

Lösungshinweise

Allgemeines

Es ist immer wieder bewundernswert, wie viel an Ideen, Wissen, Fleiß und Durchhaltevermögen in den Einsendungen zur 2. Runde eines Bundeswettbewerbs Informatik stecken. Um aber die Allerbesten für die Endrunde zu bestimmen, müssen wir die Arbeiten kritisch begutachten und hohe Anforderungen stellen. Deswegen sind Punktabzüge die Regel und Bewertungen mit Pluspunkten über die Erwartungen hinaus die Ausnahme.

Spannende bzw. schwierige Erweiterungen der Aufgabenstellung sind Extrapunkte wert, wenn sie auch praktisch realisiert wurden. Weitere Ideen ohne Implementierung und geringe Verbesserungen der bereits implementierten Lösung einer Aufgabe gelten allerdings nicht als geeignete Erweiterungen. Intensive theoretische Überlegungen wie z. B. ein korrekter Beweis zur Komplexität des Problems werden ebenfalls mit zusätzlichen Punkten belohnt.

Falls Ihre Einsendung nicht herausragend bewertet wurde, lassen Sie sich auf keinen Fall entmutigen! Allein durch die Arbeit an den Aufgaben und ihren Lösungen hat jede Teilnehmerin und jeder Teilnehmer viel gelernt; diesen Effekt sollten Sie nicht unterschätzen. Selbst wenn Sie die Lösung zu nur einer Aufgabe einreichen konnten, so kann die Bewertung Ihrer Einsendung bei der Anfertigung künftiger Lösungen hilfreich für Sie sein.

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

Bewertungsbogen

Aus der 1. Runde oder auch aus früheren Wettbewerbsteilnahmen kennen Sie den Bewertungsbogen, der angibt, wie Ihre Einsendung die einzelnen Bewertungskriterien erfüllt hat. Auch in dieser Runde können Sie den Bewertungsbogen im Anmeldesystem AMS einsehen. In der 1. Runde ging die Bewertung noch von 5 Punkten aus, von denen bei Mängeln abgezogen werden konnte. In der 2. Runde geht die Bewertung von zwanzig Punkten aus, bei denen Punkte abgezogen und manchmal auch hinzuaddiert werden konnten. In dieser Runde gibt es auch deutlich mehr Bewertungskriterien als in der 1. Runde.

Terminlage

Für Abiturientinnen und Abiturienten ist der Terminkonflikt zwischen Abiturvorbereitung und 2. Runde sicher nicht ideal. Doch leider bleibt dem Bundeswettbewerb Informatik nur die erste Jahreshälfte für diese Runde: In der zweiten Jahreshälfte läuft nämlich die zweite Runde des

Mathematikwettbewerbs, dem wir keine Konkurrenz machen wollen. Aber: die Bearbeitungszeit für die Runde beträgt etwa vier Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der wichtigen, ausführlichen Dokumentation der Aufgabenlösungen zu vermeiden. Aufgaben der 2. Runde sind oft deutlich schwerer zu lösen, als sie auf den ersten Blick erscheinen. Erst bei der konkreten Umsetzung einer Lösungsidee stößt man manchmal auf Besonderheiten bzw. noch zu lösende Schwierigkeiten, was dann zusätzlicher Zeit bedarf. Daher ist es sinnvoll, die einzureichenden Aufgaben nicht nacheinander, sondern relativ gleichzeitig zu bearbeiten, um nicht vom zeitlichen Aufwand der jeweiligen Aufgabe kurz vor Ablauf der Bearbeitungszeit unangenehm überrascht zu werden.

Dokumentation

Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und deren Umsetzung in Software fließen lassen. Doch ohne eine verständliche Beschreibung der Lösungsideen und ihrer jeweiligen Umsetzung, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine geeignete Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (welche die verschiedenen bei der Lösung des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung nur wenig wert.

Bewerterinnen und Bewerber können die Qualität Ihrer Aufgabenlösungen nur anhand dieser Informationen vernünftig einschätzen. Mängel in der Dokumentation der Einsendung können nur selten durch Ausprobieren und Testen der Programme ausgeglichen werden – wenn die Programme denn überhaupt ausgeführt werden können: Hier gibt es gelegentlich Probleme, die meist vermieden werden könnten, wenn Lösungsprogramme vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner auf Lauffähigkeit getestet würden. Insgesamt sollte die Erstellung der Dokumentation die Programmierarbeit begleiten oder ihr teilweise sogar vorangehen: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt auch keine saubere Umsetzung hin, in welcher Programmiersprache auch immer.

Einige unterhaltsame Formulierungssperlen sind im Anhang wiedergegeben.

Bewertung

Bei den im Folgenden beschriebenen Lösungsideen handelt es sich nicht um perfekte Musterlösungen, sondern um sinnvolle Lösungsvorschläge. Dies sind also nicht die einzigen Lösungswege, die wir gelten ließen. Wir akzeptieren vielmehr in der Regel alle Ansätze, auch ungewöhnliche, kreative Bearbeitungen, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind. Einige der Fußnoten in den folgenden Lösungsvorschlägen verweisen auf weiterführende Fachliteratur für besonders Interessierte; Lektüre und Verständnis solcher Literatur wurden von den Teilnehmenden natürlich nicht erwartet.

Unabhängig vom gewählten Lösungsweg gibt es aber Dinge, die auf jeden Fall von einer guten Lösung erwartet wurden. Zu jeder Aufgabe wird deshalb in einem eigenen Abschnitt, jeweils am Ende des Lösungsvorschlags erläutert, auf welche Kriterien bei der Bewertung dieser Aufgabe besonders geachtet wurde. Dabei können durchaus Anforderungen formuliert werden, die aus der Aufgabenstellung nicht hervorgingen. Letztlich dienen die Bewertungskriterien dazu,

die allerbesten unter den sehr vielen guten Einsendungen herauszufinden. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation (klare Beschreibung der Lösungsidee, genügend aussagekräftige Beispiele und wesentliche Auszüge aus dem Quellcode) einschließlich einer theoretischen Analyse (geeignete Laufzeitüberlegungen bzw. eine Diskussion der Komplexität des Problems) sowie an den Quellcode der implementierten Software (mit übersichtlicher Programmstruktur und verständlicher Kommentierung) und an das lauffähige Programm (ohne Implementierungsfehler). Wünschenswert sind auch Hinweise auf die Grenzen des angewandten Verfahrens sowie sinnvolle Begründungen z. B. für Heuristiken, vorgenommene Vereinfachungen und Näherungen. Geeignete Abbildungen und eigene zusätzliche Eingaben können die Erläuterungen in der Dokumentation gut unterstützen. Die erhaltenen Ergebnisse für die Beispieleingaben (ggf. mit Angaben zur Rechenzeit) sollten leicht nachvollziehbar dargestellt sein, z. B. durch die Ausgabe von Zwischenschritten oder geeignete Visualisierungen. Eine Untersuchung der Skalierbarkeit des eingesetzten Algorithmus hinsichtlich des Umfangs der Eingabedaten ist oft ebenfalls nützlich.

Danksagung

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

An der Erstellung der im Folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Tim Pokart (Aufgabe 1), Hans-Martin Bartram (Aufgabe 2) sowie Pascal Atzler (Aufgabe 3) und Boldizsár Mann in beratender Funktion. Allen Beteiligten sei für ihre Mitarbeit ganz herzlich gedankt.

Aufgabe 1: Schmucknachrichten

In dieser Aufgabe soll für Sybille und Pedram ein maßgeschneiderter präfixfreier Code erstellt werden. Mit diesem soll eine Textnachricht mittels Farbperlen möglichst kurz codiert werden, um schlussendlich eine kompakte Perlenkette zu ergeben. In der Aufgabenstellung wird zwischen Perlensammlungen mit

- a) jeweils gleich großen Perlen und
- b) möglicherweise unterschiedlich großen Perlen

unterschieden. Im Folgenden werden wir grundsätzlich den allgemeinen Fall mit verschiedenen Perlengrößen betrachten, aus dem sich der mit gleich großen als Spezialfall ergibt.

Im Sinne der Aufgabenstellung treffen wir außerdem folgende Annahmen:

- Es ist für uns nicht relevant, wie Sybille und Pedram schlussendlich den Startpunkt der Sequenz auf der Kette markieren. Lediglich die Bedingung, dass sich ab dem Startpunkt ein eindeutig entschlüsselbarer Code ergibt, ist relevant.
- Auf dem Beipackzettel darf jede Perle nur mit einem Unicode Buchstaben assoziiert werden. Buchstabengruppen oder gar Teilsätze dürfen nicht eingeschlossen werden.
- Das Leerzeichen “_” ist ein zu codierendes Symbol.
- Die Güte eines Codes messen wir an der Gesamtlänge der Kette, welche sich als Summe der benötigten Perlendurchmesser ergibt.

1.1 Lösungsidee

Im Folgenden werden drei Lösungsansätze zum Finden von zunächst sehr kurzen und dann optimalen Codierungen vorgestellt. Alle drei Ansätze haben dabei gemeinsam, dass sie das Problem als eine Suche auf Bäumen formulieren. Diese bieten den Vorteil, dass die so entstehenden Codes automatisch präfixfrei sind. Ein Code ist präfixfrei, wenn kein gültiges Codewort der Anfang eines anderen ist; dann kann jede codierte Nachricht eindeutig entschlüsselt werden. In Abbildung 1.1 ist die Übersetzung einer Kodierungstabelle mit den Code in einen solchen Baum skizziert.

Um Verwirrungen vorzubeugen, folgen zunächst einige Definitionen. Von der Aufgabenstellung gegeben ist ein Text mit dem Eingabealphabet $\mathcal{E} = \{e_1, \dots, e_n\}$, das aus den insgesamt n benutzten (Unicode-)Symbolen e_i besteht. Jedes Symbol aus dem Eingabealphabet kommt im Text mit der Häufigkeit $\text{Häufigkeit}(e_i)$ vor. Außerdem werden die p Perlen gegeben, die unser Codierungsalphabet $\mathcal{C} = \{c_1, \dots, c_p\}$ mit den Codierungssymbolen c_i ergeben. Die Perlen haben jeweils den ganzzahligen Durchmesser (c_i) (in Millimeter). Ein Codewort $w = c_{i_1} \cdots c_{i_m}$ ergibt sich dann aus der Aneinanderreihung von Codierungssymbolen, in dem Fall aus der Aneinanderreihung der insgesamt m Codierungssymbole c_{i_1}, \dots, c_{i_m} . Alle Codewörter zusammen ergeben den schlussendlichen Code $W = \{w_1, \dots, w_n\}$, bei dem jedem Symbol e_i aus dem Eingabealphabet E ein Codewort w_i zugeordnet wird. Wir definieren die Länge eines Codeworts $w = c_{i_1} \cdots c_{i_m}$ als Summe der Länge der Codierungssymbole, also

$$\text{Länge}(w) = \text{Durchmesser}(c_{i_1}) + \dots + \text{Durchmesser}(c_{i_m}).$$

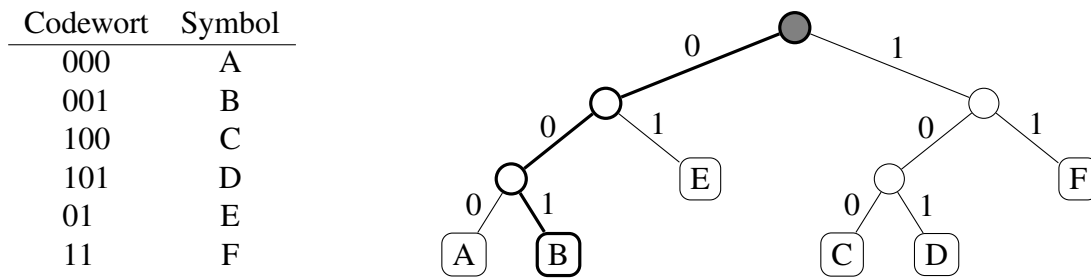


Abbildung 1.1: Übersetzung der links dargestellten Codetabelle in den rechts dargestellten Codierungsbaum. Dabei diktiert das Codewort für jedes Symbol ausgehend vom Wurzelknoten (grau) die Position im Baum. Hat dieses an der aktuellen Stelle eine 0, befindet sich das Symbol im linken Teilbaum des aktuellen Knotens, bei einer 1 im rechten. Geht man so das Codewort von links nach rechts durch, befinden sich die zu codierenden Symbole dann jeweils an den Blättern des Baums. Umgekehrt kann für jedes Symbol das Codewort durch Abgehen der Kanten erhalten. Folgt man dem fett markierten Pfad zum Symbol “B”, muss man beispielsweise die Kanten 001 entlang, was seinem Codewort entspricht.

