29. Bundeswettbewerb Informatik, 2. Runde

Lösungshinweise und Bewertungskriterien

Allgemeines

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

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

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

Dokumentation Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und die Umsetzung in Software fließen lassen. Doch ohne eine gute Beschreibung der Lösungsideen, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine gute Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (die die verschiedenen bei der Lösung

des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung wenig wert. Bewerterinnen und Bewerter können die Qualität Ihrer Einsendung nur anhand dieser Informationen vernünftig einschätzen. Mängel können nur selten durch gründliches Testen der eingesandten Programme ausgeglichen werden – wenn diese denn überhaupt ausgeführt werden können: Hier gibt es häufig Probleme, die meist vermieden werden könnten, wenn Lösungsprogramme vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner getestet würden. Insgesamt sollte die Erstellung des schriftlichen Materials die Programmierarbeit begleiten oder ihr teilweise sogar vorangehen: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt keine saubere Umsetzung in welche Programmiersprache auch immer hin.

Bewertungsbögen Kein Kreuz in einer Zeile bedeutet, dass die genannte Anforderung den Erwartungen entsprechend erfüllt wurde. Vermerkt wird also in der Regel nur, wenn davon abgewichen wurde – nach oben oder nach unten. Ein Kreuz in der Spalte „+“ bedeutet Zusatzpunkte, ein Kreuz unter „–“ bedeutet Minuspunkte für Fehlendes oder Unzulängliches. Dabei haben die Marken nicht immer den gleichen Einfluss auf die Gesamtbewertung, sondern können je nach Einsendung unterschiedlich gewichtig sein. Die Schattierung eines Feldes bedeutet, dass die entsprechenden Zusatz- bzw. Minuspunkte in der Regel nicht vergeben wurden.

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

Aufgabe 1: Kisten in Kisten in Kisten

1.1 Lösungsidee

Als Bea Y. ihr Problem ihren Informatik-Freunden schildert, wird diesen schnell klar, dass ihr sehr praktisches Problem mehrere schwere theoretische Probleme berührt. Bea verkündet zwar stolz, dass sie das Problem systematisch angegangen ist und immer die kleinste Kiste in die nächstgrößere Kiste gepackt hat, in die sie gepasst hat und diesen Prozess so lange wiederholt hat, bis keine Kiste mehr in eine andere gepasst hat. Doch ihre Freunde überzeugen sie schnell, dass ein Vorgehen dieser Art mitunter sehr schlechte Ergebnisse erzielt. Bei ihren eigenen Versuchen fällt ihnen auf, dass das Durchprobieren vieler Möglichkeiten unerlässlich scheint, um eine optimale Lösung zu erzielen. Und auch da ist Vorsicht gefragt, da eine computergestützte Hilfe zunächst berechnen muss, ob eine oder zwei Kisten in eine andere Kiste passen.

Passt Kiste A in Kiste B?

Eine Kiste: Im einfachsten Fall wollen wir nur eine Kiste in eine andere Kiste packen. Die einzige Schwierigkeit, die sich dabei stellt, ist, dass die leeren Kisten beliebig gedreht werden dürfen. Um diesem Problem auszuweichen, überlegen wir uns, die Kisten alle gleich auszurichten: Die längste Seite ist die Breite, die kleinste die Tiefe und die mittlere die Höhe. Sind die zwei Kisten K_1 und K_2 so ausgerichtet, wird klar, dass K_1 genau dann in K_2 passt, wenn jede der Kantenlängen (also die Breite, Tiefe und Höhe) von K_1 um mindestens einen Zentimeter kleiner ist als die entsprechende Kantenlänge von K_2[1]. Um dies formal auszudrücken, beschreiben wir die Kisten von nun an durch Zahlentripel $K_i = (b_i, h_i, t_i)$ aus Breite, Höhe und Tiefe in Zentimetern mit $b_i \geq h_i \geq t_i$ und erhalten:

$$K_1 \text{ passt in } K_2 \text{ genau dann, wenn } b_1 \leq b_2 - 1, h_1 \leq h_2 - 1 \text{ und } t_1 \leq t_2 - 1.$$

Zwei Kisten: Im zweiten Fall möchten wir zwei Kisten nebeneinander in eine Kiste packen. Dazu betrachten wir die verschiedenen Möglichkeiten, zwei Kisten möglichst platzsparend in eine größere Kiste zu platzieren. Ohne Beschränkung der Allgemeinheit können wir davon ausgehen, dass die Kisten nur an den Ecken der umgebenden Kiste platziert werden, da wir jede mögliche Platzierung, die dies nicht erfüllt, an die Ecken verschieben können.

Wir können nun bestimmen, ob die Kisten K_1 und K_2 gleichzeitig in Kiste K_3 passen, indem wir die verschiedenen Möglichkeiten, K_1 zu platzieren, durchprobieren und überprüfen, ob für eine dieser Möglichkeiten K_2 in den verbleibenden Freiraum passt. Dabei ist es irrelevant, in welche Ecke die Kiste platziert wird; lediglich die Ausrichtung der inneren Kiste kann einen Unterschied machen. Wie man sich in Abbildung 1 überzeugen kann, gibt es dafür maximal sechs Möglichkeiten der Ausrichtung (dies entspricht den 3! vielen Permutationen der Koordinaten). Davon müssen nicht alle tatsächlich funktionieren; obwohl eine Kiste prinzipiell in eine andere passt, muss dies nicht für alle Ausrichtungen gelten.

[1] Die Aufgabenstellung sagt aus, dass die inneren Maße einer Kiste um jeweils einen Zentimeter kleiner sind, was einem Rand mit einer Dicke von einem halben Zentimeter entspricht.

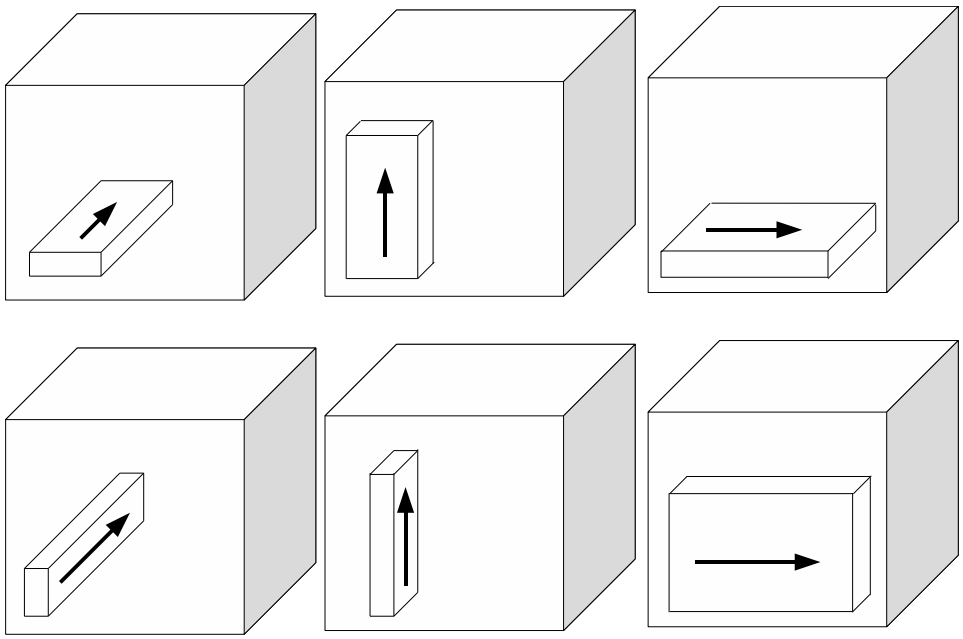


Abbildung 1: Die sechs verschiedenen Möglichkeiten eine Kiste in eine Ecke einer größeren Kiste zu stellen.

