
		15, 16	4

Tabelle 3: Anzahl der Schlüssel mit 25 Codepunkten bei bestimmter, fester Unterschiedlichkeit.

Wenn wir also gerade nicht die Zeit haben, dieses offene Problem zu lösen, dann können wir vielleicht eine andere Lösung finden, die zwar nicht garantiert eine der möglicherweise mehreren besten Lösungen findet, aber dafür in kurzer Zeit eine gute Lösung bestimmt.

Ein genetischer Algorithmus

Für unser Problem eignet sich ein sogenannter genetischer Algorithmus. Ein genetischer Algorithmus ist eine *Metaheuristik*: das sind Verfahren, die Informatiker auf ihre Probleme immer dann anwenden, wenn sie nicht wissen, wie sie sonst in annehmbarer Zeit zu einer möglichst guten Lösung kommen können.

Allgemein funktioniert ein genetischer Algorithmus wie folgt: Zu Beginn gibt es eine Anfangslösung, die zum Beispiel zufällig erzeugt werden kann. Dann versucht der Algorithmus, Teile der Lösung auszutauschen, während er die anderen Teile behält, und überprüft anschließend, ob sich die Lösung verbessert hat. Ist dies der Fall, dann behält er die Lösung; ansonsten verwirft er sie und fährt mit dem Ausprobieren fort, bis die Lösung eine gewisse Güte erreicht hat, bei welcher der Algorithmus anhält und die momentane Lösung zurückgibt.

Genau mit diesem Prinzip wollen wir nun versuchen, einen Algorithmus auf die Beine zu stellen, der zwar nicht nachweislich eine optimale Lösung, dafür aber eine sehr gute Lösung in geringer Zeit findet. Algorithmus 8 geht wie folgt vor: zunächst erzeugt er n zufällige, unterschiedliche Schlüssel. Dann versucht er, die Schlüsselmenge schrittweise zu verbessern, bis der Nutzer die Schlüsselmenge für gut genug befindet. Die Schlüsselmenge verbessert der Algorithmus wie folgt: er wählt zufällig einen Schlüssel aus der Menge aus, ersetzt ihn durch einen neuen, zufällig erzeugten Schlüssel, der noch nicht in der Schlüsselmenge vorhanden ist, und überprüft dann, ob die Schlüsselmenge nun besser oder schlechter als vorher ist. Falls sie besser ist, behält er die neue Schlüsselmenge. Ansonsten ersetzt er den neu hinzugefügten Schlüssel mit dem vorherigen, alten Schlüssel.

Algorithmus 8 Ein genetischer Algorithmus, der die Lösung langsam verbessert

Eingabe: n , die Anzahl der zu generierenden Schlüssel
Ausgabe: n Schlüssel mit möglichst großer Unterschiedlichkeit
 $S \leftarrow n$ zufällige, unterschiedliche Schlüssel
while Benutzer hat das Programm nicht beendet **do**
 $m \leftarrow B(S)$
 ersetze einen zufälligen Schlüssel $\in S$ durch einen zufällig generierten Schlüssel $\notin S$
 $n \leftarrow B(S)$
 if $n \geq m$ **then**
 mache die Veränderung rückgängig
 else
 gib die neue beste Lösung aus
 end if
end while
return S

4.3 Beispielausgaben**7 Schlüssel**

Hinweis: Diese Ausgabe ist als einzige nicht gekürzt.

Bitte geben Sie die Anzahl der zu berechnenden Schlüssel ein:

7

Druecken Sie nun irgendeine Taste, um mit der Berechnung zu beginnen!

```
Bessere Loesung gefunden: 12.380952380952381 +-1.3151927437641722
Bessere Loesung gefunden: 12.095238095238095 +-1.2426303854875282
Bessere Loesung gefunden: 12.19047619047619 +-1.1337868480725624
Bessere Loesung gefunden: 12.571428571428571 +-1.1020408163265307
Bessere Loesung gefunden: 12.952380952380953 +-1.0158730158730158
Bessere Loesung gefunden: 12.952380952380953 +-0.9206349206349206
Bessere Loesung gefunden: 12.857142857142858 +-0.8299319727891156
Bessere Loesung gefunden: 13.142857142857142 +-0.7755102040816324
Bessere Loesung gefunden: 13.047619047619047 +-0.6439909297052153
Bessere Loesung gefunden: 12.952380952380953 +-0.5532879818594103
Bessere Loesung gefunden: 12.952380952380953 +-0.5487528344671201
Bessere Loesung gefunden: 13.047619047619047 +-0.5442176870748299
Bessere Loesung gefunden: 13.047619047619047 +-0.45351473922902485
```