Für einen Text ergibt sich damit die Gesamtlänge als Summe der jeweiligen Codewortlängen multipliziert mit der Häufigkeit des Eingabesymbols, welches sie codieren, also

$$\text{Länge}(\text{Text}) = \sum_{w_i \in W} \text{Länge}(w_i) \times \text{Häufigkeit}(e_i). \quad (1.1)$$

Damit der Code präfixfrei ist (kein Codewort ist der Anfang eines anderen Codewortes), muss für jedes Paar w_i und w_j aus Codewörter, niemals $w_i = w_j \cdot s$ mit s einer beliebigen Sequenz von Codierungssymbolen gelten.

Beispiel

Im Beispieltext $T = \text{DIE_SONNE_SOLL_DIR_IMMER_SCHEINEN}$ aus der Aufgabenstellung tauchen die $n = 12$ Eingabesymbole

$$E = \{ _, C, D, E, H, I, L, M, N, O, R, S \}$$

auf; wobei beispielsweise $e_2 = C$ und $e_{11} = R$. Durch Zählen im Text ergeben sich die Häufigkeiten, so ist beispielsweise $\text{Häufigkeit}(e_2) = 1$ und $\text{Häufigkeit}(e_{11}) = 2$. Zur Codierung stehen die $p = 2$ Perlen in rot und blau zur Verfügung, also $C = \{r, b\}$. Diese haben jeweils die gleichen Durchmesser, sodass $\text{Durchmesser}(r) = \text{Durchmesser}(b) = 1$. Aus den beiden Codierungssymbolen ergeben sich die Codewörter

$$W = \{ rrb, bbbbb, bbbr, rrr, bbbbr, rbb, bbrb, brrr, rbr, brbb, brbr, brr \},$$

wobei nun beispielsweise e_2 der Code $w_2 = bbbbb$ zugeordnet wurde; seine Länge ist $\text{Länge}(w_2) = 5$. Insgesamt hat der Text eine Länge von 113.

Eine untere Schranke für die Kettenlänge

Wie gut ein verlustfreier Code, also insbesondere auch ein präfixfreier Code, eine Nachricht codieren kann, ist in der Informationstheorie unter dem Shannon Coding Theorem¹ bekannt. Im Folgenden wird dieses für die Perlenketten in der Aufgabenstellung formuliert, es reicht allerdings auch zu wissen, dass sich daraus eine minimale Länge für die Perlenketten ergibt, die diese nicht unterschreiten können.

Konkret lautet es wie folgt: Seien p_i die relativen Häufigkeiten der Eingabesymbole e_i , also

$$p_i = \frac{\text{Häufigkeit}(e_i)}{\text{Gesamtanzahl Eingabesymbole}}.$$

Dann ist durch die Entropie

$$H(\{p_i\}) = -\sum p_i \log_2 p_i$$

und die Kapazität t des benutzten Informationskanals eine obere und untere Schranke der zu erwartenden Nachrichtenlänge L durch

$$\frac{H(\{p_i\})}{-\log_2 t} \leq \frac{L}{\text{Länge}(\text{Text})} \leq \frac{H(\{p_i\})}{-\log_2 t} + 1$$

gegeben. Der Informationskanal ist in unserem Fall durch die Perlen gegeben, wir verwenden also den Bunte-Perlen-auf-Halsketten-kanal. Seine Kapazität t ist abhängig von den Perlenlänge und durch die positive Nullstelle des folgenden Polynoms gegeben:

$$t^{\text{Durchmesser}(c_1)} + \dots + t^{\text{Durchmesser}(c_p)} - 1 = 0.$$

Eine Motivation für die Form dieses Polynoms wird später in Abschnitt 1.5 skizziert. Die untere Schranke $\text{ShannonBound}(\text{Text}) = -H(\{p_i\})/\log_2 t$ bezeichnen wir als *Shannon-Bound*. Dadurch dass die Häufigkeiten der Eingabesymbole direkt aus dem zu codierenden Text stammen, ist die zu erwartende Länge L gleich der optimalen Länge. In diesem Sinne ist es also sinnvoll, die Shannon-Bound als untere Schranke für die Länge des codierten Texts zu verwenden.

Beispiel

Für das oben beschriebene Beispiel aus der Aufgabenstellung ist die Kanalkapazität durch die Gleichung $t^1 + t^1 - 1 = 0$ zu bestimmen, also ist $t = 0.5$. Rechnen wir das mit $-\log_2 t = \log_2 2 = 1$ in Bits um, bedeutet das also, dass der Kanal mit jedem Symbol genau ein Bit transportiert; das ergibt Sinn, da die beiden Perlen durch ihren identischen Durchmesser ebenso gut als 0 und 1 verstanden werden können. Die Entropie ergibt sich aus den Häufigkeiten

$$\begin{array}{llllll} p_{\text{U}} = 5/33, & p_{\text{C}} = 1/33, & p_{\text{D}} = 2/33, & p_{\text{E}} = 5/33, & p_{\text{H}} = 1/33, & p_{\text{I}} = 4/33, \\ p_{\text{L}} = 2/33, & p_{\text{M}} = 2/33, & p_{\text{N}} = 4/33, & p_{\text{O}} = 2/33, & p_{\text{R}} = 2/33, & p_{\text{S}} = 3/33. \end{array}$$

¹https://en.wikipedia.org/wiki/Shannon%27s_source_coding_theorem

Damit ergibt sich die Gesamtentropie als $H(\{p_i\}) \approx 3,409$ Bits. Der Text ist insgesamt 33 Zeichen lang, die Gesamtlänge der codierten Nachricht L liegt im Erwartungswert also zwischen

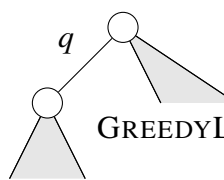
$$112 \leq L \leq 113.$$

1.2 Greedy-Ansatz durch n -ary Huffman Codierung

Für binäre Codierungsalphabete mit gleichen Längen kann mit Hilfe der Huffman Codierung ein optimaler präfixfreier Code gefunden werden. Für allgemeinere Codierungsalphabete mit mehr als zwei Symbolen ist der sich mit einer Verallgemeinerung des Verfahrens ergebende Code allerdings nicht optimal, dient jedoch als Startpunkt für den quasi-polynomiellen Ansatz in Abschnitt 1.4.

Der Grundgedanke des Greedy Verfahrens ist folgender: Sortiert man die Eingabesymbole nach ihrer Häufigkeit, so ist es voraussichtlich von Vorteil, Symbolen mit einer höheren Häufigkeit Perlen mit einer kleineren Länge zuzuordnen. Um eine möglichst optimale Zuordnung zu finden, definieren wir die Tabelle GREEDYLÄNGE. Ihre Einträge ergeben sich wie folgt: Für die Eingabesymbole mit Indizes zwischen i_{start} und i_{end} wird in $\text{GREEDYLÄNGE}(i_{\text{start}}, i_{\text{end}}, q)$ die vom Greedy Verfahren als optimal gefundene Codewortlänge eingetragen, bei der alle Codewörter nur mit den Perlen von 1 bis q beginnen. Nach dem ersten Codierungssymbol können wieder alle p Perlen verwendet werden. Für insgesamt n Eingabesymbole und p Perlen, ist die als optimal gefundene Länge für den gesamten Text also $\text{GREEDYLÄNGE}(1, n, p)$.

Nun verwenden wir einen dynamischen Ansatz, um die Tabelle GREEDYLÄNGE zu füllen. Dabei ist die zugrundeliegende Idee, dass dynamisch kleinere Teilbäume zusammengesetzt werden. Betrachten wir beispielsweise den optimalen Codierungsbaum für das Intervall von i_{start} bis i_{end} mit Perlen $1, \dots, q$. Dieser hat entsprechend bis zu q abgehende Kanten. Das Problem kann nun annähernd optimal zerlegt werden, indem für ein k im Intervall die ersten Eingabesymbole von i_{start} bis k in einem Teilbaum mit der Perle q untergebracht werden. Die übrigen Eingabesymbole von $k+1$ bis i_{end} können dann höchstens die Perle $q-1$ verwenden, um noch einen gültigen Codierungsbaum zu ergeben. An der Kante mit der Perle q kann dann wieder ein Teilbaum angehängt werden, der alle verfügbaren Perlen verwendet. Nun minimiert man über alle möglichen Einteilungen k und kann dadurch die Greedy Länge ausrechnen. Das Verfahren ist in der folgenden Gleichung graphisch dargestellt:

$$\text{GREEDYLÄNGE}(i_{\text{start}}, i_{\text{end}}, q) = \min_k$$


Die Gesamtkosten dieses Codewortbaums ergeben sich aus den Kosten, den linken Teilbaum zusätzlich mit der Perle q zu versehen und den Kosten des rechten Teilbaums, also

$$\text{GL}_k(i, j, q) = \text{GL}(i, k, p) + \text{GL}(k+1, j, q-1) + \text{Durchmesser}(q) \times \sum_{l=i}^k \text{Häufigkeit}(l)$$

In der Umsetzung müssen wir die folgenden Beobachtungen berücksichtigen:

Algorithmus 1 Dynamischer Greedy n -ary Huffman**Require:** Sortierte Häufigkeiten der n Eingabesymbole und Größe der p Perlen

```

function GREEDYLÄNGE( $i_{\text{start}}, i_{\text{end}}, 1$ )
  if  $q = 1$  then
    return Durchmesser( $1$ )  $\times \sum_{l=i_{\text{start}}}^{i_{\text{end}}} \text{Häufigkeit}(l) + \text{GREEDYLÄNGE}(i_{\text{start}}, i_{\text{end}}, p)$ 
  Minimum  $\leftarrow \infty$ 
  if  $q > 2$  then  $\triangleright$  Mehr als 2 Perlen verfügbar
    Minimum  $\leftarrow \text{GREEDYLÄNGE}(i_{\text{start}}, i_{\text{end}}, q - 1)$   $\triangleright$  Teilbaum ohne Perle  $q$ 
  for  $k$  im Intervall  $i_{\text{start}}$  bis  $i_{\text{end}} - 1$  do
    Minimum  $\leftarrow \min\{ \text{GL}_k(i, j, q), \text{Minimum} \}$ 
  return Minimum

```

- Steht nur noch eine Perle zur Verfügung, also $q = 1$, muss diese für das gesamte Intervall zur Codierung verwendet werden.
- Wenn $i_{\text{start}} = i_{\text{end}}$ muss nur ein Element kodiert werden. Dafür ist keine weitere Perle notwendig und entsprechend ist $\text{GREEDYLÄNGE}(i, i, \star) = 0$ unabhängig vom dritten Argument.
- Wenn mehr als zwei Perlen noch zur Verfügung stehen, also $q > 2$, kann es sich lohnen, nicht alle Perlen zu verwenden. Ein Beispiel für einen solchen Fall ist, wenn noch 5 Perlen zur Codierung von 3 Wörtern zur Verfügung stehen; die beiden größten können dann gekonnt ignoriert werden.

In Algorithmus 1 ist der entsprechende Algorithmus als Pseudocode gegeben.

Post-Processing. Das Greedy Verfahren produziert nicht optimale Ergebnisse. Insbesondere produziert es aber auch Codierungen, die nicht der Codewortlänge nach sortiert sind. Bevor also Codewörter und Eingabesymbole zugeordnet werden können, müssen die Codewörter noch einmal ihrer Länge nach sortiert werden. Danach können jeweils den häufigsten Eingabesymbolen die kürzesten Codewörter zugeordnet werden.

Laufzeit. Die Tabelle GreedyLänge hat $n \times n \times p$ Einträge. Für jeden Eintrag müssen höchstens $k \leq n$ mögliche Positionen für die Einteilung überprüft werden. Insgesamt ergibt sich also eine Laufzeit von $\mathcal{O}(pn^3)$ und ein Speicheraufwand von $\mathcal{O}(pn^2)$.

Rekonstruktion. In jedem Schritt des Greedy Verfahrens kann mitgeschrieben werden, an welcher Stelle k im Intervall die Einteilung stattgefunden hat. So kann am Ende der Codierungsbaum für die gefundene Lösung erhalten werden.