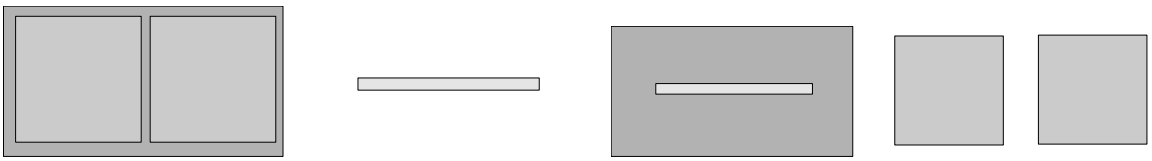
Der Freiraum, der durch das Platzieren der ersten Kiste entsteht, lässt sich als Vereinigung von drei Quadern beschreiben. Konsequenterweise passt eine zweite Kiste nur dann zu der gegebenen Anordnung der ersten Kiste in der größeren Kiste, wenn sie in einen dieser Quader passt. Diese Problem entspricht dann wieder dem obigen ersten Fall: Für alle drei Quader, die den Freiraum überdecken, bringe die Koordinaten in die Standardform (d.h. sortiere sie) und überprüfe die obige Bedingung. Damit ergeben sich für den zweiten Fall maximal 6 Anordnungen $\cdot$ 3 Quader $=$ 18 Tests gemäß dem ersten Fall.

Welche Kiste soll in welche Kiste?

Eine nützliches Hilfsmittel zur Lösung des Problems ist das Volumen unserer Kisten, denn uns fällt auf: Eine Kiste kann nur dann eine andere Kiste enthalten, wenn sie ein größeres Volumen hat. Wenn wir unsere Kisten nach aufsteigendem Volumen sortieren, haben wir also eine Reihenfolge festgelegt, in der eine Kiste nur in Kisten passt, die nach ihr kommen.

Gierige Heuristik: Sortiere die Kisten nach aufsteigendem Volumen. Behandle die Kisten in dieser Reihenfolge und packe eine Kiste in die jeweils nächste größere Kiste, in die sie noch passt. Dabei muss beachtet werden, welche Kisten schon andere Kisten enthalten. Passt sie in keine Kiste, so ist diese Kiste eine äußere Kiste.

Ähnliche Verfahren könnten auch zurückschauen statt voraus: Wir gehen die Kisten wieder nach aufsteigendem Volumen durch und überprüfen für jede Kiste, welche (Paare) von den übrig gebliebenen Kisten mit kleinerem Volumen noch hineinpassen und wählen die Möglichkeit aus, bei der das Volumen der ausgewählten Kiste(n) am größten ist.



(a) Optimale Lösung (b) Greedy-Lösung

Abbildung 2: (Darstellung auf die Ebene projeziert) Die längliche Kiste ist zwar die kleinste, eignet sich aber schlecht als Anfangskiste. Bei der optimalen Lösung wird die längliche Kiste außen stehen gelassen.

Diese Strategien sind zweifelsfrei einfach zu beschreiben und zu implementieren, sind jedoch nicht optimal. Ein Beispiel für einen schwierigen Fall für Greedy-Verfahren findet sich in Abbildung 2. Es fällt auch schwer, abzuschätzen, wie gut die Lösungen sind. Wir suchen vielmehr nach einer genauen Lösung, damit Bea ihren Platz zu Hause auch bestmöglich ausnutzen kann.

Naive Methode: Wir wollen alle validen Packungsanleitungen rekursiv erzeugen. Dazu werden wir immer eine Menge von Kisten um eine große Kiste erweitern und dabei eine valide Packungsanleitung auf alle möglichen Arten erweitern. Wir starten mit dem Basisfall der kleinsten Kiste und sehen, dass diese Kiste alleine eine äußere Kiste sein muss; wir finden nur diese eine valide Packungsanleitung. Haben wir jetzt bereits eine Menge von Kisten $\mathscr{K}$ und eine valide Packungsanleitung gegeben, sowie eine Kiste K_{gross}, die größeres Volumen hat als alle Kisten in $\mathscr{K}$, so erhalten wir valide Packungsanleitungen für $\mathscr{K} \cup \{K_{\text{gross}}\}$ wie folgt:

In $\mathscr{K}$ gibt es keine Kiste, in die K_{gross} hineinpassen könnte. Stattdessen können maximal zwei *äußere* Kisten von $\mathscr{K}$ in K_{gross} gepackt werden. Also testen wir alle äußeren Kisten und Paare äußerer Kisten der gegebenen Packungsanleitung darauf, ob sie in K_{gross} hineinpassen. Wenn ja, erzeugen wir eine neue valide Packungsanleitung für $\mathscr{K} \cup \{K_{\text{gross}}\}$, indem wir hinzufügen, dass die ausgewählte(n) Kiste(n) in K_{gross} gepackt werden. Weiterhin gibt es für jede valide Packungsanleitung für $\mathscr{K}$ die Möglichkeit, K_{gross} leer zu lassen, woraus wir eine zusätzliche Packungsanleitung erzeugen. Für jede erzeugte Packungsanleitung rekurrieren wir nun, betrachten also die nächstgrößere Kiste.

Mit diesem Verfahren können wir rekursiv alle validen Packungsanleitungen erzeugen. Nun reicht es, für jede dieser Möglichkeiten die Summe der Volumen der äußeren Kisten zu bestimmen. Das Minimum aller dieser Werte ist der Wert der optimalen Lösung.

Da wir in jedem Schritt zwei Kisten aussuchen müssen, die in die neue Kiste gepackt werden, betrachten wir (grob abgeschätzt) $O((n!)^2)$ Packungsanleitungen, erhalten also eine mindestens so große Laufzeit. Mit einem ähnlichen Vorgehen kann man sich auch auf $O(n!)$ viele Packungsanleitungen beschränken, das hier beschriebene Verfahren lässt sich jedoch einfacher verbessern, was wir im nächsten Abschnitt tun werden.