Schrittweise Verbesserung angehalten und bestes Ergebnis ausgegeben.

```
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 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 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
- 0 0 0 -
0 0 - - -

```

Die erzeugten Schlüssel unterscheiden sich durchschnittlich um 13.047619047619047 \pm 0.45351473922902485.

20 Schlüssel

Bitte geben Sie die Anzahl der zu berechnenden Schlüssel ein:
20

Druecken Sie nun irgendeine Taste, um mit der Berechnung zu beginnen!

Bessere Loesung gefunden: 12.278947368421052 \pm 2.066703601108033

Bessere Loesung gefunden: 12.226315789473684 \pm 2.027700831024931

...

Bessere Loesung gefunden: 13.0 +-0.7157894736842105

Bessere Loesung gefunden: 13.0 +-0.7052631578947368

Schrittweise Verbesserung angehalten und bestes Ergebnis ausgegeben.

```
- 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 - -
- 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
```

Die erzeugten Schluessel unterscheiden sich durchschnittlich um 13.0 +-0.7052631578947368.

1000 Schlüssel

Bitte geben Sie die Anzahl der zu berechnenden Schluessel ein:

1000

Druecken Sie nun irgendeine Taste, um mit der Berechnung zu beginnen!

Bessere Loesung gefunden: 12.494576576576577 +-2.015851004950905

Bessere Loesung gefunden: 12.494816816816817 +-2.0157276239602973

Bessere Loesung gefunden: 12.494622622622623 +-2.0155969731974235

...

Bessere Loesung gefunden: 12.51171971971972 +-1.9464158962565623

Bessere Loesung gefunden: 12.511751751751751 +-1.9463656664070996

Bessere Loesung gefunden: 12.511723723723724 +-1.9463113061409585

Bessere Loesung gefunden: 12.5116996996997 +-1.9463022867192037

Bessere Loesung gefunden: 12.511583583583583 +-1.9462730583717536

Schrittweise Verbesserung angehalten und bestes Ergebnis ausgegeben.

```
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 0 - - 0
- - 0 - 0
- - 0 - -
```

```
- - 0 0 0
- 0 - 0 -
- 0 - 0 0
- 0 - - 0
- 0 - - 0
```

Die erzeugten Schlüssel unterscheiden sich durchschnittlich um $12.511583583583583 \pm 1.9462730583717536$.

4.4 Verbesserungsmöglichkeiten

Man kann das angegebene Verfahren sicher noch weiter verbessern. Hier sind einige Ideen:

Andere Metaheuristiken Es gibt noch weitere Metaheuristiken, mit denen man versuchen kann, Lösungen für komplexe Probleme effizient anzunähern. Ein Beispiel ist das sogenannte Simulated Annealing (simulierte Abkühlung), bei der die Wahrscheinlichkeit, ein gutes Ergebnis zu verwerfen, mit der Zeit immer geringer wird, bis das Ergebnis schließlich fest steht. Um dieses Verfahren benutzen zu können, müsste man eine gute Zeit finden, für die die Simulation laufen soll. Diese wäre abhängig von der Anzahl der zu findenden Schlüssel.

Mehrere Schlüssel austauschen Statt immer nur einen Schlüssel auszutauschen, könnte man mehrere gleichzeitig austauschen.

Auswahl des auszutauschenden Schlüssels Bei dieser Auswahl könnte man eine Heuristik statt den Zufall einsetzen. Zum Beispiel könnte man bestimmen, welcher Schlüssel die

geringste Unterschiedlichkeit zu allen anderen Schlüsseln hat, und diesen so lange durch einen anderen zufälligen Schlüssel ersetzen, bis man ein besseres Ergebnis hat.