Beispiel

Anhand des Beispiels aus der Aufgabenstellung wollen wir hier ein paar ausgewählte Schritte des Verfahrens erklären. Dafür ist in Abbildung 1.2 der finale Codierungsbaum dargestellt, wobei jeweils die Einträge von GreedyLänge über den Knoten aufgeführt sind. Im ersten Schritt von Algorithmus 1 muss das k gefunden werden, so dass die Länge des Codes minimal wird, wenn die Codewörter der Eingabesymbole vom ersten (K) bis zum k -ten mit einer roten Perle

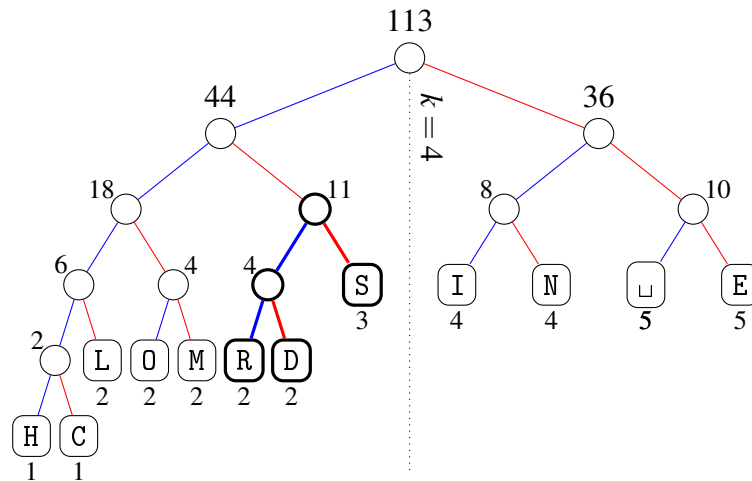


Abbildung 1.2: Der vom Greedy Verfahren erzeugte Codierungsbaum für das Beispiel aus der Aufgabenstellung. Beginnend vom Wurzelknoten gibt die Farbe der Kante jeweils an, ob eine blaue (links) oder rote (rechts) Perle verwendet wird. Unter den Eingabesymbolen (Blätter des Baums) ist deren absolute Häufigkeit angegeben; sie sind nach rechts aufsteigend sortiert. Über jedem internen Knoten ist der dazugehörige Wert von GREEDYLÄNGE angegeben (siehe Haupttext). Die erste optimale Teilung findet für $k = 4$ statt. Die Berechnung der Einträge von GreedyLänge im fett markierten Teilbaum wird im Haupttext erklärt.

beginnen und die vom $k + 1$ -ten bis zum letzten (H) mit einer blauen beginnen. In unserem Beispiel ist das für $k = 4$ gegeben, also beginnen die Codewörter von E, U, N und I jeweils mit einer roten Perle und alle anderen mit einer blauen. Diese Einteilung wurde gefunden, in dem zunächst für alle Einteilungen die Einträge in GreedyLänge berechnet wurden; am Ende ist dann

$$\underbrace{\text{GL}(1, n = 33, p = 2)}_{133} = \underbrace{\text{GL}(1, 4, 2)}_{36} + \underbrace{\text{GL}(5, 33, 1)}_{44+1 \times 15} + \underbrace{\text{Durchmesser}(2) \times \sum_{l=1}^4 \text{Häufigkeit}(l)}_{1 \times 18}$$

die kürzeste Länge. Zur Erklärung der Einträge in GreedyLänge betrachten wir den in Abbildung 1.2 fett markierten Teilbaum mit den Eingabesymbolen S (Index $i = 5$), D ($i = 6$) und R ($i = 7$). Für diesen wollen wir $\text{GL}(i_{\text{start}} = 5, i_{\text{end}} = 7, q = 2)$ berechnen, also die beste Codierung der drei Symbole mit den beiden verfügbaren Perlen. Zunächst für den linken Teilbaum, in diesem sind nur R und D codiert, also ist nach der Aufteilung des Baums zwischen den beiden Elementen

$$\text{GL}(6, 7, 2) = \underbrace{\text{GL}(7, 7, 1)}_2 + \underbrace{\text{GL}(6, 6, 2)}_0 + \underbrace{\text{Durchmesser}(2) \times \text{Häufigkeit}(\text{D})}_{1 \times 2} = 4.$$

Diese Formel beschreibt dabei lediglich das, was zu erwarten ist: codiere ich zwei Symbole mit den beiden Perlen, so kann ich dem ersten Symbol die rote und dem zweiten die blaue zuweisen; die Gesamtlänge des codierten Textes ist dann lediglich die absolute Häufigkeit der beiden Symbole addiert.

Genauer betrachtet ergeben sich die Terme wie folgt: für das Symbol R bei Index 7 greift die

Regel, dass es in einem Teilbaum mit nur einer verfügbaren Perle ($q = 1$) ist; dementsprechend muss zunächst erst diese angehängt werden, was die Gesamtkosten 2 erzeugt. Das Symbol D befindet sich nach der Aufteilung in einem Teilbaum mit mehr als zwei verfügbaren Perlen und bedarf keiner weiterer Kosten, weil $GL(i, i, \star) = 0$. Zum Schluss müssen noch die Kosten zum Anhängen des Blatts D berücksichtigt werden, wofür alle Symbole im rechten Teilbaum (lediglich D) die zusätzlichen Kosten $Durchmesser(2) = 1$ erzeugen.

Im nächsten Schritt soll nun zusätzlich das S berücksichtigt werden. Dafür wird vor den linken Teilbaum eine blaue Perle geschaltet, dem S wird eine rote Perle zugewiesen. Der gesamte Teilbaum hat damit die Codewortlänge

$$GL(5, 7, 2) = \underbrace{GL(6, 7, 1)}_{1 \times 4 + GL(6, 7, 2)} + \underbrace{GL(5, 5, 2)}_0 + \underbrace{Durchmesser(2) \times Häufigkeit(S)}_{1 \times 3} = 11,$$

was eben wieder der Länge entspricht, die sich ergibt, wenn wir an den linken Teilbaum eine blaue und an den rechten eine rote Perle anhängen. Nach dem selben Prinzip lassen sich alle anderen Einträge der Tabelle in Abbildung 1.2 nachrechnen.

1.3 ILP-Ansatz

Ein Algorithmus um die optimale Kodierung zu erhalten, wurde von Richard M. Karp in einem 1960 erschienenen Aufsatz² erläutert. Dabei wird das Finden des Optimalen Codes in ein Integer Linear Program übersetzt, dessen Struktur im Folgenden erklärt werden soll.

Integraler Bestandteil der ILP Formulierung sind die Variablen $a_j \geq 0$ und $b_j \geq 0$ mit $j > 0$. Dabei bezeichnet a_j die Anzahl an Codewörter, die die Länge j haben. Entsprechend muss $a_0 = 0$ sein, da kein Codierungssymbol die Länge 0 haben sollte. Die Größe b_j beschreibt die Anzahl der Präfixe, die die Länge j haben; für $j = 0$ gibt es genau einen gültigen Präfix $b_0 = 1$: das leere Wort.³ Durch Anfügen von Codierungssymbolen an bestehende Präfixe können insgesamt

$$\sum_{k=1}^m b_{j-\text{Länge}(c_k)}$$

Präfixe b_j oder Codewörter a_j erhalten werden. Damit ergibt sich die Randbedingung für gültige Codes als

$$a_j + b_j \leq \sum_{k=1}^m b_{j-\text{Länge}(c_k)},$$

wobei die Ungleichheit ermöglichen soll, dass nicht alle Codewörter genutzt werden. Die zu

²R. Karp, "Minimum-redundancy coding for the discrete noiseless channel," in IRE Transactions on Information Theory, vol. 7, no. 1, pp. 27-38, January 1961, doi: 10.1109/TIT.1961.1057615

³Angenommen zur Codierung stehen insgesamt drei Perlen zur Verfügung, von denen zwei den Durchmesser 1 haben, die zweite sei größer. Dann können auf der ersten Ebene bis zu zwei gültige Codewörter entstehen, was sich in den Gleichungen durch $a_1 + b_1 \leq 2b_0$ widerspiegelt. Beispiellösungen sind $a_1 = 0, b_1 = 2$ (kein abgeschlossenes Codewort, zwei offene Präfixe); $a_1 = 1, b_1 = 1$ (ein abgeschlossenes Codewort, ein offener Präfix) oder $a_1 = 2, b_1 = 0$ (zwei abgeschlossene Codewörter, kein offener Präfix). Alles das sind Beschreibungen sinnvoller, aber nicht unbedingt kürzester Codebäume. Die Länge wird dann durch die Kostenfunktion beschrieben.

Algorithmus 2 ILP Formulierung nach Karp**Require:** Sortierte Häufigkeiten der n Eingabesymbole und Größe der p Perlen**Minimiere**

$$\text{Länge}(T) = \sum_j \sum_{i=1}^{\text{Offen}(j)} \text{Häufigkeit}(e_i)$$

mit Randbedingungen

$$\begin{aligned} a_j + b_j &\leq \sum_{k=1}^m b_{j-\text{Länge}(c_k)}, & a_0 &= 0, \quad b_0 = 1, \\ \text{Offen}(j) &= \text{Offen}(j-1) - a_j, & \text{Offen}(0) &= n. \end{aligned}$$

minimierende Kostenfunktion sind die Gesamtkosten des Codes, also $\text{Länge}(\text{Text})$ aus Gleichung (1.1).

Um die Längen der konkreten der Codewörter $\text{Länge}(w_i)$ auszurechnen, kann entweder wie von Karp vorgeschlagen eine Erweiterungen der a_i auf die Matrix

$$y_{ij} = \begin{cases} 1 & \text{wenn } \text{Länge}(w_i) = j \\ 0 & \text{sonst,} \end{cases}$$

oder eine Zählfunktion nach M. Golin und G. Rote genutzt werden, welche die noch offenen Eingabesymbole wie folgt zählt:

$$\text{Offen}(j) = \text{Offen}(j-1) - a_j.$$

Dabei sind am Anfang alle Eingabesymbole noch offen, also $\text{Offen}(0) = n$. Dann ist entsprechend

$$\text{Länge}(T) = \sum_j \sum_{i=1}^{\text{Offen}(j)} \text{Häufigkeit}(e_i)$$

die zu minimierende Kostenfunktion. Damit ergibt sich das in Algorithmus 2 formulierte Lineare Programm. Dieses kann mit modernen ILP-Solvern wie beispielsweise HiGHS⁴ gelöst werden.

Laufzeit. Die maximale Tiefe des Codebaumes für n Eingabesymbole ist durch den Durchmesser P der größten Perle und die Anzahl p von Perlen über $P \log_p n$ beschränkt. Dies entspricht einem Baum mit p Kanten pro Knoten, die jeweils P lang sind. Auf jeder Ebene können bis zu n Codewörter offen sein, also können wir die Gesamtlaufzeit als $\mathcal{O}(n^{P \log_p n})$ schätzen.

⁴<https://highs.dev/>

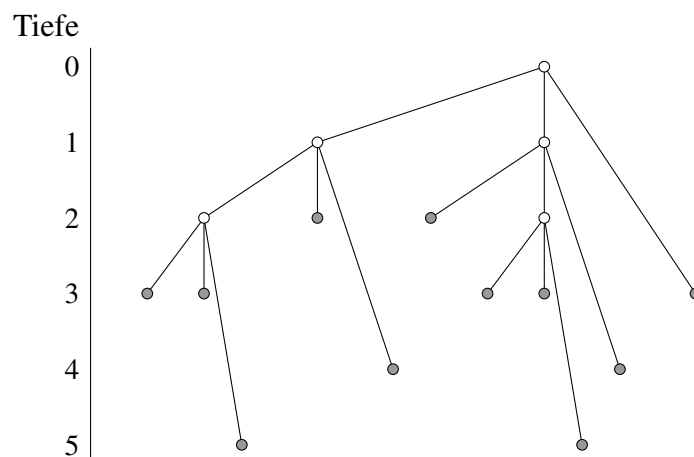


Abbildung 1.3: Ein Codierungsbaum mit 16 Knoten der sich aus den drei Perlen mit Längen 1, 1 und 3 ergibt; entsprechend hat jeder interne Knoten exakt drei Kinder. Abgeschlossene Codewörter sind grau markiert. Die Tiefe eines Knoten ergibt sich aus der Länge des entsprechenden Präfixes oder Codeworts, welchen er repräsentiert.

1.4 Quasi-Polynomieller Ansatz

Ein Ansatz, der polynomiell in der Anzahl n der Wörter im Eingabealphabet aber exponentiell in der größten Größe der Perlen ist, wurde von M. Golin und G. Rote⁵ beschrieben. Dabei wird die Suche nach dem optimalen Code als Suche nach dem kürzesten Weg auf einem Graphen umformuliert. Entsprechend wird im Folgenden zunächst diese Graphenstruktur erläutert. Danach werden Anpassungen diskutiert, die es ermöglichen, den Algorithmus auch für große Eingabealphabete zu verwenden.