Betrachtung aller möglichen Konfigurationen: Bei der naiven Methode werden viele Möglichkeiten unnötigerweise mehrfach betrachtet. Beispielsweise ist es bei drei Kisten K_a, K_b und K_c am Ende gleichwertig, ob K_c in K_b in K_a steckt oder K_c und K_b nebeneinander in K_a

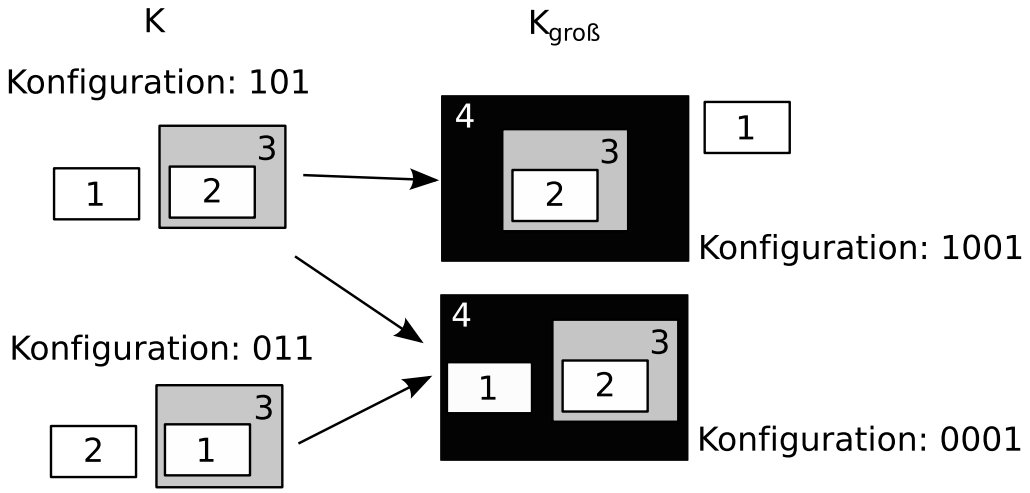


Abbildung 3: Beispiele für Konfigurationen für eine größere Menge an Kisten, die aus möglichen Konfigurationen der kleineren Menge gewonnen werden. Zur einfachen Darstellung wurden die Kisten auf eine Ebene projeziert. Die dargestellten Möglichkeiten sind nicht alle validen Konfigurationen, verdeutlichen aber, dass verschiedene Konfigurationen die gleiche neue Konfiguration erzeugen können.

gepackt sind. Um die mehrfache Betrachtung von Packungsmöglichkeiten zu umgehen, führen wir *Konfigurationen* ein. Eine Konfiguration für m Kisten ist eine Zuordnung von Kisten zu Zuständen z_i, die angeben, ob eine Kiste bereits verpackt ist oder eine äußere Kiste ist. Wir sagen, dass eine Konfiguration valide ist, wenn es eine Möglichkeit gibt, die Kisten so zu packen, dass jede Kiste i mit

- $z_i = 1$ eine äußere Kiste ist,
- $z_i = 0$ eine Kiste ist, die sich in einer anderen Kiste befindet.

Wir können nun obigen naiven Ansatz dahingehend abändern, dass wir statt Packungsanleitungen nur Konfigurationen speichern. Das bedeutet insbesondere, dass wir beim Testen aller Möglichkeiten verschiedene Packungsmöglichkeiten ignorieren, wenn sie dieselben Zustände erzeugen. Zur Verdeutlichung dient Abbildung 3.

Haben wir n Kisten gegeben, reduziert sich die Eingabe der rekursiven Erzeugungsmethode auf einen Bitstring einer Länge m (mit $1 \leq m \leq n$), nämlich einer Konfiguration für die kleinsten m Kisten. Aus diesem Bitstring kann man sofort ablesen, welches die nächstgrößere Kiste ist (nämlich die $m+1$-te), genauso können wir ablesen, welche Kisten äußere Kisten sind. Also reicht solch ein Bitstring aus, um eine Eingabe zu kodieren; wir haben folglich nur $O(2^n)$ verschiedene Eingaben. Wir können nun zu jeder Eingabe das Ergebnis in einer Tabelle abspeichern und auf diese Tabelle zugreifen, sofern wir die gleiche Eingabe ein zweites Mal bekommen (memorization). So reduziert sich die Anzahl an rekursiven Aufrufen von $O((n!)^2)$ auf $O(2^n)$. Für jeden Aufruf testen wir $O(n^2)$ erweiterte Konfigurationen durch Auswahl eines Paares von äußeren Kisten oder einer äußeren Kiste, es ergibt sich also eine Laufzeit von $O(n^2 2^n)$.

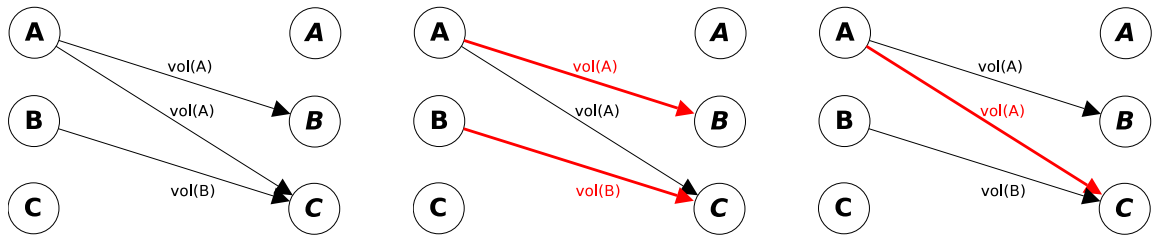


Abbildung 4: Darstellung verschiedener Packungsmöglichkeiten als Weighted Matching in einem bipartiten Graphen

Wie können wir eine Packungsanleitung aus der Konfiguration mit dem besten Außenvolumen rekonstruieren? Dazu kann zu jeder neuen validen Konfiguration beim Aufbau gespeichert werden, wie sie entstanden ist. Speichern wir zum Beispiel den „Vorgänger“ jeder validen Konfiguration und die Packungsanweisung, die zur neuen Konfiguration geführt hat, so können wir rekursiv alle Packungsanweisungen bestimmen, die zu der Konfiguration geführt haben. Gibt es mehrere Vorgänger, so speichern wir uns nur einen davon. Geben wir die Packungsanweisungen dann (umgekehrt) aus, so erhalten wir eine Vorschrift, um die gegebenen Kisten optimal zu packen.

Man kann eine Verbesserung einbauen, die in der Praxis einen großen Zeitgewinn darstellt: das Filtern von *dominierten Lösungen*. Eine valide Konfiguration k dominiert $k' \neq k$, wenn k nur äußere Kisten enthält, die bereits in k' äußere Kisten sind. Anders gesagt enthält k' die äußeren Kisten von k und eine oder mehr zusätzliche äußere Kisten. In Abbildung 3 findet sich ein Beispiel; hier dominiert die Konfiguration 0001 die Konfiguration 1001. Dominierte Konfigurationen können weggelassen werden, da es immer besser ist, nur einen Teil der äußeren Kisten noch übrig zu haben. Baut man dies in den Erstellungsprozess der Konfigurationen ein, so können frühzeitig viele Konfigurationen weggelassen werden, womit die Anzahl betrachteter Konfigurationen erheblich verringert werden kann.