4.5 Bewertungskriterien

- Die Fragen in Teil 1 sollten korrekt beantwortet sein.
- Ein Unterschiedlichkeitsmaß soll nicht nur für Schlüsselpaare, sondern auch für eine ganze Schlüsselmenge angegeben sein. Alternativ muss klar werden, wie evtl. algorithmisch mit Hilfe eines Maßes für Schlüsselpaare eine Schlüsselmenge bewertet wird (womit dann zumindest implizit ein Maß für Schlüsselmenge gegeben ist).
- Ein nahe liegendes, einfaches Unterschiedlichkeitsmaß für eine Schlüsselmenge ist der Mittelwert der Paar-Unterschiedlichkeiten. Dieses Maß verhindert aber nicht, dass die Schlüsselmenge (nahezu) identische Schlüssel enthält. Es sollte behandelt worden sein, dass derartig einfache Unterschiedlichkeitsmaße für Mengen Nachteile haben können: indem diese Nachteile explizit angegeben oder durch den Algorithmus zur Schlüsselbestimmung kompensiert werden.
- Für das Verfahren zur Bestimmung geeigneter Schlüsselmenge soll nachvollziehbar erläutert bzw. begründet sein, wie es die Schlüsselmenge anhand des gewählten Maßes bestimmt.
- Das Verfahren darf nicht zu aufwändig sein, muss aber dennoch ordentliche Ergebnisse liefern. Alle Verfahren, die diese Anforderung einigermaßen erfüllen und z.B. auch für $N = 1000$ brauchbare Unterschiedlichkeiten liefern, sind akzeptabel. Ein häufig verwendetes, einfaches Verfahren liefert zu geringe Unterschiedlichkeiten für $N = 1000$: Für N Schlüssel werden die Binärdarstellungen der Zahlen $1, \dots, N$ in einheitlicher, minimaler Länge (z.B. 10 für $N = 1000$) aufgelistet und auf die Länge 25 „aufgeblasen“, und zwar durch Vervielfachung der vorhandenen Bits und geeignetes Auffüllen.
- Die Aufgabenstellung legt nahe, dass das Verfahren sich auf symmetrische Muster beschränken könnte; das würde bei einem einfachen Verfahren außerdem Laufzeit sparen. Gefordert ist das aber nicht.
- Für $N = 7$ und $N = 20$ sollen die Lochmuster angegeben sein (bei $N = 20$ sind Auslassungen akzeptabel). Für diese Werte und für $N = 1000$ müssen Aussagen über die Unterschiedlichkeitswerte erzeugter Schlüsselmenge gemacht werden. Eine Ausgabe der Schlüssel für $N = 1000$ ist nicht nötig (und in der Dokumentation ohnehin nur in kleinen Auszügen sinnvoll).

Aufgabe 5: Groker

5.1 Lösungsidee

Das Spiel „Groker“ hat etwas geradezu Unsportliches: Nicht der höchste Einsatz gewinnt, sondern ein genügend kleinerer. Ist der kleinere Einsatz aber nicht klein genug, hat dann doch wieder der höhere Einsatz einen Vorteil. Das ist vertrackt, was wiederum die beste Voraussetzung dafür ist, als Aufgabe im Bundeswettbewerb Informatik zu Ehren zu kommen.

Wenn im Folgenden über mögliche Spielstrategien gesprochen wird, reden wir vom Spieler S (diese Rolle soll dann die zu programmierende KI übernehmen) und vom Gegner G . In einer Runde fallen die Einsätze e_S und e_G ; das sind einfach Zahlen (die Chips können wir ignorieren). Nach der Runde verzeichnen die Gewinnhaufen der beiden die Zuwächse $z_S \in \{0, e_S\}$ und $z_G \in \{0, e_G\}$. Entscheidend ist, wer – Spieler oder Gegner – in einer Runde den Gewinn $|z_S - z_G|$ einstreichen und damit die Runde für sich entscheiden kann. Da Groker „gedächtnislos“ ist (frühere Runden haben keinen Einfluss darauf, wer die aktuelle Runde für sich entscheiden kann), haben Rundengewinne den entscheidenden Einfluss darauf, wer ein Spiel gewinnt.