Graphenstruktur. Ähnlich wie im ILP Ansatz wird für den Graphen ein Baum in einer bestimmten Tiefe durch zwei Eigenschaften charakterisiert: Erstens seine Anzahl an bereits abgeschlossenen Knoten beziehungsweise für ein Code die Anzahl an bisher festgelegten Codewörtern. Zweitens seine Anzahl an noch offenen Knoten, an denen später Teilbäume angehängt werden können. Die Kombination dieser beiden Eigenschaften wird Signatur genannt. Abweichend von der Skizze in Abbildung 1.1 müssen die Knoten dafür ebenfalls mit einer Tiefe versehen werden, welche angibt, wie lang der bisher entstandene Präfix ist. In Abbildung 1.3 ist so ein Codierungsbaum, welcher mit einer Tiefe versehen wurde, skizziert. Sei P die Länge der größten Perle, dann definieren wir die Signatur eines Baums T als

$$\text{sig}_i(T) = (m; l_1, \dots, l_P)$$

wobei m die Anzahl an bereits festgelegten Codewörtern in T bis zur Tiefe i ist und l_1, \dots, l_P die jeweilige Anzahl der noch offenen Präfixe der Tiefe $1, \dots, P$, ausgehend von dieser Stufe sind. Für den Baum in Abbildung 1.3 sind die entsprechenden Signaturen in Abbildung 1.4 dargestellt.

⁵M. J. Golin and G. Rote, "A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs," in IEEE Transactions on Information Theory, vol. 44, no. 5, pp. 1770-1781, Sept. 1998, doi: 10.1109/18.705558

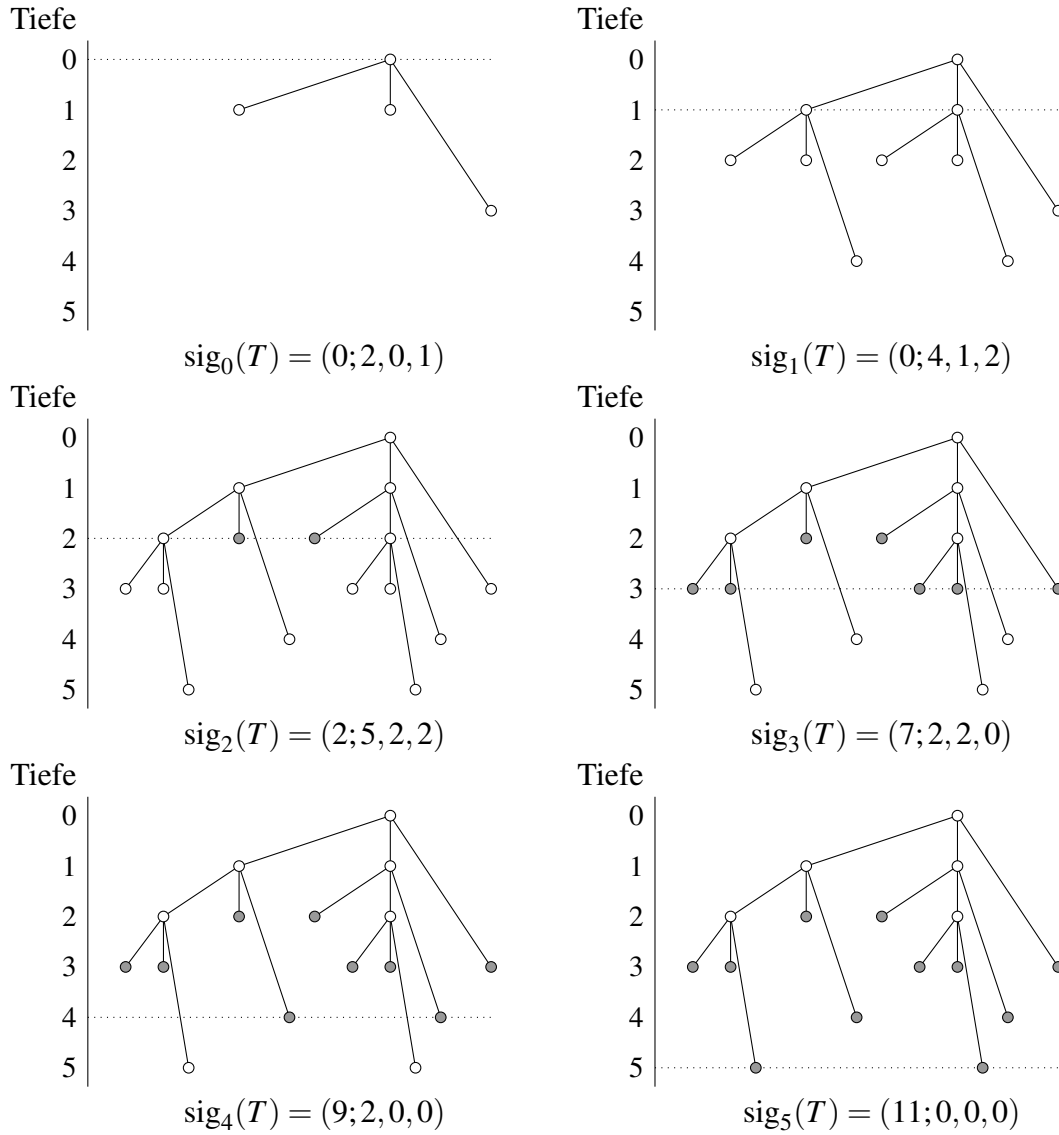


Abbildung 1.4: Die verschiedenen Signaturen für den Codierungsbaum aus Abbildung 1.3. Da die höchste Perle die Länge 3 hat, ist $P = 3$. Entsprechend hat jede Signatur die Form $\text{sig}_i(T) = (m; l_1, l_2, l_3)$. Dabei gibt m die Anzahl abgeschlossener Codewörter (grau) mit Tiefe $\leq i$ an. Die l_1 , l_2 und l_3 sind jeweils die Anzahl noch offener Knoten mit Tiefe $i+1$, $i+2$ und $i+3$. Beispielsweise ist $\text{sig}_2(T) = (2; 5, 2, 2)$, da sich auf Ebene 2 genau 2 abgeschlossene Knoten befinden und dann in Tiefe 3, 4 und 5 jeweils 5, 2 und 2 noch offene Knoten.

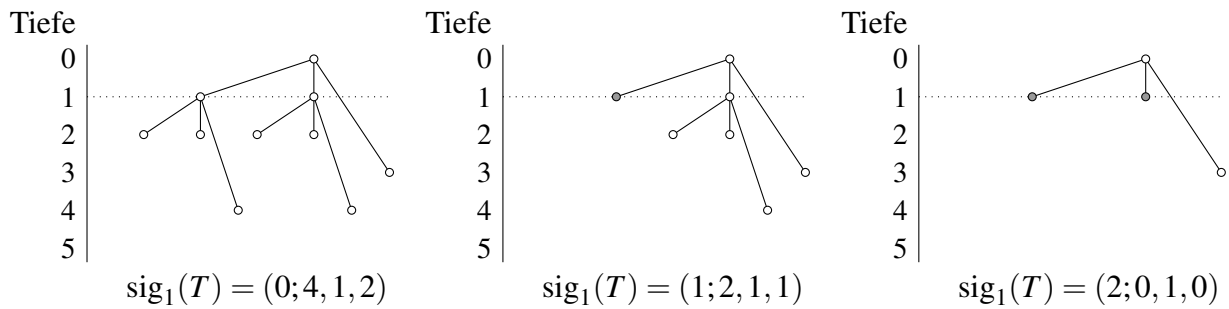


Abbildung 1.5: Die verschiedenen Signaturen auf der ersten Ebene $\text{sig}_1(T)$ für den Codierungsbaum aus Abbildung 1.3. Die Anzahl abgeschlossener Codewörter (grau) mit Tiefe ≤ 1 kann entweder 0, 1 oder 2 sein, woraus sich die drei Möglichkeiten ergeben, der Rest der Signatur ergibt sich wie in Abbildung 1.4.

Zur Suche nach dem kürzesten Code für einen Eingabetext verwenden wir nur einen Graphen, dessen Knoten mögliche Codierungsbaumsignaturen sind. Für ein wohldefiniertes Suchproblem muss nun noch eine Übergangsregel zwischen Signaturen festgelegt werden, die später angibt, welche Kanten im Graphen existieren. Außerdem muss noch eine Kostenfunktion festgelegt werden, welche die Länge eines Codes mit der aktuellen Signatur angibt.

Übergangsregel. Erhöht man die Tiefe einer Signatur um eins, werden aus den Knoten l_1 solche auf der aktuellen Stufe, aus welchen der Tiefe 2 werden welche auf Tiefe 1 und so weiter. Die Signatur ändert sich also wie folgt:

$$\text{sig}_i = (m; l_1, \dots, l_p) \xrightarrow{\text{Nächste Tiefe}} \text{sig}_{i+1} = (m + l_1; l_2, \dots, l_p, 0).$$

Von den l_1 Knoten, die jetzt zusätzlich auf der aktuellen Ebene sind, können an $0 \leq q \leq l_1$ ein neuer Teilbaum angehängt werden, wodurch diese als interner Knoten verbleiben. Dafür kommen p (Gesamtzahl der Perlen) neue Knoten hinzu, welche jeweils den entsprechenden Durchmesser der Perle tiefer als der entfernte Knoten hängen. Mathematisch lässt sich das wie folgt ausdrücken: Sei $d = (d_1, \dots, d_p)$ ein Vektor, der in jedem Eintrag d_i angibt, wie viele Perlen mit Durchmesser i es gibt. Dann ändert sich eine Signatur durch anfügen von q Perlen wie folgt:

$$\text{sig} = (m; l_1, \dots, l_p) \xrightarrow{q \text{ Teilbäume anhängen}} \text{sig}' = (m - q; l_1 + qd_1, \dots, l_p + qd_p).$$

Der Vektor d heißt auch *charakteristischer Vektor*.

Zusammengenommen kann eine Signatur $(m; l_1, \dots, l_p)$ durch das Erhöhen der Tiefe um eins und das Anfügen von $0 \leq q \leq l_1$ Teilbäumen zu folgenden Signaturen werden:

$$(m; l_1, \dots, l_p) \xrightarrow{q} (m + l_1 - q; l_2 + qd_1, \dots, l_p + qd_{p-1}, qd_p) \quad (1.2)$$

Jede Signatur die so von einer in die andere überführbar ist, verbinden wir im Graphen durch eine Kante. In Abbildung 1.5 sind beispielhaft Signaturen für den Baum aus Abbildung 1.3 gegeben, die sich auf der ersten Ebene ergeben können.

Kostenfunktion. Wird die Tiefe um eins erhöht, so müssen alle noch nicht zugeordneten Eingabesymbole mindestens ein Codewort erhalten, das noch einen Millimeter länger als die aktuelle Tiefe ist. Entsprechend kommt beim Übergang von einer Signatur $(m; l_1, \dots, l_p)$ zu einer anderen $(m'; l_1, \dots, l_p)$ zusätzlich die Häufigkeit der verbleibenden Eingabesymbole (in Millimeter) hinzu, also

$$\text{Länge}(m'; l_1, \dots, l_p) = \text{Länge}(m; l_1, \dots, l_p) + \sum_{i=m+1}^n \text{Häufigkeit}(e_i). \quad (1.3)$$

Start und Ende. Der Codierungsbaum, welchen wir am Ende tatsächlich erhalten wollen, hat die Signatur $\text{sig}_* = (n; 0, \dots, 0)$, weil für jedes der n Eingabesymbole ein Codewort verfügbar sein muss. Anfangs gibt es noch keine abgeschlossenen Codewörter und die Startsignatur ist $\text{sig}_0 = (0; d_1, \dots, d_p)$, weil beginnend vom leeren Wort einmal jede Perle angefügt werden kann.

Reduzieren einer Signatur. Manchmal kommt es vor, dass eine Signatur $(m; l_1, \dots, l_p)$ nicht mehr zielführend ist, weil die Gesamtanzahl von bereits festgelegten Codewörtern m und der noch offenen Knoten $\sum_i l_i$ die Gesamtzahl an Eingabesymbolen n überschreitet. Sobald also

$$m + \sum_i l_i \geq n \quad (1.4)$$

müssen die l_i vom hintersten Eintrag l_p beginnend so lange verringert werden, bis die obige Bedingung wieder erfüllt ist. Dieses Verfahren wird REDUZIEREN einer Signatur genannt.

Datenstrukturen und Praktisches. Beginnend von der Startsignatur sig_0 können durch die Vorschrift in Gleichung (1.2) sukzessive mögliche neue Signaturen von Codierungsbäumen erhalten werden. Für diese kann dann wieder Gleichung (1.2) angewandt werden; schreibt man dabei für jede erreichte Signatur die jeweils kürzeste benötigte Codelänge aus Gleichung (1.3) mit, kann am Ende vom der benötigten Codelänge zum Zielknoten sig_* der kürzeste Code rekonstruiert werden. Dafür verwenden wir eine Zuordnungstabelle OPT (Assoziatives Array, Map), in dem für jede Signatur die kürzeste bisher benötigte Codelänge gespeichert wird.