Spezialfall – Unary is . . . efficient: Wir beschränken uns nun darauf, dass wir nicht mehr zwei Kisten nebeneinander in eine größere Kiste packen können, sondern nur noch eine. Dann können wir das Kistenpackproblem als ein klassisches, sehr gut untersuchtes Graphproblem auffassen: das Finden einer größten gewichtete Paarung (maximum weighted matching) in einem bipartiten Graphen. Dazu konstruieren wir einen Graphen G, für den ein Beispiel in Abbildung 4 zu finden ist. Wir führen für jede Kiste $K \in \mathcal{K}$ zwei Knoten in G ein: den einen Knoten fügen wir der Menge $\mathcal{K}_{\text{klein}}$ hinzu, der andere wird Element von $\mathcal{K}_{\text{gross}}$. Nun fügen wir nur Kanten hinzu, die von $\mathcal{K}_{\text{klein}}$ nach $\mathcal{K}_{\text{gross}}$ gehen, womit der entstehende Graph bipartit ist. Dabei gibt es eine Kante zwischen $u \in \mathcal{K}_{\text{klein}}$ und $v \in \mathcal{K}_{\text{gross}}$ genau dann, wenn u in v hinein passt. Zusätzlich bekommt jede Kante ein Gewicht, nämlich das Volumen der beteiligten kleinen Kiste, also $\text{vol}(u)$.

Für diesen Graphen G ist ein größte gewichtete Paarung eine Auswahl an Kanten E, sodass gilt:

1. Jeder Knoten in $\mathcal{K}_{\text{klein}}$ ist Endknoten von maximal einer Kante in E.
2. Jeder Knoten in $\mathcal{K}_{\text{gross}}$ ist Endknoten von maximal einer Kante in E.

3. Unter allen Auswahlen der Kanten, die 1. und 2. erfüllen, ist die Summe der Gewichte aller Kanten in E maximal.

Inwiefern spiegelt diese Formulierung unser Problem wieder? Jede Kante $e \in E$ betrachten wir als Packungsanweisung, das heißt, wenn wir diese Kante, die von einem Knoten $u \in \mathscr{K}_{\text{klein}}$ zu einem Knoten in $v \in \mathscr{K}_{\text{gross}}$ führt, auswählen, so sagt uns dies, dass wir u in v packen sollen. Dann besagt die Bedingung 1 lediglich, dass keine Kiste gleichzeitig in zwei Kisten gepackt sein darf. Das ist so zu verstehen, dass wir nur „direktes Packen“ in zwei Kisten gleichzeitig verbieten; eine Kiste a darf in eine andere Kiste b gepackt werden, die wiederum in einer Kiste c enthalten ist. Es geht jedoch nicht, dass a in b und c enthalten ist, ohne dass b in c gepackt wurde.

Bedingung 2 spiegelt wider, dass keine Kiste zwei oder mehr Kisten nebeneinander enthalten darf. Sind die ersten beiden Bedingungen erfüllt, so haben wir eine Packungsmöglichkeit gefunden. Wie gut ist diese? Für jede Kante, die wir ausgewählt haben, ist die zugehörige kleine Kiste verpackt, kann also keine äußere Kiste mehr sein. Weiterhin ist jede kleine Kiste, die an keiner ausgewählten Kante anliegt, eine äußere Kiste. Summieren wir also über die Gewichte der ausgewählten Kanten, bekommen wir das Gesamtvolumen der *inneren* Kisten. Da wir das Volumen der äußeren Kisten minimieren wollen, können wir genauso gut das Volumen der inneren Kisten maximieren. Damit entspricht Bedingung 3 der Anforderung, eine optimale Packungsmöglichkeit zu finden.

Damit reduziert sich die Lösung von Beas (vereinfachtem) Problem auf das Finden eines größten gewichteten Paares. Dafür ist ein schnelles Lösungsverfahren bekannt, eine Anpassung von Maximalen-Fluss-Algorithmen auf das Paarungsproblem.

1.2 Bewertungskriterien

- Der Test, welche Kisten in andere Kisten passen, muss korrekt sein und nachvollziehbar beschrieben werden. Dabei ist die Effizienz der Lösung weniger wichtig, solange nicht besonders umständlich vorgegangen wird.
- Eine naive Lösung ist nicht ausreichend: es muss eine bessere Lösung als das Durchprobieren von $\geq n!$ vielen Möglichkeiten gefunden werden. Ein Greedy-Verfahren oder andere Heuristiken wurden vielfach verwendet, lösen aber ohne schlüssige Abschätzung der Lösungsgüte oder zumindest Ergebnisvergleiche die Aufgabenstellung nicht zufriedenstellend. Bei einem Greedy-Verfahren wird die Qualität entscheidend von der Sortierung der Kisten beeinflusst, das sollte zumindest ansatzweise betrachtet werden. Sehr gut ist eine garantierte Laufzeitverbesserung wie etwa bei der Nutzung des Konfigurationenansatzes, unter Umständen reicht auch die Unterstützung der vollständigen Suche durch geeignete Heuristiken oder Pruning-Techniken.
- Die Lösung sollte auch nicht anderweitig ineffizient sein (etwa durch einen besonders umständlichen Einpassungstest).
- Auch Fehler bei der Umsetzung können zu Abzügen führen, insbesondere wenn dadurch die Ergebnisse beeinflusst werden.

- Es muss ausgegeben werden, wie die Kisten ineinander zu packen sind. Eine Auflistung von Packungsanweisungen reicht aus, wenn die Anweisungen in der richtigen Reihenfolge stehen. Nicht ausreichend sind unübersichtliche lineare Notationen wie verschachtelte Listen, während eine hierarchisch organisierte Ausgabe des „Verpackungsbaums“ besonders übersichtlich ist.
- Dieses Problem ist inhärent schwierig, so dass Überlegungen zum Laufzeitverhalten der Lösung erwartet werden. Pluspunkte kann es für gute formale Charakterisierungen des Problems geben. Optimalitätsbehauptungen müssen (korrekt) begründet sein.

1.3 Mögliche Erweiterungen

- Die Einschränkung des „binary is beautiful“-Paradigmas vereinfacht die Aufgabenstellung erheblich. Lässt man auch drei und mehr Kisten zu, die nebeneinander in der selben Kiste Platz finden dürfen, so wird der Test, welche Kisten worein passen, viel schwieriger und bedarf neuer algorithmischer Ideen. Aber auch hier gilt: einfach nur alle vorstellbaren Möglichkeiten auszuprobieren oder eine nicht optimale Greedy-Strategie anzuwenden ist eher nicht zufriedenstellend.
- Eine weitere Einschränkung kann aufgehoben werden: Schrägpackungen könnten zugelassen werden, womit mehr Freiheiten der Kistenpackung bestehen. Dies zu modellieren ist einiges aufwändiger, selbst ein langsamer, aber korrekter Test dieses Falls stellt eine gute Erweiterung da.