Zunächst kann man sich überlegen, dass hohe Einsätze, wie sie etwa im Beispiel der Aufgabenstellung vorkommen, für den Spieler nicht sinnvoll sind. Sie machen es dem Gegner nur leicht, deutlich kleinere Einsätze abzugeben und die Runde zu gewinnen. Die Aufgabenstellung weist bereits darauf hin, dass Einsätze aus $\{1, \dots, 6\}$ sicher sind; sie kann man auf jeden Fall als Zuwachs einheimsen. Der Gegner kann diese sicheren Einsätze, wie ebenfalls angedeutet, zwar überbieten; aber viel mehr als 11 sollte er nicht setzen. Denn: Liegt der Einsatz des Spielers über dem des Gegners, kann der Spieler einen Gewinn von höchstens 5 erreichen. Setzt der Spieler aber 12 (oder mehr), hat der Gegner die Chance auf einen Gewinn von (mindestens) 6.

Strategien

Im Folgenden werden kurz einige Strategien bzw. Klassen von Spielstrategien vorgestellt und ihre Nachteile angesprochen. Dabei werden die diesen Klassen zuzuordnenden Strategien immer komplizierter und aufwändiger zu realisieren. Die im Turnier erfolgreichsten KIs haben mit Strategien aus der letzten Klasse operiert.

Strategien mit festem Setzverhalten

Konstant Diese Strategie setzt konstant 6 (oder eine andere Zahl). Vorteile bei 6: kein Verlust. Nachteile: sehr leicht zu überwinden, indem immer 5 mehr oder 6 weniger gesetzt werden.

Immer 5 mehr Diese Strategie setzt immer 5 mehr, als der Gegner im letzten Zug gesetzt hat. Vorteile: gewinnt gegen konstante Strategien. Nachteile: leicht zu überwinden, indem man immer 1 weniger oder 10 mehr als im eigenen letzten Zug setzt.

Immer 6 weniger Diese Strategie setzt immer 6 weniger als der Gegner im letzten Zug gesetzt hat; falls der Gegner aber unter 7 setzt, setzt sie 5 mehr. Vorteile: gewinnt gegen konstante Setzverfahren. Nachteile: leicht zu überwinden, indem man immer 1 weniger als im letzten Zug setzt.

Durchschnitt Diese Strategie berechnet immer den Durchschnitt der Einsätze des Gegners und setzt immer etwas mehr oder deutlich weniger. Vorteile: gewinnt meist gegen Zufall (s.u.) und Konstant. Nachteile: kann man überwinden, indem man abwechselnd viel und wenig setzt.

Strategien mit zufälligem Setzverhalten

Zufall Diese Strategie setzt eine zufällige Zahl zwischen a und b . Dabei gilt: (1) $a \leq 6$, da man sonst immer 6 weniger als a setzen könnte, und (2) $b = a + 10$, da man sonst mit 5 mehr als a gewinnen würde. Vorteile: für adaptive Algorithmen (s.u.) ist es schwer festzustellen, dass es sich um einen Zufallsalgorithmus handelt; falls erkannt, ist es relativ schwer, den besten Einsatz zu bestimmen. Nachteile: es gibt immer eine Zahl, mit der man im Durchschnitt gewinnt.

Zufall 2 Diese Strategie setzt Zahlen zwischen a und b nach einer modifizierten, dadurch nur noch pseudo-zufälligen Verteilung. Vorteile: siehe oben; außerdem: wird von adaptiven Algorithmen häufig als die Strategie 'Zufall' erkannt; deshalb wird die Verteilung der Einsätze so modifiziert, dass auch gegen die von adaptiven Algorithmen festgelegte Zahl für eine echt zufällige Verteilung zwischen a und b im Durchschnitt gewonnen wird. Nachteile: kann durch die Strategie 'Durchschnitt' überwunden werden.