Dabei müssen die Signaturen so abgelaufen werden, dass zunächst alle erreichbaren Signaturen der abgeschlossenen Knotenanzahl m nach aufsteigend abgearbeitet werden. Die optimale Codelänge OPT für eine Signatur mit $m' > m$ kann sich nämlich durch einen noch günstigeren aber noch nicht begangenen Pfad im Graphen noch ändern. Andersherum hat eine Signatur mit m keinen Einfluss mehr auf Signaturen mit weniger festgelegten Knoten $m' < m$, weil diese eben bereits fixiert sind. Genauso verhält es sich für die Anzahl an noch nicht festgelegten Knoten l_1, \dots, l_p , wo jeweils Knoten mit niedrigeren l_1 , dann welche mit niedrigeren l_2 usw. bevorzugt werden müssen. Entsprechend können die Signaturen der Reihenfolge nach abgearbeitet werden, wenn durch die lexikographisch sortierte Liste von Vektoren

$$(m, m + l_1 + 1, m + l_1 + l_2 + 2, \dots, m + l_1 + \dots + l_p + P) \quad (1.5)$$

iteriert wird. Lexikographische Ordnung heißt hierbei, dass jeder Eintrag strikt kleiner als sein rechter Nachbar ist. In dem man jeweils Nachbarn voneinander subtrahiert kann die Signatur abgelesen werden. Der obige Vektor hat $P + 1$ Einträge, die jeweils von 0 bis $n + P + 1$ reichen,

Algorithmus 3 Quasi-Polynomieller Ansatz nach M. Golin und G. Rote

Require: Sortierte Häufigkeiten der n Eingabesymbole, GREEDYLÄNGE aus Algorithmus 1 und charakteristischer Vektor $d = (d_1, \dots, d_P)$ der Perlen

Initialisiere Heap $\mathcal{H} = \{\text{sig}_0\}$ mit Startsignatur $\text{sig}_0 = \text{REDUZIERE}(0; d)$

Initialisiere $\text{OPT}(\text{sig}_0) = 0$

while Heap \mathcal{H} nicht leer **do**

 Entferne kleinste Signatur $\text{sig} = (m; l_1, \dots, l_P)$ aus \mathcal{H}

 Berechne neue Codelänge $C = \text{Opt}(\text{sig}) + \sum_{i=m+1}^n \text{Häufigkeit}(e_i)$ siehe Gleichung (1.3)

 Überspringe wenn $C \geq \text{GREEDYLÄNGE}$

for q in 0 bis l_1 **do**

 Berechne neue Signatur $\text{sig}' = (m'; l'_1, \dots, l'_P) \stackrel{q}{\leftarrow} (m; l_1, \dots, l_P)$

 Wenn Gleichung (1.4) verletzt, setze $\text{sig}' \leftarrow \text{REDUZIERE}(\text{sig}')$

 Setze $\text{OPT}(\text{sig}') = \min\{\text{OPT}(\text{sig}'), C\}$

 Füge sig' in Heap \mathcal{H} ein, wenn nicht vorhanden

entsprechend gibt es insgesamt

$$\binom{n+P+1}{P+1}$$

solche Vektoren, was $\mathcal{O}(n^{P+1})$ entspricht. Durch kombinatorische Überlegungen kann jedem Vektor in Gleichung (1.5) eindeutig ein Index zugewiesen werden.⁶ Um bei jedem Schritt die lexikographisch nächstkleinste Signatur zu betrachten, werden diese in einen Heap einsortiert.

Die aus dem Greedy Verfahren in Algorithmus 1 gewonnene Codelänge kann als obere Schranke für das Verfahren verwendet werden. Durch diesen Branch-and-bound Schritt können viele nicht zielführende Signaturen bereits frühzeitig abgebrochen werden.

In Algorithmus 3 ist entsprechender Pseudocode gegeben.

Laufzeit. Wie oben beschrieben gibt es $\mathcal{O}(n^{P+1})$ Signaturen, was gleichzeitig den Speicherbedarf darstellt. Für jede Signatur kommen $l_1 = \mathcal{O}(n)$ Kandidaten in Frage, die sich aus dieser ergeben. Entsprechend ist die Laufzeit $\mathcal{O}(n^{P+2})$.

1.5 Erweiterungen

Für Pedram und Sybille stellt das Auffädeln der Perlen nun den größten Zeitaufwand dar. Außerdem merken beide, dass ihre Kunden vor allem den sentimentalen Wert der Kette schätzen und sich keiner von ihnen die Mühe macht, die Nachricht wirklich anhand des Beipackzettels zu entschlüsseln. Viel mehr zählt für die Kunden einzig und allein, dass die Nachricht dekodierbar

⁶Dadurch, dass die Vektoren in Gleichung (1.5) nach rechts aufsteigend sortiert sind, lässt sich Ihnen eindeutig ein Index zuordnen, beispielsweise ist $(0, 1, \dots, n+P)$ der erste, $(0, 1, \dots, n+P+1)$ der zweite und so weiter. Umgekehrt kann einem Index ein Vektor über die folgende Beobachtung zugeordnet werden: hat der Vektor an der Stelle j den Wert i , so kommen für die Einträge rechts von ihm $\binom{n+P+1-i}{P+1-j}$ Kombinationen in Frage. Beginnend von der ersten Stelle, wird der dortige Eintrag im Vektor so lange erhöht, bis die Anzahl in Frage gekommener Kombinationen größer als der Index ist. Dann ist der Eintrag im Vektor genau der vorherige vom aktuellen. Nun kann man zum nächsten Eintrag gehen und sich so sukzessive den Vektor rekonstruieren.

ist. So kommen die beiden auf die Idee, einen noch effizienteren Code, als einen präfixfreien Code, zu verwenden. Dieser soll weiterhin eindeutig dekodierbar sein, wenn auch nicht mehr unbedingt von Hand. Beide entscheiden sich also ihren Kunden zusätzlich die Möglichkeit der arithmetischen Kodierung anzubieten, wobei insgesamt weniger Perlen verwendet werden müssen.

Der Nachteil eines präfixfreien Code ist es, dass die Eingabesymbole eine fixe ganzzahlige Codierungslänge haben, man optimalerweise aber lieber reelle Codierungslängen benutzen würde. Die Idee der arithmetischen Codierung ist es nun, der gesamten Nachricht ein Intervall zwischen $[0, 1]$ zuzuordnen. Dieses kann dann im Codierungsalphabet oder im Eingabealphabet dargestellt werden. Insgesamt handelt es sich also nur um einen Wechsel der Darstellung, wobei eine erhöhte Komprimierung im Vergleich zum präfixfreien Code durch den Erhalten des Kontexts ermöglicht wird.

Wir beschreiben nun die beiden Verfahren, um einerseits eine Zeichenfolge in einem Alphabet \mathcal{A} mit n Buchstaben a_i in ein Intervall (u, v) mit rationalen Zahlen $u, v \in \mathbb{Q}$ zu übersetzen und andererseits ein Intervall (u', v') in einen Text zu übersetzen. Im Sinne der Aufgabe könnte man dann den Codierungsschritt als Übersetzung vom \mathcal{E} in die rationale Zahl und dann ins Codierungsalphabet \mathcal{C} verstehen, also

$$\mathcal{E} \text{ codiert in } \mathcal{C} = \text{Text in } \mathcal{E} \xrightarrow{\text{übersetzt in}} \text{rationales Intervall } (u, v) \xrightarrow{\text{übersetzt in}} \text{Text in } \mathcal{C}.$$

Der Decodierungsschritt ist entsprechend einfach das Verfahren umgekehrt, also

$$\mathcal{C} \text{ dekodiert in } \mathcal{E} = \text{Text in } \mathcal{C} \xrightarrow{\text{übersetzt in}} \text{rationales Intervall } (u', v') \xrightarrow{\text{übersetzt in}} \text{Text in } \mathcal{E}.$$

Der Unterschied zur üblichen arithmetischen Codierung ist, dass üblicherweise ein Binäralphabet (Bits) oder mindestens ein Alphabet mit gleich teuren Codierungssymbolen für die Darstellung von (u, v) verwendet wird. Dadurch reicht es beispielsweise die Mitte des Intervalls in der entsprechenden Binärbasis darzustellen, was jedoch für die unterschiedlich breiten Perlen nicht ausreicht.

Decodierungsschritt. Wir wollen das rationale Intervall (u, v) für eine Abfolge von Zeichen im Codierungsalphabet \mathcal{A} finden. Dafür wird für die n Symbole a_i das Intervall $[0, 1]$ in n Teilintervalle $(x_i, y_i]$ untergliedert, wobei $x_0 = 0$, $y_n = 1$ und $y_i = x_{i+1}$ gelten muss. Die Idee ist nun, die Intervalle der Codierungssymbole so in einer zu verschachteln, dass (u, v) immer näher eingegrenzt wird. Dafür muss zunächst das Intervall $(x_i, y_i]$ und damit das Codierungssymbol a_i gefunden werden, in dem (u, v) liegt. Das entsprechende Symbol wird dann an den Code angehängt. Liegt (u, v) in zwei oder mehr Intervallen, ist der finale Code gefunden. Ist dies nicht der Fall, wird das Intervall $[x_i, y_i)$ auf $[0, 1)$ vergrößert (es wird “hereingezoomt”), so dass die Grenzen des Intervalls $x_i \rightarrow 0$ und $y_i \rightarrow 1$ werden. Damit ändern sich u und v zu

$$u' = \frac{u - x_i}{y_i - x_i}, \quad v' = \frac{v - x_i}{y_i - x_i}. \quad (1.6)$$

Zu beachten ist hierbei, dass die Perlennachricht nach diesem Verfahren nachdem sie einmal codiert und dann decodiert wurde, etwas länger sein kann. Entsprechend muss die ursprüngliche Länge ebenfalls vermerkt werden, nach der die Decodierung abgebrochen werden kann.

Codierungsschritt. Um einem Text im Codierungsalphabet \mathcal{A} nun ein Intervall (u, v) zuzuordnen, wird das Verfahren effektiv rückwärts durchlaufen. Wieder muss für jedes Symbol a_i ein Teilintervall $[x_i, y_i)$ gefunden werden. Dann beginnt man bei $(u, v) = [0, 1]$ und schachtelt das Intervall für jedes auftretende Symbol a_i so, dass (u, v) auf den durch $[x_i, y_i)$ gegebenen Teilabschnitt eingeschränkt wird. Entsprechend ist

$$u' = u + x_i(u - v), \quad v' = u + y_i(u - v). \quad (1.7)$$

So kann einmal durch den gesamten Text verfahren werden.

Optimale Intervallbreiten. Die Intervallbreiten sollten so gewählt werden, dass sie der relativen Häufigkeit der Alphabetsymbole entsprechen. Für das Eingabealphabet \mathcal{E} können die optimalen Intervallbreiten also nach der Häufigkeit der Eingabesymbole e_i festgelegt werden. Dort gilt entsprechend

$$\text{Intervallbreite}(e_i) = y_i - x_i = \frac{\text{Häufigkeit}(e_i)}{\text{Gesamtanzahl Eingabesymbole}}.$$

Für die Codierungssymbole ist die Zuordnung etwas komplizierter und stark verwandt mit der Kapazität t des Perleninformationskanal in Abschnitt 1.1. Angenommen t sei die Breite des Intervalls einer Perle mit Durchmesser von 1 mm. Dann wäre es wünschenswert, wenn diese Perle doppelt so Häufig genutzt wird, wie eine Perle mit 2 mm, dreimal so oft wie eine mit 3 mm Durchmesser und so weiter. Die Intervalle müssen also jeweils die relativen Gewichte t^2, t^3, \dots haben, damit sie doppelt, dreifach, ... so selten getroffen werden. Am Ende sollen die Intervalle insgesamt $[0, 1]$ ausfüllen, die Intervallbreiten müssen sich also zur 1 addieren. Die optimale Breite t erfüllt dementsprechend die Gleichung

$$t^{\text{Durchmesser}(1)} + \dots + t^{\text{Durchmesser}(p)} = 1,$$

was exakt der Gleichung für die Kanalkapazität entspricht. Alternativ kann die optimale Kanalbreite auch dadurch gefunden werden, dass für die Intervalle der Perlen mit Breite b_i der relative Informationsgewinn pro Längeneinheit maximiert wird. Der Informationsgewinn ist die Entropie

$$S(b_i) = - \sum_i b_i \log b_i,$$

die mittlere Länge ist

$$\text{Länge}(b_i) = \sum_i b_i \times \text{Durchmesser}(i).$$

Maximiert man nun $S(b_i)/\text{Länge}(b_i)$ erhält man die gleiche Einteilung wie nach der optimalen Breite t .