Aufgabe 2: Containerklamüsel

2.1 Einführendes Beispiel

Betrachten wir uns einen einfachen Fall als Einstieg: Die Container liegen in der in Abbildung 5 dargestellten Reihenfolge. Hier ist der kürzeste Weg offensichtlich, da der Kran den ersten Container (Nr. 3) aufnehmen, an die richtige Stelle bringen und auf den Zug legen kann. Dort findet er sofort den dritten Container (Nr. 2) und bringt ihn an die richtige Stelle, usw, bis er Container Nr. 1 letztendlich nach vorne bringt und so zwangsläufig wieder an der Startposition endet.

Minimal ist dieser Weg, da niemals eine Strecke gefahren wird, ohne dabei einen Container zu transportieren und jeder Container für sich genommen auf direktem Weg zum Ziel gelangt.

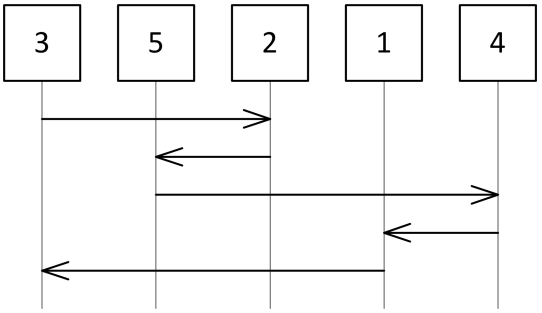


Abbildung 5: Kürzester Weg durch Abarbeitung eines einfachen Zyklus

Wir werden sehen, dass nicht jede zufällig gewählte Permutation der Container einen zusammenhängenden Zyklus wie in Abbildung 5 bildet, wollen zur Herleitung aber zunächst den Algorithmus für diesen Fall formalisieren (*Stellplatz* und *Waggon* beziehen sich hier immer nur auf die aktuelle Position des Krans, also das, was er gerade anfassen kann):

Algorithmus 1 Container-Algorithmus für zusammenhängenden Zyklus

```
repeat
    Nehme Container von Stellplatz auf
    Fahre zur Zielposition des Containers
    Stelle Ladung auf Waggon
until Ausgangsposition wieder erreicht
```

2.2 Permutationen und Zyklen

Permutationen lassen sich in sogenannte *Zyklen* faktorisieren. Dabei handelt es sich um voneinander unabhängige Folgen von Vertauschungen. Man betrachte ergänzend zu Abbildung 5

die Permutation $P = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 5 & 2 & 1 & 4 & 7 & 6 \end{pmatrix}$. Algorithmus 1 folgend arbeiten wir die Positionen in der Reihenfolge $(1\,3\,2\,5\,4)$ ab, was einen Zyklus in P darstellt und kehren wieder zu 1 zurück. Dabei übersehen wir den zweiten Zyklus $(6\,7)$.

Es scheint also sinnvoll, die Container-Permutation P zunächst zu faktorisieren. Dazu beginnen wir bei Position $i := 1$ und ersetzen fortlaufend $i := P[i]$, bis wir auf eine bereits durchlaufene Position stoßen. Die so durchlaufenen Werte von i bilden den ersten Zyklus. Dann fahren wir mit der kleinsten noch nicht besuchten Position fort (In unserem Beispiel $i := 6$, sodass wir nach Besuchen aller Positionen die oben genannte Zerlegung $P = (1\,3\,2\,5\,4) \circ (6\,7)$ speichern.)

2.3 Verbinden angrenzender Zyklen

Es sieht nun so aus, als müsste man für zwei nebeneinander existierende Zyklen eine Art Brücke einbauen. Realisierbar ist dies, indem der Algorithmus bei Besuch einer Position i prüft, ob ein Zyklus beginnend mit $i+1$ angrenzt. In diesem Fall fährt er ohne Ladung dorthin, arbeitet diesen (rekursiv) ab und kehrt bei Erreichen von $i+1$ wieder nach i zurück, um hier fortzufahren. Dieses Verhalten ist in Abbildung 6 illustriert.

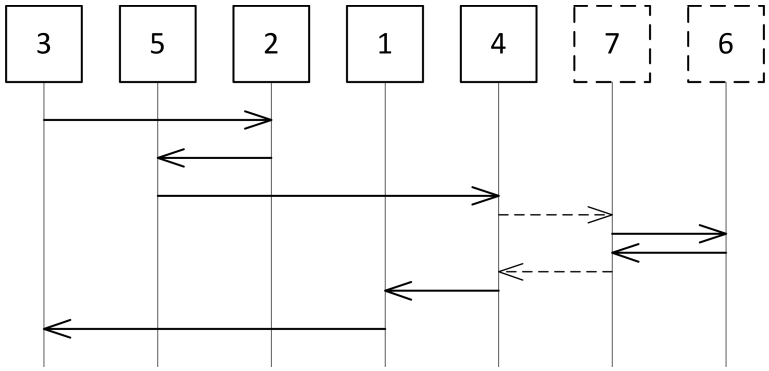


Abbildung 6: Kürzester Brückenschlag zwischen angrenzenden Zyklen.

2.4 Verbinden verschachtelter Zyklen

Nicht jede Faktorisierung liefert angrenzende Zyklen. Für die in Abbildung 7 vorgestellte Permutation P der Container erhalten wir die Zerlegung $P = (1\,3\,2\,7\,6) \circ (4\,5)$. Da wir Container zwischen Waggons und Stellplatz vertauschen können, ist hier der Brückenschlag von i auf $i+1$ weniger ratsam, da Wegstrecke verschwendet würde, ohne etwas zu transportieren.

Stattdessen sollte man den Container einfach auf dem ersten Waggon des umschlossenen Zyklus ablegen – da er ohnehin in die Richtung transportiert werden muss – und dann den inneren Zyklus abarbeiten. Dabei wird dem Kran beim Wiedererreichen der Startposition der zwischengelagerte Container im Weg sein, was wir jedoch durch Vertauschen der Container auf Stellplatz und Waggon lösen können.

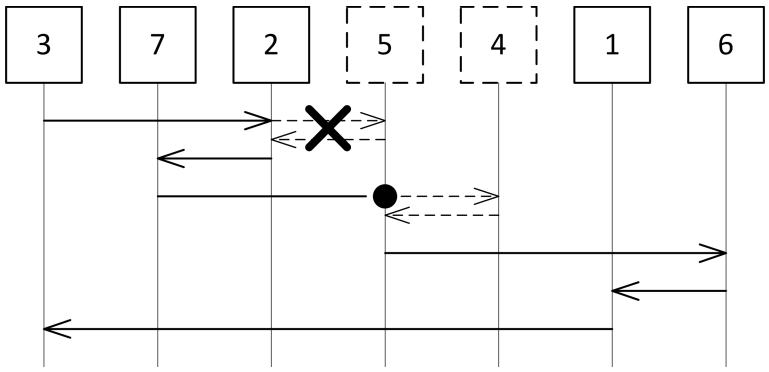


Abbildung 7: Kürzester Brückenschlag zwischen verschachtelten Zyklen.