Zufällig weniger Diese Strategie basiert auf einer anderen Strategie S und setzt zufällig etwas weniger als die durch S berechneten Einsätze. Vorteile: Die Basisstrategie S ist schwerer von adaptiven Algorithmen zu erkennen; andere Strategien müssen angepasst werden und bringen nicht so viel Ertrag. Nachteile: die eigene Strategie bringt vor allem bei großer zufälliger Änderung weniger Ertrag; die Eigenschaften der Basisstrategie werden nicht grundlegend verändert.

Adaptiver Algorithmus

Der eigene Setzalgorithmus kann versuchen, die gegnerische Strategie G zu erkennen und durch eine speziell für G implementierte Gegenstrategie die besten Einsätze zu bestimmen. Vorteil: falls die gegnerische Strategie richtig erkannt wird und dafür eine ausreichende Gegenstrategie implementiert ist, gewinnt dieser Algorithmus zuverlässig. Nachteil: selbst wenn für viele gegnerische Strategien eine Erkennung möglich und eine Gegenstrategie implementiert ist, wird es nie möglich sein, alle gegnerischen Strategien zu erkennen und zu überwinden.

Universeller adaptiver Algorithmus

Um möglichst viele Strategien zu überwinden, braucht man deshalb einen Algorithmus, der unabhängig von der speziellen Strategie des Gegners nur auf Grund dessen Setzverhaltens selbstständig die besten Einsätze bestimmt. Alle oben genannten Strategien verhalten sich entweder unabhängig von anderen Einsätzen oder abhängig von gegnerischen Einsätzen verhalten. Außerdem ist es möglich, Einsätze auch abhängig von vorherigen eigenen Einsätzen zu bestimmen. Bei der Analyse des gegnerischen Setzverhaltens sollte man alle diese „Abhängigkeitsvarianten“ berücksichtigen. Außerdem muss in Betracht gezogen werden, dass sich die gegnerische Strategie ändern kann. Deshalb sollten die letzten Einsätze des Gegners stärker gewichtet werden als die vorherigen.

Man geht also einmal unabhängig, einmal abhängig vom eigenen vorletzten Zug und einmal abhängig vom gegnerischen vorletzten Zug so vor: Für alle Einsätze bis zum maximalen Einsatz, den man setzen möchte, bildet man jeweils die Differenz zwischen der Zahl, die man selber im Vergleich zum letzten gegnerischen Zug für diesen Einsatz bekommen hätte, und der Zahl, die der Gegner für den letzten Zug im Vergleich zu diesem Einsatz bekommen hätte. Dazu addiert man die mit der Gewichtung multiplizierten Differenzen der letzten Runden. Zuletzt ist es nur noch nötig, den höchsten dieser Werte herauszufinden, um den besten Einsatz (bei der jeweiligen Abhängigkeitsvariante) herauszufinden. Idealerweise versucht man dann noch herauszufinden, welche Abhängigkeitsvariante der Gegner nutzt, indem man weitere simulierende Berechnungen anstellt.

Eine andere universell-adaptive Vorgehensweise ist es, die Einsätze des Gegners statistisch zu analysieren. Aus Wahrscheinlichkeiten für gegnerische Einsätze lässt sich berechnen, welcher eigene Einsatz den höchsten Gewinn erwarten lässt. Da ein guter Gegner sich aber auch auf eine derartige Vorgehensweise einstellen kann, sollten idealerweise verschiedene Verfahren zur Abschätzung des nächsten gegnerischen Zuges verwendet werden. Aus deren Resultaten kann dann per Zufall oder nach anderen Kriterien für den nächsten eigenen Zug gewählt werden.

5.2 Umsetzung

Die gewählte Spielstrategie war im vom Turniersystem gegebenen Rahmen umzusetzen. In diesem Rahmen kann auf den letzten Zug des Gegners zurückgegriffen werden. Damit ergibt sich die Möglichkeit, alle eigenen Züge und alle Züge des Gegners zu speichern. Anhand dieser Daten kann man versuchen, die Reaktionen des Gegners auf eigene Züge sowie die Entwicklung der gegnerischen Züge zu analysieren und dessen Strategie zu ermitteln – um selber geeignet agieren zu können. Alle oben genannten Arten von Strategien sind im Rahmen des Turniersystems umsetzbar.