Beispiel

Anhand des Beispiels aus der Aufgabenstellung, soll hier das Verfahren noch einmal erläutert werden. Die Intervallbreiten haben wir bereits vorher durch $t = 0.5$ gelöst, wir weisen also der

bbbbrrbbrrbrbbrbr mit einer Gesamtlänge von 112 mm. Um das Codewort zu entschlüsseln, kann die gesamte Prozedur rückwärts durchgeführt werden. Der Beipackzettel von Sybille und Pedram sollte dann (i) die entsprechenden Intervalle sowohl im Eingabealphabet als auch im Codierungsalphabet und (ii) die Gesamtlänge des Ursprungstexts enthalten. Für das Beispiel ist dies in Tabelle 1 gegeben.

1.6 Beispiele

In Tabelle 2 sind die Ergebnisse aufgeführt; diese lassen sich mit Ausnahme von *schmuck9.txt* innerhalb weniger Millisekunden berechnen. Um diese Datei mit Algorithmus 3 zu lösen, sind wenige Stunden notwendig. Auf den folgenden Seiten sind beispielhafte Beipackzettel für die Eingabedateien von der Seite des BWINF angehängen. In diesen sind jeweils zuerst das entsprechende Symbol und dann der Perlencode aufgeführt. Der Perlencode ist nicht in Farben, sondern in Zahlen gegeben, wobei eine 1 für die erste Perle usw. steht. In *schmuck7.txt* ist die zehnte Perle mit einem A codiert.

Eingabedatei	Untere Schranke	Greedy		ILP / Polynomiell	Arithmetisch
		ohne PP	mit PP		
<i>schmuck0.txt</i>	112	113	113	113	112
<i>schmuck00.txt</i>	362	372	372	372	362
<i>schmuck01.txt</i>	1107	1150	1150	1150	1106
<i>schmuck1.txt</i>	187	192	191	191	188
<i>schmuck2.txt</i>	131	135	135	135	122
<i>schmuck3.txt</i>	252	285	285	279	252
<i>schmuck4.txt</i>	131	137	137	137	129
<i>schmuck5.txt</i>	3132	3308	3240	3162	3134
<i>schmuck6.txt</i>	227	236	234	234	225
<i>schmuck7.txt</i>	129249	134902	134902	134559	129572
<i>schmuck8.txt</i>	3231	3331	3300	3287	3230
<i>schmuck9.txt</i>	36387	37614	36936	36597	36387

Tabelle 2: Die Kettenlängen in Millimeter für die verschiedenen Algorithmen. Die Ergebnisse vom ILP Ansatz bzw. dem quasi-polynomiellen Ansatz sind identisch und **optimale Längen für präfixfreie Codes**, sie sollten entsprechend nicht unterschritten werden. Für einen eindeutig decodierbaren Code sollte die aus der Shannon Bound abgeleitete untere Schranke nicht signifikant unterschritten werden. Leichte Unterschreitungen können möglich sein, da die Schranke nur im Mittel gilt, wie beispielsweise bei den Ergebnissen aus der arithmetischen Codierung. Für das Greedy Verfahren sind die Werte mit und ohne Postprocessing (PP) angegeben.

Beipackzettel für schmuck0.txt.

S	211	R	2122	H	22222	M	2211	E	111	I	122
D	2121	␣	112	L	2221	C	22221	O	2212	N	121

Beipackzettel für schmuck00.txt.

n	22	G	3231	E	33113	d	321	A	3322	b	3333
c	233	m	3232	␣	11	f	33111	o	3313	?	3321
h	231	a	313	l	311	u	3222	i	13	s	232
k	3323	t	21	D	3221	w	3223	e	12	r	312
g	3233	Z	3312	P	3331	W	3332				

Beipackzettel für schmuck01.txt.

i	33	.	514	,	523	ä	553	V	543	E	522
z	545	s	34	N	5555	W	5554	l	42	O	544
S	542	r	32	g	511	D	524	K	5543	o	45
ü	532	v	541	t	41	d	515	G	5542	␣	1
w	531	b	525	R	5551	M	5553	c	44	B	5541
m	512	n	31	H	5544	a	35	I	533	ö	551
SS	5545	F	552	u	513	e	2	h	43	k	521
p	534	f	535	...	5552						

Beipackzettel für schmuck1.txt.

w	13	D	231	N	312	I	2221	m	323	f	213
h	232	e	11	F	311	ü	321	W	322	r	211
"	2213	B	2211	d	2223	␣	122	i	212	u	33
n	123	b	2222	o	223	t	121	s	2212	k	233
a	313										

Beipackzettel für schmuck2.txt.

d	21112	c	211112	e	2112	h	212	f	2111111	b	222
␣	221	g	2111112	a	1						

Beipackzettel für schmuck3.txt.

f 312 311 g 313 d 321 c 12 h 33
 e 322 b 2 a 11

Beipackzettel für schmuck4.txt.

h 11112 i 1121 g 211 k 1122 n 11111111 e 111112
 j 1112 l 212 f 11111112 c 1212 m 122 a 1211
 b 22 d 1111112

Beipackzettel für schmuck5.txt.

o 211 f 323 g 421 ; 622 l 24 v 44
 T 53 n 221 p 33 ' 55 3 11 y 41
 , 423 q 43 s 212 A 612 d 23 H 623
 S 63 r 222 5 611 2 73 h 321 t 13
 9 71 l 54 a 213 x 51 G 613 z 72
 j 621 i 14 m 31 u 322 e 12 c 223
 b 34 . 422 w 521 F 522 - 64

Beipackzettel für schmuck6.txt.

馬 1122 志 1222 時 133 學 12113 禹 212 去 223
 鄧 1132 督 123 武 311 , 1111 妻 132 不 2112
 英 2113 非 1123 希 313 無 12112 遇 1133 肯 312
 公 1213 。 1113 雄 23 晉 1131 郵 213 未 32
 大 33 文 1121 也 1212 齊 1221 有 1112 止 12111
 、 131 都 2111 望 221 古 222

Beipackzettel für schmuck7.txt.

r 51 O 876 o 65 w 72 W 775 _ 875
 x A71 “ 92 C A77 F 767 f 71 D 772
 J 866 s 52 5 973 G 862 R 861 h 55
 SS 771 Ä A1 8 A2 4 96 „ 93 p 766
 T 863 U 877 6 976 i 4 c 62 . 73
 : 874 1 95 B 78 P 94 Q A6 j 88

Auf der nächsten Seite fortgesetzt

Beipackzettel für schmuck7.txt. (Fortsetzung)

(872	q	98	l	61	S	761	v	765	d	54
b	66	K	777	└	1	z	75	k	74	Z	864
H	83	L	85	u	57	N	776)	873	!	974
2	975	7	A3	y	A8	g	63	n	3	Ö	A76
M	81	;	871	'	A5	a	53	ä	762	E	773
0	977	?	972	9	A4	A	82	j	865	V	774
-	867	m	64	[91	e	2	t	56	,	67
ü	763	ö	764	3	971	I	84				

Beipackzettel für schmuck8.txt.

王	2223	奮	534	異	42214	樓	253	清	2522	金	3322
出	2312	許	41112	雙	4241	怒	42122	郵	3314	鐘	41224
莫	3243	；	144	省	4242	嚴	21224	來	22214	里	31212
於	31124	自	2232	定	3232	目	2424	文	22122	分	41122
底	254	濟	2421	尤	2234	誠	4133	煙	4411	熟	3421
寫	41222	攬	433	律	2132	龍	32114	圍	4215	遙	42213
辦	3321	,	111	低	3123	遇	32122	丞	42111	況	41111
空	4313	而	2115	者	21223	似	4213	督	42223	柳	32222
子	3133	》	1211	言	2131	飯	3142	餘	4414	甲	3214
先	315	才	2324	意	21213	如	31111	押	32213	《	1213
在	15	韓	235	上	233	般	5113	生	515	相	133
暮	443	詩	134	調	2125	翰	4244	馬	2113	今	41211
何	22124	性	21123	成	41123	九	514	英	3215	非	2422
象	41223	愛	4423	拙	3424	篇	2241	外	2143	料	4214
榮	3134	深	2412	豈	4422	蝶	2514	骨	21124	經	4143
國	4321	若	3313	陰	32221	老	3323	至	425	四	2322
「	114	仙	531	問	2144	訟	325	有	1223	好	22221
中	2311	否	31112	不	1222	耶	5213	春	4233	城	3141
最	4224	葉	344	率	3324	望	41124	首	4125	萬	2513
草	42121	辛	415	托	31113	扈	5114	也	145	拳	335
時	124	租	524	皋	525	星	31122	希	5211	貧	4311
日	2114	鏡	3423	小	4142	妻	3422	寄	42113	開	22111
樹	3411	乎	2213	登	354	韻	32112	和	31121	趣	22121
腔	343	無	2413	席	5222	迴	55	可	533	志	2313
心	532	到	42211	冰	351	門	3224	口	3311	郎	32211

Auf der nächsten Seite fortgesetzt

Beipackzettel für schmuck8.txt. (Fortsetzung)

輕	445	都	2141	半	4132	憐	32214	午	31213	談	215
華	4424	」	113	水	2411	張	3312	李	22113	作	21212
通	22123	炊	41214	居	5214	多	22213	思	4114	二	3223
為	1224	」	2225	野	2521	大	2243	三	22223	看	22212
須	4412	囧	41113	公	2111	婦	3114	盼	3132	勞	435
客	31223	然	3124	恩	2511	化	5112	已	22114	陽	3225
蹕	31211	世	42112	同	2233	毆	31221	後	22224	頌	4225
招	42221	是	143	楊	523	寧	444	架	5122	卓	2523
皇	4134	則	4421	曲	5221	枝	22112	仗	2524	及	4243
箏	31224	能	255	猶	4324	山	41212	桐	3242	誰	21122
年	225	專	352	事	453	肯	4141	家	42114	略	3144
指	3213	其	131	雅	3143	吟	2224	近	4131	學	3413
霄	4124	此	244	人	132	酬	4314	逋	32223	將	2321
騎	4113	火	4323	未	2134	一	125	雄	31222	:	1212
鄧	41213	渾	42222	踏	32121	見	2215	蘄	3125	云	123
君	243	侯	5123	以	2124	叟	3231	字	513	情	21222
懷	4115	謂	3113	齊	2512	己	32124	。	112	知	21121
歸	31123	歌	4322	西	5224	範	451	風	141	便	2242
格	1221	絕	334	臨	5111	元	32212	增	42212	繩	2314
蝴	41114	斜	4232	之	1214	十	21214	得	2244	入	345
曰	2414	命	31114	窺	5223	〈	4413	兩	3234	祠	4312
士	4234	詠	54	封	41121	奇	32113	禹	2214	光	2323
靈	42124	古	3233	句	4223	訊	333	種	434	丑	4144
氣	353	林	2231	皆	21211	天	2142	齋	42123	玩	3414
浪	32224	百	2123	秀	5124	去	31214	卒	3131	侮	42224
詞	452	晉	245	與	234	描	4123	七	2423	易	3115
屈	4231	滿	22222	武	142	、	21221	妙	3412	止	41221
花	2133	散	3241	哉	3244	鄂	5212	色	32111	絲	32123
解	22211	寶	454	從	5121						

Beipackzettel für schmuck9.txt.