2.5 Ein vollständiger Algorithmus

Aus diesen beiden Fällen lässt sich Algorithmus 1 rekursiv zu Algorithmus 2 erweitern. Man sieht, dass richtig stehende Container einen „1-er Zyklus“ bilden und damit von den oben erläuterten Verfahren ebenfalls abgedeckt werden. Die einzelnen Befehle sind hier als Anweisungen an den Kranführer zu verstehen, die der Algorithmus ausgeben müsste.

Algorithmus 2 Allgemeiner Container-Algorithmus

```
repeat
    Nehme Container von Stellplatz auf
    while Zielposition nicht erreicht do
        Bewege Kran einen Schritt in Richtung Container-Zielposition
        if neuer Zyklus beginnt an Kranposition then
            Stelle Ladung auf Waggon
            Vertausche
            Arbeite Zyklus rekursiv ab
            Nehme Container von Stellplatz auf
        end if
    end while
    Stelle Container auf Waggon
    if Kranposition ist am rechten Ende des aktuellen Zyklus
    and neuer Zyklus beginnt rechts neben Kranposition then
        Bewege Kran nach rechts
        Arbeite Zyklus rekursiv ab
        Bewege Kran nach links
    end if
until Ausgangsposition wieder erreicht
```

Die Minimalität der Fahrstrecke begründet sich aus der Tatsache, dass für jeden Zyklus der kürzeste Weg errechnet wird (s. einführendes Beispiel) und die Übergänge zwischen den Zyklen kürzest möglich gestaltet werden. Zwischen verschachtelten Zyklen wird stets ein Container transportiert und zwischen angrenzenden Zyklen werden genau zwei Wegabschnitte ohne

Ladung befahren, einer zum Erreichen und einer zur Rückkehr. Man sieht, dass sich dies nicht vermeiden lässt und ein Containertransport über diese Strecke keinen Container seinem Ziel näher bringt.

2.6 Komplexität

Algorithmus 2 ist noch nicht optimal. Es kann u.a. hinsichtlich der Tatsache optimiert werden, dass Rückwärts-Bewegungen keine unbekannten Zyklen überlaufen und so direkt dorthin gesprungen werden kann (Es ergeht an den Kran eine Anweisung, dass er sich zu einer konkreten Position bewegen soll). Ergänzt wird der Algorithmus um eine Lookup-Datenstruktur in Form einer Liste, die zuvor berechnete Zyklen aufsteigend nach ihrem kleinsten Element sortiert aufbewahrt. Abgearbeitete Zyklen werden in konstanter Zeit entfernt und die Startposition des nächsten Zyklus ist dem Algorithmus sofort bekannt, sodass er auch Vorwärts-Sprünge bis zum Erreichen eines Zyklus oder einer Container-Zielposition vornehmen kann. Eine optimale Implementierung erreicht hier selbst im Worst-Case Linearzeit $O(n)$, da der Algorithmus jeden Container maximal zweimal betrachtet.

Das Faktorisieren der Eingangspermutation nach oben vorgestelltem Ansatz geht ebenfalls in $O(n)$ vonstatten.

Die Speicherkomplexität ist durch den rekursiven Ansatz nicht zu unterschätzen. Da im Worst-Case der umgekehrten Container-Reihenfolge bis zu $n/2$ Zyklen verschachtelt sind, ist die Stack-Tiefe hier auch nur linear nach oben beschränkt.

Nicht damit zu verwechseln ist die Wegstrecke, die der Kran zurücklegen muss. Diese kann deutlich länger sein als die Anzahl der Bewegungs-Anweisungen, die der Algorithmus erzeugt. Im Worst-Case lässt sich mittels Reihensumme eine Wegstrecke von $\lfloor n^2/2 \rfloor$ errechnen und im Best-Case muss lediglich verladen und zurückgekehrt werden, was sich zu $2(n-1)$ Abschnitten aufaddiert. Für den unoptimierten, schrittweisen Algorithmus 2 bedeutet dies, dass er ebenfalls eine obere Laufzeitschranke in $O(n^2)$ aufweist.

2.7 Verwandte Lösungsideen

Das Problem lässt sich beispielsweise auch als spezielles **Eulerkreisproblem** auf einem Graphen betrachten, dessen Knoten die Positionen und dessen Kanten die nötigen Transportwege darstellen. Effiziente Algorithmen für diese Problemklasse ähneln dem hier vorgestellten in der Hinsicht, dass sie schrittweise Teil-Zyklen eliminieren.

Durch Backtracking lässt sich zwar auch die kürzeste Wegstrecke finden, für mehr als 6 Container erweist diese sich jedoch als unbrauchbar, da selbst optimierte Algorithmen hierfür einige Stunden benötigen können.

2.8 Bewertungskriterien

- Die charakteristischen Eigenschaften des Problems (Permutationszyklen) sollte in ihrem Wesen erkannt worden sein; eine informelle Beschreibung, auch oder gerade mit selbst geschaffenen Begriffen, ist in Ordnung. Die Lösung sollte auf dieser Grundlage mindestens die Qualität von Algorithmus 2 haben, also angrenzende und verschachtelte Zyklen korrekt und optimal behandeln.
- Positiv ist, wenn weitere Optimierungen vorliegen, z.B. bzgl. der Verwaltung von Zyklen.
- Das gewählte Verfahren sollte nicht unnötig ineffizient sein, egal in welcher Hinsicht.
- Dieses Problem lässt sich gut formalisieren (Permutationszyklen, Eulertouren in Graphkomponenten etc.). Die Qualität der vorgestellten Lösung sollte deshalb klar und nicht nur mit Intuition begründet sein.
- Da die meisten Lösungsverfahren auch für größere Beispiele nicht zu Laufzeitproblemen führen, wird positiv bewertet, wenn doch eine vernünftige Komplexitätsabschätzung unternommen wurde.
- Die Beispiele sollen die verschiedenen Möglichkeiten des Auftretens von Zyklen (angrenzend, verschachtelt) abdecken. Auch größere Beispiele (mindestens 20 Container) sollten behandelt worden sein.

Aufgabe 3: Traumdreiecke

In dieser Aufgabe versuchen wir, eine valide Färbung von Kugeln in einem n-Traumdreieck zu finden. In der ersten Ebene sind $n-0$ Kugeln, in der zweiten sind $n-1$ Kugeln, ... in der i-ten Ebene sind $n-(i-1)$ Kugeln. Die Färbung muss die Bedingung erfüllen, dass die Ecken keines gleichseitigen Dreiecks aus zwei Kugeln auf einer Ebene und einer Kugeln auf einer höheren Ebene gleichfarbig sind. Es ist jedoch in Ordnung, wenn zwei der drei Eckkugeln die gleiche Farbe haben. Die Eckkugeln eines Dreiecks mit zwei Kugeln auf einer Ebene und einer Kugel auf einer tieferen Ebene dürfen gleichfarbig sein. Es sollten möglichst wenig Farben für das n-Traumdreieck verwendet werden.

Das Programm muss für Traumdreiecke der Größe $n \in \mathbb{N}; 1 < n \leq 1000$, gültige Lösungen finden. Dabei ist n die Anzahl der Kugeln in der untersten Ebene und zugleich die Anzahl der Ebenen.