5.3 Beispiele

Auch bei einer Turnieraufgabe ist es durchaus möglich, Beispielabläufe z.B. aus einer Herausforderung zu dokumentieren. Alternativ kann man den für das Turniersystem entwickelten KI-Code mit nicht allzu viel Aufwand so in einen Programmrahmen einbetten, dass man als menschlicher Spieler gegen die eigene KI spielen kann. Auf diese Weise lassen sich beliebige Beispiele generieren. Bei diesem sehr einfachen Spiel besteht ein Spielablauf aber ohnehin nur aus einer Folge von Zahlenpaaren. Da ist es auch akzeptabel, anhand von hypothetischen Spielabläufen das Vorgehen der eigenen KI zu demonstrieren.

5.4 Bewertungskriterien

- Das gewählte Verfahren (also die eigene Strategie) soll nachvollziehbar beschrieben und – insbesondere wenn ihr Verhalten nicht allzu offensichtlich ist – an mindestens einem Ablaufbeispiel demonstriert werden.

- Die Strategie sollte zumindest über eine fixe Setzstrategie hinausgehen. Ohne Zufallskomponente oder Adaptivität lässt sich im Turnier kein Blumentopf oder sonst etwas gewinnen. Nachteile der eigenen Strategie sollten zumindest im Fall fixer Strategien erkannt sein.
- Die Regeln bzw. Eigenschaften des Spiels „Groker“ sollen korrekt verstanden und berücksichtigt sein.
- Die Lösungsidee soll in eine im Turniersystem lauffähige KI umgesetzt sein.
- Für das abschließende Turnier muss die KI angemeldet sein und soll dort nicht schlechter abschneiden als eine sehr einfache Zufallsstrategie. Für einen Platz im letzten Fünftel des Feldes gibt es Punktabzug.

Aus den Einsendungen: Perlen der Informatik

Allgemeines

Wort des Wettbewerbs: Algorhytmus

Die letzten Tage waren sozusagen frei, weil wir uns hier mit wirklich einfachen Lösungen zufrieden geben, obwohl es viel coolere Algorithmen gibt. Aber die heben wir uns für die zweite Runde auf.

Ich hatte Freude an der Lösung der Aufgabe, keine bei der Dokumentation selbiger.

Danke an die Person(en), die sich die Zeit nehmen, dieses eher glanzlose Programm zu begutachten.

Eine Möglichkeit, das Problem zu lösen, ist [...]. Dafür habe ich einen Logarithmus geschrieben.

Da es zu kompliziert ist, das mit Worten zu erklären, ist hier der Code.

Die Applikation wurde in Java [...] implimiert.

Das Programm wird in Delphi geschrieben, der Schulunterricht hat mich bisher keine andere Sprache näher gelehrt, daher: Ich kann nicht anders.

Die Schule hat uns alle benötigten Materialien zur Verfügung gestellt, dementsprechend sind keine Kosten angefallen.

Es kann oftmals zu Schwierigkeiten mit Abhängigkeiten kommen. Hier heißt es: den Mut nicht verlieren und weitermachen. Die Tools sind noch nicht ausgereift, aber daran bin ich nicht schuld ;) !

Die Programme funktionieren leider nur teilweise, und ich fand, dass die Aufgaben ziemlich schwer waren.

Das Programm läuft so gesehen fehlerfrei, arbeitet allerdings nicht richtig.

Gibt es ein besseres Beispiel als jenes, welches man auf dem Aufgabenblatt findet? Wahrscheinlich schon.

... da man sonst eine unendliche Rechnung, eine sogenannte Prozessorheizung, verursacht, ...

Codeschnipsel: `while (gehtsNoch())`

Denn mal ganz unter uns: Mir fehlt es immer an Zeug zum Programmieren; mir fällt nie etwas Kreatives ein. Deswegen bin ich dankbar, dass es den BwInf gibt. :-)

Landnahme

Da das Einlesen und die Aufgabe nebensächliche und triviale Dinge sind, spare ich mir hierbei die Erklärung.

Eine Recherche bei meinem Vater hat mich zu folgender eindeutigen Bedingung geführt (danke Papa): ... *Wie schön: die gute alte Offline-Recherche bei Papipedia bringt also gelegentlich doch noch etwas!*

Kassiopeia

Sollte die Eingabedatei in irgendeiner Weise nicht gültig sein (z.B. nicht rechteckig), wird das Programm abstürzen.