牧	221213	帽	21221112	驅	111221112	外	21134
一	43	旅	4121112	問	13121112	護	22121112
帳	112121112	匕	1112124	末	31121112	易	121121112
輝	211121112	褒	1111121112	殘	122111112	六	212111112

Auf der nächsten Seite fortgesetzt

Beipackzettel für schmuck9.txt. (Fortsetzung)

堂	21132	人	1224	然	1112111112	ル	11213
撃	1121213	儀	41124	東	41111112	所	131124
式	111134	優	131111112	岩	221111112	指	311213
ぬ	221124	の	14	裂	1121111112	夕	4122
口	311111112	杵	1211111112	紀	2111111112	頑	1111111112
厳	1121124	尊	311124	隻	1211124	線	212212
キ	2111124	廊	1211213	ぜ	41111121	昇	131111121
縦	221111121	示	11111213	荷	2111213	改	1121111121
フ	211113	輪	11111124	冷	311111121	莫	1211111121
体	2111111121	疑	11111111121	好	211122111	応	1111122111
面	211314	鉄	12224	区	1111314	老	122214
詰	122112111	眺	212112111	団	212214	そ	313
緒	1112112111	真	41112111	を	33	妻	131112111
運	21224	る	1122	観	221112111	敷	111224
雲	1121112111	登	311112111	隊	1112214	角	1211112111
達	41214	託	2111112111	チ	13122	篠	11111112111
塔	4124	聞	1112121111	越	1221113	放	131214
裕	41121111	市	1112212	識	131121111	乗	21223
描	221121111	根	221214	差	13124	車	111132
委	1121214	得	22124	忘	1121121111	説	2121113
郭	311121111	年	112124	跡	1211121111	ク	31114
利	41212	巻	2111121111	世	311214	様	1211214
者	111213	州	11111121111	終	211311111	四	31124
ら	34	抱	1111311111	防	121124	侮	122211111
百	212211111	謙	1112211111	だ	134	ね	111223
碎	41211111	類	131211111	値	221211111	逃	1121211111
づ	11121113	図	2111214	民	411113	信	11111214
。	132	復	1311113	持	1221114	停	311211111
表	1211211111	導	2111211111	医	211124	寧	11111211111
板	1221111111	保	2211113	事	22122	こ	312
負	2121114	渡	2121111111	構	11121111111	像	411111111
合	1211113	あ	1124	秀	1311111111	七	2211111111
っ	11112	ン	1213	目	131212	沈	11211111111
方	1111124	地	121114	理	3111111111	足	11211113
か	2112	心	11121114	テ	411114	ふ	12111111111
有	1311114	性	21111111111	や	3114	匂	2211114

Auf der nächsten Seite fortgesetzt

Beipackzettel für schmuck9.txt. (Fortsetzung)

扉	1211224	演	2111224	よ	4123	摘	11111224
れ	214	部	13123	議	1221124	袖	2121124
沛	11121124	ラ	111113	肩	411124	虚	1311124
軽	2211124	午	11211114	開	11211124	た	23
も	1214	史	3111124	束	12111124	載	21111124
低	111111124	走	3111113	峡	11121214	ろ	22123
敬	411214	漏	1311214	名	12111113	進	3111114
革	2211214	ム	12111114	呼	21111114	被	11211214
造	3111214	何	111111114	へ	12111214	ん	3113
紹	122114	不	221212	ゼ	21111214	日	1211223
介	212114	頭	2111223	元	11111223	揖	111111214
ほ	112123	貶	2113114	拝	11113114	間	1222114
！	2122114	特	21111113	組	11122114	賀	412114
聳	1312114	二	1221123	代	111111113	門	2212114
多	2121123	細	11121123	丈	411123	頷	1311123
遙	11212114	め	4113	正	3112114	貌	12112114
償	21112114	決	2111113	余	111112114	口	211114
若	1112114	後	1121212	挨	12211114	経	21211114
共	111211114	デ	4111114	背	13111114	手	112122
ぶ	11111113	ャ	22111114	厶	11114	堤	1211222
逆	112111114	鄙	31111114	側	41114	追	121111114
回	211111114	同	1111111114	雰	21112213	重	311212
塩	111112213	広	131114	服	12211213	意	221114
大	111114	今	2211123	要	21211213	国	111211213
答	4111213	ィ	13111213	サ	12222	素	1121114
撥	22111213	眉	112111213	リ	311114	じ	2111222
消	31111213	・	1223	行	1211212	飛	121111213
ば	21222	助	11111222	員	1112122	御	211111213
攻	1111111213	測	111212113	ぎ	2111212	料	4112113
明	11111212	積	11211123	ォ	3111123	筋	13112113
祭	41122	送	1221112	秘	22112113	労	112112113
短	31112113	ガ	1211114	湖	121112113	幅	211112113
以	12111123	よ	224	当	1111112113	町	1221122
上	13113	暗	111111123	工	21111123	珍	21131113
集	11121213	き	223	紗	111131113	蒸	12221113
歌	21221113	水	31123	念	111221113	言	31122

Auf der nächsten Seite fortgesetzt

Beipackzettel für schmuck9.txt. (Fortsetzung)

岸	411213	戸	4121113	機	13121113	着	2111114
券	1311213	計	22121113	蔑	112121113	深	2211213
砂	11211213	北	2121122	境	3111213	無	11121122
姿	31121113	礼	121121113	薄	211121113	ト	2124
船	411122	治	12111213	グ	21111213	捨	1111121113
に	113	海	12214	ヴ	111111213	苦	122111113
支	2113113	流	11113113	誰	212111113	万	1112111113
充	41111113	び	121123	ひ	11111114	天	131111113
移	221111113	建	111222	席	1121111113	む	21214
界	311111113	完	1211111113	プ	2111111113	墓	11111111113
主	211123	囲	41111122	取	2121112	換	131111122
時	1222113	ミ	1112123	受	221111122	務	1121111122
」	413	歴	311111122	毛	1211111122	覆	2122113
暴	2111111122	し	24	声	11111111122	さ	314
閉	211122112	管	1111122112	気	131122	雄	122112112
台	212112112	げ	1111114	一	21133	駅	11121112
節	1112112112	嫌	41112112	ケ	131112112	必	221112112
実	41123	量	1311122	ア	112113	抑	1121112112
「	1313	別	311112112	案	1211112112	見	11124
押	2111112112	ウ	2214	交	131123	屋	221123
エ	22113	ド	221122	晴	2211122	少	11211122
エ	11111112112	払	1112121112	突	41121112	ワ	1121123
南	3111122	ゃ	12111122	再	131121112	築	1111123
定	221121112	ご	411112	使	1121121112	商	21111122
関	311121112	傘	1211121112	マ	122113	付	2111121112
浮	11111121112	波	11122113	泊	211311112	三	412113
男	1111311112	五	122211112	と	114	的	212211112
潮	1112211112	迎	41211112	養	131211112	想	221211112
脇	1121211112	買	1312113	幕	311211112	レ	11214
恭	1211211112	ホ	2212113	届	2111211112	昔	11111211112
社	11212113	戦	1221111112	港	1311112	司	212113
丁	2121111112	直	3112113	ダ	111111122	動	12112113
い	32	思	31113	ゆ	21112113	切	11121111112
高	4111111112	扱	1311111112	山	111112113	十	12211113
楽	21211113	階	2211111112	円	11211111112	な	42
小	11121212	中	111214	抜	3111111112	出	1111113

Auf der nächsten Seite fortgesetzt

Beipackzettel für schmuck9.txt. (Fortsetzung)

織	12111111112	家	1121122	煉	21111111112	対	111211113
セ	111111111112	全	4111113	ふ	1111111111111	到	411212
ぞ	1311212	格	2211212	ぼ	311123	激	21112214
け	2213	注	13111113	お	11123	知	22111113
ギ	111112214	前	2211112	オ	1112113	調	12211214
独	21211214	ァ	311122	威	111211214	せ	121113
拭	4111214	内	11211112	修	11211212	夫	13111214
陸	1211123	験	22111214	ど	12114	弁	112111214
ズ	31111214	売	121111214	往	211111214	ポ	112111113
断	1111111214	す	1114	味	3111212	霧	111212114
架	4112114	套	31111113	ぐ	2111123	用	11111123
参	121111113	巨	12111212	く	44	生	21111212
雨	41113	单	13112114	稻	22112114	連	112112114
族	31112114	態	121112114	域	211112114	風	211111113
嘆	1111112114	ち	12113	ず	111111212	話	2113112
費	21131114	度	1111111113	ス	12212	襟	111131114
喜	12221114	形	21221114	ベ	21112212	音	11113112
任	131113	能	111112212	つ	21212	浸	111221114
千	4121114	下	12211212	去	13121114	イ	414
探	22121114	馬	1211122	客	112121114	う	124
縮	31121114	ペ	121121114	パ	1222112	子	211121114
奏	3111112	通	221113	徴	1111121114	職	21211212
場	2111122	属	122111114	拶	212111114	相	111211114
み	4114	諺	41111114	身	211313	シ	111211212
返	131111114	己	221111114	が	222	英	1121111114
一	2123	誇	311111114	《	1211111114	道	13114
窓	4111212	等	13111212	制	2111111114	沢	11111111114
彼	12213	役	22111212	割	41111123	安	1111313
比	131111123	眠	2122112	由	112111212	毎	221111123
感	31111212	抗	1121111123	粗	311111123	古	11122112
伝	1211111123	折	2111111123	聖	21213	散	412112
長	11111111123	院	211122113	川	11111122	莊	1111122113
敵	122112113	抵	212112113	他	1312112	仕	2212112
物	22114	》	1112112113	ま	1123	吟	41112113
床	121111212	ッ	12111112	作	131112113	自	211312
入	1121113	瓦	221112113	歩	11212112	伸	211111212

Auf der nächsten Seite fortgesetzt

Beipackzettel für schmuck9.txt. (Fortsetzung)

変	1121112113	快	311112113	教	122213	分	1111312
幾	1211112113	醸	2111112113	望	1111111212	便	111212112
耐	11111112113	河	3112112	ブ	4112112	藁	1112121113
石	12112112	汽	13112112	首	41121113	ヨ	112114
え	21113	半	131121113	は	213	亀	221121113
カ	1314	設	1121121113	羊	22112112	配	311121113
数	212213	艦	1211121113	、	1113	寄	112112112
神	31112112	草	21112112	待	2111121113	ジ	21114
権	121112112	り	133	友	21111112	刻	11111121113
ゴ	211311113	べ	111133	路	1111311113	々	111112112
空	12211112	考	21211112	で	1212	領	122211113
崇	212211113	簡	1112211113	濡	211112112	ざ	111111112
約	41211113	産	131211113	埋	221211113	拘	1121211113
舞	111211112	美	1111112112	横	21131112	一	122212
申	311211113	ノ	1112213	胆	1211211113	印	2111211113
都	111131112	過	41213	彩	11111211113	わ	2114
立	311113	善	1221111113	幹	2121111113	会	12223
惜	11121111113	選	4111111113	降	1211221	泥	13111111113
ザ	2211111113	緑	11211111113	街	12221112	急	31111111113
ニ	12111111113	認	21111111113	振	131213	業	111111111113
て	123	尋	111111111112				

1.7 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

- (1) *Problem adäquat modelliert:* Die Modellierung muss es ermöglichen, einen beliebigen präfixfreien Code unter Verwendung aller Perlen darzustellen. Die Darstellung muss garantieren, dass:

- der Code präfixfrei ist,
- jedes Eingabesymbol ein Codewort erhält und
- alle Perlen genutzt werden können.

Mit der in der Eingabe vorhandenen Zeichen muss korrekt umgegangen werden. Fehler führen ggf. zu deutlich anderen Ergebnissen, weshalb hier 4 Punkte dafür abgezogen werden – aber nicht bei 1.6 für allein daraus folgende schlechte Ergebnisse.

- (2) *Verfahren nicht unnötig ineffizient:* Der Algorithmus soll strukturiert und ohne unnötige Wiederholungen arbeiten. Beispielsweise darf die gleiche Codierung nicht mehrfach überprüft werden. Suchstrategien müssen zielgerichtet und effizient sein.
- (3) *Laufzeit des Verfahrens in Ordnung:* Wenn das Verfahren wegen zu hoher Laufzeit Ergebnisse ...
- nur für Teilaufgabe a), also `schmuck0.txt`, `schmuck00.txt` und `schmuck01.txt` berechnen kann, sind 6 Punkte abzuziehen.
 - für mindestens ein Beispiel aus `schmuck1.txt` bis `schmuck8.txt` nicht berechnen kann, sind 4 Punkte abzuziehen.
 - einzig für `schmuck9.txt` nicht berechnen kann, sind 3 Punkte abzuziehen.

Falls mehrere Verfahren implementiert wurden, wird das Hauptverfahren gewertet.

- (4) *Speicherbedarf in Ordnung:* Der Speicherbedarf sollte nicht so groß sein, dass die Beispielergebnisse bzw. Teilaufgabe b) deswegen nicht bearbeitet werden können.
- (5) *Verfahren mit korrekten Ergebnissen:* Die Mindestkettenlänge darf die *untere Schranke* nur um wenige mm unterschreiten.
- (6) *Verfahren mit guten Ergebnissen:* Hier werden keine Punkte für fehlende Ergebnisse abgezogen.
- Werden die Beispiele der Teilaufgabe a) (`schmuck0.txt`, `schmuck00.txt` und `schmuck01.txt`) im Bereich *okay* oder schlechter gelöst, wird 1 Punkt abgezogen. Für Ergebnisse im Bereich *gut* werden keine Punkte abgezogen.
 - Liegen die Ergebnisse für die Beispiele `schmuck1.txt` bis `schmuck8.txt` im Bereich *okay*, wird 1 Punkt abgezogen. Bei größeren Abweichungen werden 3 Punkte abgezogen. Liegen die Ergebnisse mindestens im Bereich *gut*, werden keine Punkte abgezogen. Ein Ausreißer wird toleriert.
 - Für das Beispiel `schmuck9.txt` gibt es keine Abzüge. Liegt das Ergebnis im Bereich *okay*, wird 1 Pluspunkt vergeben, für eine Ergebnis im Bereich *gut* 2 Pluspunkte.