3.1 Verifizierung einer Färbung

Die Verifizierung der Korrektheit einer Färbung kann erfolgen, indem für jedes mögliche Dreieck überprüft wird, ob es einfarbig ist. Dazu kann einfach eine Schleife von oben nach unten über jede Kugel laufen. Die aktuelle Kugel stellt dann für die zu überprüfenden Dreiecke die obere Kugel dar. Somit wird sichergestellt, dass nur $\frac{1}{6} \cdot (n^3 - n)$ Dreiecke in einem n-Traumdreieck überprüft werden. Es kann keinen Verifizierungsalgorithmus geben, der besser als $O(n^3)$ ist, da zur Verifizierung jede der $\frac{1}{6} \cdot (n^3 - n)$ Bedingungen überprüft werden muss.

3.2 Lösungsideen

Triviallösungen

Die einfachste oberste Schranke für die Anzahl der Farben ist die Anzahl der Kugeln: $\frac{n^2+n}{2}$. Eine deutlich kleinere obere Schranke ist n, da jede Ebene mit genau einer Farbe gefärbt werden kann. Doch selbst diese Schranke kann mit $\lceil \sqrt{n} \rceil$-Farben unterboten werden (also 32 Farben für $n = 1000$). Dazu sieht man sich am besten ein n-Traumdreieck an, bei dem n eine Quadratzahl ist. Da einfach Zeilen weggelassen werden können um ein kleineres Traumdreieck zu erzeugen, stellt dies kein Problem dar. Für nicht quadratische n wird die nächstgrößere Quadratzahl gewählt.

Die Färbung funktioniert folgendermaßen:

Eine Liste von $\sqrt{n}$ Farben wird erstellt. i läuft nun von 1 bis n.

1. Die Zeile i wird in Abschnitte der Länge $\sqrt{n}$ unterteilt. Beim ersten Durchlauf gibt es $\sqrt{n}$ Abschnitte. Später ist der letzte Abschnitt kürzer.
2. Der erste Abschnitt wird mit der ersten Farbe in der Farbliste gefärbt, der zweite Abschnitt mit der zweiten Farbe, usw.
3. Die letzte Farbe in der Farbliste wird an die erste Stelle verschoben.

Dieser Ansatz wird als trivial bezeichnet, da die Färbung einer Kugel durch eine einfache Vorschrift angegeben werden kann, ohne auf die Färbung anderer Kugeln Bezug zu nehmen.

Bruteforce, also das Durchgehen aller möglichen Färbungen, führt theoretisch auch irgendwann zu optimalen Lösungen. Allerdings gibt es für c Farben $c^{\frac{n^2+n}{2}}$ Färbungen. Dieser Lösungsweg ist als für größere n nicht möglich.

Backtracking

Backtracking ist eine Möglichkeit, eine optimale Lösung zu finden. Es wird versucht, das Traumdreieck so weit wie möglich zu färben. Sobald es mehrere Färbungsmöglichkeiten gibt, muss man sich für eine entscheiden und weitermachen. Falls das die falsche Möglichkeit war, geht man zurück und nimmt die nächste mögliche Färbung.

Beim Backtracking muss zuvor entschieden werden, wie viele Farben die Lösung haben soll. Falls zu wenige gewählt werden, wird keine Lösung gefunden, aber jede Möglichkeit ausprobiert. Falls eine Lösung gefunden wird, kann man es mit weniger Farben versuchen.

Sobald mit c Farben keine Lösung errechnet wird, aber mit $c+1$ Farben, weiß man, dass $c+1$ Farben optimal für das gegebene n-Traumdreieck sind. Diese Lösungsidee führt zwar auf jeden Fall zu einer optimalen Lösung, wird der Aufgabenstellung jedoch nicht gerecht, da die Ausführungszeit bereits für $n = 17$ sehr hoch ist. Tabelle 1 zeigt die Anzahl der benötigten Farben für einige kleinere n.

n	2 - 4	5 - 17
Farben	2	3

Tabelle 1: Ergebnisse mit dem Backtracking-Algorithmus

Die Anzahl der optimalen Lösungen nimmt übrigens stark zu. Macht man nur die Einschränkung, dass die Farbe 1 an der Spitze sein soll, gilt Tabelle 2. Es sind natürlich einige symmetrische Lösungen dabei.

Anzumerken ist beim Backtracking noch, dass nichtdeterministische Algorithmen häufig schneller sind. Wenn man aus den möglichen Farben eine zufällig auswählt, kommt man um ein vielfaches schneller ans Ziel.

n	Farben	Färbungen
2	2	3
3	2	8
4	2	13
5	3	375.695
6	3	34.364.322

Tabelle 2: Anzahl der optimalen Färbungen mit der Farbe 1 an der Spitze

Iterativer Algorithmus

Einen Kompromiss zwischen einer optimalen Lösung und einer vertretbaren Ausführungszeit bietet folgender Algorithmus:

Eine Liste beinhaltet alle bisher verwendeten Farben.
Eine Schleife läuft über jede Kugel.
Für die aktuelle Kugel wird bestimmt, welche Farben nicht verwendet werden dürfen, weil sie zu einem einfarbigem Dreieck führen würden.
Dann wird die Differenzmenge A aus der Liste der bisher verwendeten Farben und der nicht zu verwendenden Farben gebildet.
Ist nur eine Farbe in A, wird die aktuelle Kugel mit dieser gefärbt.
Ist keine Farbe in A, muss eine weitere Farbe verwendet werden.
Sind zwei oder mehr Farben in A, kann durch ein Wahlverfahren eine Farbe bestimmt werden. Dieses Wahlverfahren kann eine zufällige Farbe aus A auswählen, die Farbe wählen, die bisher am seltensten im Traumdreieck vorkommt, oder eine beliebige andere Heuristik verwenden.

Diese Methode benötigt mindestens $\frac{1}{12} \cdot (2n^3 + 8n^2 + 5n)$ und maximal $\frac{1}{6} \cdot (2n^3 + 5n^2 + 2n)$ Lese-Operationen. Zusätzlich fallen noch zwischen x und $2x$ Vergleichsoperationen an, wobei x die Anzahl der Dreiecke ist. Die Zeitkomplexität des Algorithmus liegt also in $O(n^3)$.

n	2	3	4	9	300	400	500	600	700	800	900	1000
Farben	2	2	3	4	16	18	20	22	23	25	27	27

Tabelle 3: Ergebnisse mit dem iterativen Algorithmus

Andere Zufallsverfahren

Bei Optimierungsproblemen wie diesem werden immer häufiger und vielfach sehr erfolgreich Verfahren eingesetzt, die im Wesentlichen auf dem Einsatz des Zufallsprinzips beruhen. Eine gute Idee ist es, die Kugeln in zufälliger Folge abzuarbeiten und bei jeder Kugel die Farben solange zu probieren, bis eine gültige Färbung gefunden ist. Mit Zufallsverfahren können für $n = 1000$ Färbungen mit weniger als 20 Farben gefunden werden. Diese Ergebnisse lassen sich noch verbessern, wenn anschließend versucht wird, Farben zu eliminieren. Hierfür sind Suchverfahren nötig, die in Suchtiefe oder Laufzeit begrenzt werden müssen. Mit sehr viel

Aufwand konnte eine Lösung für $n = 1000$ eine Färbung mit 15 Farben bestimmen, andere kamen mit 16 Farben aus.