Nun wird ein Platz im Array als Kassiopeias Strandort festgelegt.

Wenn der Kraken, der sich auf Kassiopeias Position befindet, nun unendliche Arme ausstreckt, wird die von Tentakeln bedeckte Fläche äquivalent zur Fläche, die Kassiopeia erreichen kann. *Genau genommen würde ein Arm genügen – Kassiopeia kann ja auch nur einen Weg gehen.*

Wir sprechen von Kassi, um Kassiopeia, die Schildkröte, zu bezeichnen. Kassiopeia ist ein viel zu langer Name.

Das kleine „c“ steht für Cessi.

Kassiopeias Weg

Die Kernidee, die sich in meinem Algorithmus wiederfindet, ist, dass das Zentrum der Aufgabe die Schildkröte ist.

Die Methode heißt „alleImHerbstErreichbar“, weil in der Aufgabenstellung steht, dass in „bestimmten Jahreszeiten“ (wie zum Beispiel im Herbst) die Felder nur einmal betretbar sind.

Codeschnipsel:

```
int border = getBorderNumber()?; // was gibt border number zurück?
if self.map == None: print("well shit")
```

Diverse Ausgaben für den Fall, dass Kassiopeia nicht alle Felder betreten kann:

Kassiopeia wird verhungern!

Es ist kein solcher Weg vorhanden, daher muss Kassiopeia elendig verhungern.

Kassiopeia ist tot. :(

Der Hamilton-Algorithmus ist ein geschlossener Pfad, ...

Das Programm funktioniert gut und verarbeitet fast alle der folgenden Beispiele richtig.

Wir nutzen eine Brute-Force-Attacke, um einen möglichen Weg zu finden.

Ameisenfutter

Phytagoras *Pythagoras, pheromongesteuert?*

Armeisen *sehr kompakt für: Ameisen beim Militär*

Da bei dieser Aufgabe vor allem Implementierung gefordert war, werde ich hier erklären, wie ich die Aufgabenstellung verbessert habe.

Im Prinzip ist es gar nicht schwer, die Ameisen diagonal laufen zu lassen. Dies ist nur meine persönliche künstlerische Entscheidung.

Bei der großen Anzahl der Ameisen sind auch wenige Futterquellen schnell gefunden und werden zügig in Teamarbeit abgebaut. Allerdings ächzt Greenfoot etwas unter der Last der großen Anzahl der Objekte. Dies verlangsamt die Bewegung der Ameisen.

Man kann sehen, dass die Ameisen nun mit der Situation überfordert sind.

Mehr Ameisen erhöhen die Laufzeit.

Eine Lösungsidee ist „irrelevant“, da man das Verhalten der Ameisen exakt so implementieren kann, wie es in der Aufgabenstellung erklärt ist. *Also wenn die Aufgabenstellung exakt war, fressen wir 'nen Besen.*

... , beginnt die eigentliche Simulation, deren Schritte in einer for-Schleife, die scheinbar unendlich ist und nur dadurch beendet wird, dass alle Futtereinheiten im Nest sind, was eine if-Bedingung in der Schleife überprüft, indem sie überprüft, ob die Anzahl der Futtereinheiten im Nest den Futtereinheiten pro Futterquelle multipliziert mit der Anzahl der Futterquellen, da das die Anzahl aller Futtereinheiten ist, entspricht, stehen. *Ein Beitrag zur Weltmeisterschaft im Schachteln setzen, äh, Schachtelsätzen.*

Um eine neue Simulation zu starten, muss der Restknopf genutzt werden.

Die positive x-Achse verläuft nach rechts und die positive y-Achse verläuft nach links.

Damit man die Orte nicht erneut tritt, ...

Wenn die Ameise das Nest erreicht, stirbt sie sofort.

Generell ist zu beobachten, dass sich die Ameisen nur langsam vom Nest entfernen und sich dabei immer weiter ausdehnen.

Die Ameisen wollen die Menschheit vernichten und signalisieren dies, indem sie ein großes rotes Kreuz formen.

Flaschenzug

Ich habe mich für die Aufgabe „Flaschenzug“ entschieden, da diese in meinen Augen am meisten mit der Realität zusammenhängt und am ehesten benötigt wird.

Bereits nach kurzer Zeit konnte eine mathematische Lösung (Formelaufstellung) auf Basis unseres Wissensstands ausgeschlossen werden.

Um diese Optimierung auszureizen, wird der Array von Körben davor per Bubblesort der Größe nach sortiert, ...

Nun merkt sich B_n (der Behälter), dass er eigentlich ein Anrecht auf die Flaschen hat. Meine Kisten sind irgendwie sehr egoistisch.

Nun ist die Kiste sehr enttäuscht und behält einfach eine der Flaschen, um es nochmal zu versuchen.

Diese Beispiele nehmen zu viel Zeit in Anspruch, um sie hier darzustellen. Vielleicht reiche ich sie in zwei oder drei Jahren ein, wenn der Rechner fertig ist. ;-)

Mein Programm stößt an keine Grenzen, solange genügend Zeit vorhanden ist. *Die braucht das Programm auch, bei seiner exponentiellen Laufzeit.*

Anschließend wird die Methode solve() aufgerufen, welche alle Behälter immer mehr befühlt. *Hoffentlich ist sie dabei einigermaßen zärtlich, die Methode.*

Die Anzahl der Möglichkeiten lautet: vier Sextilliarden zweihundertsiebenunddreißig Sextillionen sechshundertachtzehn Quintilliarden dreihundertzweiunddreißig Quintillionen einhundertachtundsechzig Quadrilliarden einhundertdreißig Quadrillionen sechshundertdreiundvierzig Trilliarden siebenhundertvierunddreißig Trillionen dreihundertfünfundneunzig Billiarden dreihundertfünfunddreißig Billionen zweihundertzwanzig Milliarden achthundertdreiundsechzig Millionen vierhundertachttausendsechshundertachtundzwanzig. Ein gewöhnlicher Computer benötigt dafür ungefähr 110 Millisekunden.

Außerdem kann es zu Laufzeitproblemen kommen, deswegen habe ich einen Link zu einem 10h Katzenvideo eingefügt. Das sollte genug Zeit sein, damit das Programm für Sie rechnet, während Sie die niedlichen, kleinen und flauschigen Kätzchen beobachten, wie sie zum Beispiel Laserpointern hinterherjagen oder probieren auf Gegenstände zu springen.

Schlüssellöcher

Für die Generierung beschränke ich mich auf asymmetrische Schlüssel, um den Gästen die USB-Stick-Problematik von „passt nicht“, drehen, „passt nicht“, drehen, „passt“ zu ermöglichen.

Ich verwende die Symmetrie aus Aufgabenteil 1. Das sieht auch viel besser aus, Ästhetik ist in der Hotelbranche durchaus von Bedeutung.

Die Hälfte der Hälfte, die mit 1 beginnt, beginnt mit 1, die andere mit 0. Genauso bei der Hälfte, die mit 0 beginnt. *Wir denken immer noch über diesen Satz nach.*

So schafft es das Programm, viele Schlüssel mit niedriger Laufzeit zu erzeugen, die zu 100% nicht doppelt vorkommen und sich zu 100% alle unterscheiden.

Nach der Erzeugung wird noch einmal geprüft, ob tatsächlich alle Schlüssel verschieden sind (sicher ist sicher).

Dennoch ist er zwar nicht besser als Brute Force, jedoch in den meisten Fällen um Einiges schneller und zuversichtlicher.

$N = 1000000000000$: Das geht dafür nicht mehr. So viele Schlüssel braucht aber auch kein Schwein!

Groker

In Runde zwei soll auf 100 gesetzt werden, um die gegnerische KI zu verwirren.

Die Tatsache, dass einige meiner Ansätze gegen einen Bot mit festem Einsatz verloren, hat mir dann zu denken gegeben.

Schließlich habe ich mir überlegt, wie ich selbst so ein Spiel spielen würde. Aber mein Denken ist zu komplex, um es in ein paar Zeilen Code zu formulieren.