Weitere Methoden

Die hier vorgestellten Lösungsmöglichkeiten erheben keinesfalls Anspruch auf Vollständigkeit. Weitere Heuristiken sind vorstellbar, auch grundsätzlich andere Vorgehensweisen:

- Es könnte ein Integer Program Solver oder Constraint Satisfaction Problem Solver zur Lösungsfindung benutzt werden. Dann müsste „nur“ die Modellierung vorgenommen werden.
- Das n-Traumdreick kann als gerichteter Graph aufgefasst werden. Dann könnte man versuchen, die Knoten in möglichst wenige Teilmengen aufzuteilen, sodass in keiner Teilmenge ein gerichtetes Dreieck gebildet werden kann. Siehe Abbildung 8.

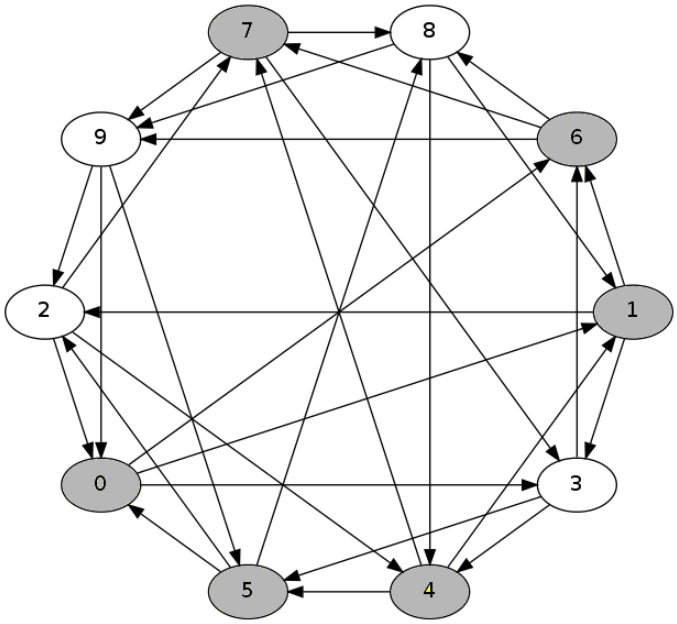


Abbildung 8: Gefärbter Graph eines 4-Traumdreiecks. Ein ungerichteter Graph hätte für (0, 1, 6) ein Dreieck.

Allerdings müssten auch diese Lösungsmöglichkeiten in akzeptabler Zeit ablaufen.

Auch Hybridmethoden, die mehrere Lösungsansätze kombinieren, sind sehr gut vorstellbar. Dabei tritt besonders die Notwendigkeit zu Tage, mit verschiedenen Ansätzen zu experimentieren, da der Effekt und die Qualität der Heuristiken sehr schwer zu beurteilen sind.

3.3 Bewertungskriterien

- Es muss eine gültige Färbung erzeugt werden. Dazu ist entweder ein korrektes Prüfverfahren nötig, oder es muss begründet werden, warum das realisierte Verfahren nur gültige Färbungen erzeugt.
- Das Erzeugen der gültigen Färbung darf nicht zu langsam sein. Die Färbung für $n = 1000$ muss in akzeptabler Zeit möglich sein.
- Eine Lösung für den maximalen Fall darf auch nicht anderweitig, etwa durch übermäßige Speichernutzung bei rekursiven Verfahren, verhindert werden.
- Es gibt Pluspunkte, wenn die Lösung besonders gute Ergebnisse liefert. Es gibt Minuspunkte, wenn die Lösung (etwa eine der Triviallösungen) schlechte Ergebnisse liefert. Maximale Pluspunkte gibt es, wenn die Lösung für Dreiecke mit 1000 Ebenen weniger als 20 Farben benötigt.
- Weniger gute Ergebnisse oder andere Probleme können natürlich auch auf Fehlern in der Umsetzung bzw. Implementierung beruhen.
- Ein allzu einfaches Lösungsverfahren führt zu starker Abwertung.
- Der Trade-Off zwischen Qualität/Optimalität und Zeitbedarf sollte angesprochen werden; Komplexitätsüberlegungen sind also Pflicht. Die Qualität heuristischer Lösungen (inkl. der randomisierten Verfahren) sollte begründet bzw. abgeschätzt sein, z.B. durch Vergleiche.
- Zu einer repräsentativen Auswahl von Dreieckshöhen soll die Anzahl der verwendeten Farben übersichtlich dargestellt sein, typischerweise tabellarisch. Mindestens drei Färbungsbeispiele müssen überschaubar angegeben sein. Besonders gut nachvollziehbare Darstellungen werden positiv bewertet.

Perlen der Informatik – aus den Einsendungen

Allgemeines

Eine Überprüfung, ob es auch stimmt, ist überflüssig.

Wenn man in der Datei `main.java` die Konstante `BLABLA` auf `true` ändert, erhält man mehr Programmausgaben.

Worte des Wettbewerbs: expotenzionell, Murmelnator, Kistenator

Aufgabe 1: Kisten in Kisten in Kisten

Funktion: `findPlace4NewKiste`

Die Klasse „World“ erbt von „Box“. *Die Welt ist eine Kiste?*

Als allererstes kommt einem natürlich wie bei fast jedem Problem ein Brute-Force Algorithmus in den Sinn.

... Problem der optimalen Verschachtelung oder Verkistelung.

Damit er beim Tauschen nicht ins Unendliche gleitet, ...

Aufgabe 2: Containerklamüsel

Wenn sich das Schiff nicht in 2 Teile teilen lässt ...

Diese müssen aussortiert werden, weil sie nicht erfordern gelöst zu werden und so den Algorithmus durcheinander bringen würden.

Aufgabe 3: Traumdreiecke

Wir möchten eine gute Lösung bekommen, bevor die Erde [in] die Sonne stürzt.

Für $n > 9$ ist ein Beweis durch Probieren zu umfangreich.

Dieses Dreieck *($n = 1000$)* abzudrucken erscheint mir etas Wald-beeintrechtigend. [sic!]

Diese endgültige Niederlage bezüglich des Konzepts der rohen Gewalt brachte mich letztendlich zu meinem letzten Algorithmus.

Die Funktion faerbekugel() besteht im Grunde aus einer fußgesteuerten Zählschleife.

Die Herausforderung liegt klar in der dritten Dimension! *Anmerkung zu einer Erweiterung auf Tetraeder.*

Die Klasse Triangle ist der eigentliche Ursprung von Allem. *Wir hatten ja immer schon vermutet, dass der BwInf auch religiöse Aspekte hat ...*