Wenn die optimalen Ergebnisse nur durch einen ILP-Solver erreicht wurden, werden keine Pluspunkte vergeben.

Dateiname	Untere Schranke	Optimum	gut	okay
<code>schmuck0.txt</code>	112	113	115	119
<code>schmuck00.txt</code>	362	372	379	391
<code>schmuck01.txt</code>	1107	1150	1173	1208
<code>schmuck1.txt</code>	187	191	194	201
<code>schmuck2.txt</code>	131	135	137	145
<code>schmuck3.txt</code>	252	279	285	293
<code>schmuck4.txt</code>	131	137	139	144
<code>schmuck5.txt</code>	3132	3162	3240	3321
<code>schmuck6.txt</code>	227	234	238	246
<code>schmuck7.txt</code>	129249	134559	137250	141287
<code>schmuck8.txt</code>	3231	3287	3352	3452
<code>schmuck9.txt</code>	36387	36597	37328	40000

- (7) *Teilaufgabe b) bearbeitet:* Wurde die Teilaufgabe a) nicht bearbeitet, werden 10 Punkte abgezogen.

2. Theoretische Analyse

- (1) *Verfahren / Qualität insgesamt gut begründet:* Es muss klar dargestellt werden, ob das Verfahren optimale oder lediglich gute Ergebnisse liefert. Qualität und Korrektheit der Ergebnisse müssen argumentativ begründet sein. Wenn die Ergebnisse mehrerer Verfahren kommentarlos angegeben werden, können 2 Punkte abgezogen werden. Für eine Einsendung, die mehrere Verfahren vergleicht und Begründungen für deren Qualität liefert, können Bonuspunkte vergeben werden.
- (2) *ILP-Ansatz ausreichend erläutert:* Wird der ILP-Ansatz von Karp verwendet (zum Beispiel über das zitierte Paper), reicht ein Verweis nicht aus. Herleitung und Funktionsweise des Ansatzes müssen verständlich und eigenständig erläutert werden; es soll klar werden, dass der Ansatz verstanden wurde.
- (3) *Gute Überlegungen zur Laufzeit des Verfahrens:* Die Laufzeit des Verfahrens muss nicht zwingend formal, aber nachvollziehbar und korrekt charakterisiert werden.

3. Dokumentation

- (3) *Vorgegebene Beispiele dokumentiert:* Es müssen alle 12 Beispiele dokumentiert werden, also `schmuck0.txt`, `schmuck00.txt`, `schmuck01.txt` und `schmuck1.txt` bis `schmuck9.txt`. Es gibt keinen Abzug, wenn nur das Beispiel aus der Aufgabenstellung (`schmuck0.txt`) fehlt.
- (4) *Ergebnisse nachvollziehbar dargestellt:* Für jede Beispieleingabe muss
 - die finale Codelänge angegeben werden und
 - der vollständige Beipackzettel bereitgestellt werden.

Wird die finale Codelänge nicht angegeben, aber der Code mitgeliefert, können bis zu 2 Punkte abgezogen werden. Für sehr lange Dateien (zum Beispiel `schmuck8.txt` und `schmuck9.txt`) darf der Beipackzettel (mit einem Verweis) in externe, menschenlesbare Textdateien ausgelagert werden.

Aufgabe 2: Simultane Labyrinth

Bei dieser Aufgabe sind zwei $n \times m$ Labyrinth gegeben und gesucht ist eine möglichst Kurze Liste an Anweisungen, mit der beide Spieler vom Startpunkt $(0,0)$ das Ziel $(n-1, m-1)$ erreichen. Eine Anweisung besteht aus einem Pfeil, der die Bewegungsrichtung angibt. Die möglichen Bewegungsrichtungen sind \uparrow (Norden), \rightarrow (Osten), \downarrow (Süden) und \leftarrow (Westen). Wenn sich ein Spieler an der Position (x,y) befindet, ändert sich seine Position nach einer Anweisung wie folgt:

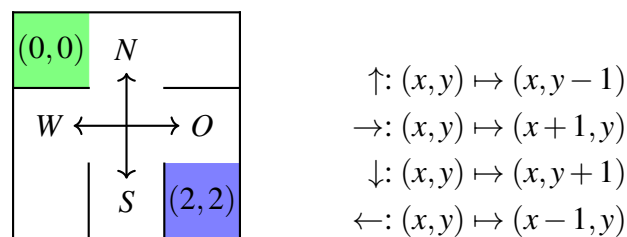


Abbildung 2.1: Notation des Labyrinths

Der Start $(0,0)$ wird in den folgenden Erklärungen grün und das Ziel $(n-1, m-1)$ blau markiert.

Sollte ein Spieler eine Anweisung aufgrund einer Wand nicht ausführen können oder bereits im Ziel sein, ignoriert er diese Anweisung. Wenn eine Bewegung auf eine Grube führt, wird der Spieler auf die Startposition zurückgesetzt. Um nun einen Lösungszustand vollständig zu beschreiben, wird nur die Position der beiden Spieler benötigt. Wir schreiben die Position als Tupel zweier Positionen: $((x_1, y_1), (x_2, y_2))$.

2.1 Breitensuche

Wir betrachten zunächst den Ansatz einer Breitensuche auf einem einzelnen Labyrinth und erweitern diesen dann auf zwei Labyrinth.

Bei einer Breitensuche über alle möglichen Anweisungen werden mithilfe von Backtracking alle möglichen Listen von Anweisungen in aufsteigender Länge generiert und überprüft, ob die Zielposition erreicht wird. Sobald die erste Lösung gefunden wurde, hat sie somit auch die kürzeste Länge. Der Aufwand hierfür ist jedoch sehr hoch, da die Anzahl der möglichen Anweisungen exponentiell mit der Länge der Liste wächst.

Ein einzelnes $n \times m$ Labyrinth hat nm mögliche Positionen, die ein Spieler einnehmen kann. Somit bräuchte der längst mögliche Weg $nm - 1$ Anweisungen. Bei 4 Anweisungen wäre das bereits ein Suchraum von 4^{nm-1} möglichen Lösungen. Dieser Ansatz allein ist somit nicht ausreichend, um ein größeres Labyrinth zu lösen.

Anstatt alle möglichen Listen von Anweisungen zu untersuchen ist es sinnvoll, nur die Positionen zu betrachten, die durch eine Anweisung erreicht werden können. Wir erstellen hierfür eine Warteschlange (Queue) von Positionen, die noch nicht untersucht wurden. Wir beginnen mit der Startposition und fügen alle möglichen Positionen, die durch eine Anweisung erreicht werden können, in die Warteschlange ein. Nun wird die nächste Position aus der Warteschlange genommen und wieder alle möglichen Positionen, die durch eine Anweisung erreicht werden können, in die Warteschlange eingefügt. Hierbei ist es wichtig, dass bereits besuchte Positionen nicht erneut besucht werden. Sobald das Ziel erreicht wird, wurde eine Lösung gefunden.

In der folgenden Abbildung wird die Breitensuche auf einem Labyrinth dargestellt. Die grauen Felder sind bereits besuchte Positionen und die orangenen Felder sind Positionen, die noch in der Warteschlange sind.

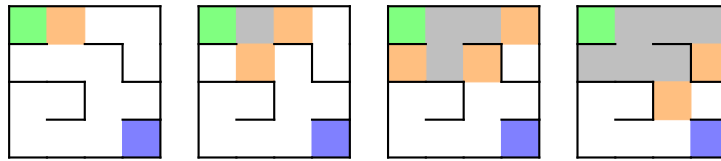


Abbildung 2.2: Breitensuche auf einem Labyrinth

Um die Liste der Anweisungen die zur Lösung geführt haben zu erhalten, wird für jede Position gespeichert, von welcher Position sie erreicht wurde.

Wenn die Zielposition in der Breitensuche erreicht wurde, kann die Liste der Anweisungen durch die gespeicherten Vorgänger rekonstruiert werden. Hierfür werden, beginnend bei der Zielposition, die Vorgänger zurückverfolgt, bis die Startposition erreicht wurde.

Um nun zwei Labyrinth zu Lösen wird zunächst die Position $((0,0), (0,0))$ in die Warteschlange eingefügt. Von einer Position $((x_1, y_1), (x_2, y_2))$ müssen nun erneut alle 4 möglichen Anweisungen durchprobiert werden. Wie zuvor beschrieben gilt, dass ein Spieler eine Anweisung ignoriert, wenn er sie wegen einer Wand nicht ausführen kann oder bereits im Ziel ist.

Sollte so die Zielposition für beide Spieler $((n-1, m-1), (n-1, m-1))$ erreicht werden, wurde eine Lösung gefunden und der Lösungsweg kann durch die Vorgänger rekonstruiert werden.

2.2 Aufwand Breitensuche

Eine Breitensuche auf einem einzelnen Labyrinth hat eine Laufzeit von $O(nm)$, da jede Position maximal einmal besucht wird. Bei zwei Labyrinth gibt es jedoch $(nm)^2$ mögliche Kombinationen von Positionen, die ein Spieler einnehmen kann. Somit ist der Aufwand für die Breitensuche bei zwei Labyrinth $O((nm)^2)$, was für die größeren Labyrinth in den Beispieldateien nicht mehr praktikabel ist.

2.3 A-Star Algorithmus

Die Breitensuche ist nicht effizient genug, um größere Labyrinth zu lösen. Eine Möglichkeit den Algorithmus zu verbessern ist die Verwendung einer Distanzheuristik. Hierfür wird die Distanz zwischen der aktuellen Position und dem Ziel abgeschätzt um die Suche in die richtige Richtung zu lenken.

Der A-Star Algorithmus verwendet genau diese Idee und benötigt eine untere Grenze für die Distanz zwischen der aktuellen Position und dem Ziel. Für das Problem mit zwei Labyrinth ist es sinnvoll die Distanz zwischen beiden Spielern und dem Ziel zu betrachten. Benötigt Spieler 1 beispielsweise noch 5 Schritte um das Ziel (allein) zu erreichen und Spieler 2 noch 3 Schritte, dann ist klar das mindestens $\max(5, 3) = 5$ Schritte benötigt werden, um beide Spieler zum Ziel zu bringen.

Um diese Distanz im Algorithmus effizient zu verwenden, wird für jede Position in einem Labyrinth die Distanz zum Ziel berechnet und in einer Tabelle gespeichert. Die kann mithilfe einer Breitensuche beginnend vom Ziel berechnet werden.

In dem folgenden Beispiel wird die Distanz für die Beiden Labyrinth aus der Aufgabenstellung dargestellt.

8	3	2	6	3	2
7	4	1	5	4	1
6	5	0	6	1	0

Abbildung 2.3: Beispiel für die Berechnung der Distanz zum Ziel (blau)

Hierbei ist zu beachten, dass die Distanz auf einer Grube (rot) nicht wohldefiniert ist, da der Spieler nicht auf dieser Position stehen kann. Im Algorithmus wird daher nie die Distanz auf einer Grube gelesen und stattdessen die Distanz auf der Startposition.

Mit der Distanzheuristik kann nun der A-Star Algorithmus verwendet werden, um eine Lösung zu finden. Die Warteschlange aus der Breitensuche wird durch eine Prioritätswarteschlange ersetzt, die die Positionen nach den abgeschätzten Kosten zum Ziel sortiert. Die abgeschätzten Kosten an einer Position $((x_1, y_1), (x_2, y_2))$ sind die Summe aus der Anzahl der Anweisungen bis zu dieser Position und der dem Maximum der Distanz beider Spieler zum Ziel.

Algorithmus 4 A-Star

$D_1 = \text{berechneDist}(L_1)$

▷ Distanztabelle von Labyrinth 1

$D_2 = \text{berechneDist}(L_2)$

▷ Distanztabelle von Labyrinth 2

procedure $H(p_1, p_2)$

▷ Distanzheuristik

return $\max(D_1[p_1], D_2[p_2])$

$S = ((0, 0), (0, 0))$

▷ Startposition

$Z = ((n - 1, m - 1), (n - 1, m - 1))$

▷ Zielposition

$N = [(S, S, 0, H(S))]$

▷ Prioritätswarteschlange

while N nicht leer **do**

$(s, p, c, _) = N.\text{pop}()$

▷ Entferne Element mit niedrigsten erwarteten Kosten

if s bereits besucht **then**

continue

 markiere s als besucht

 markiere p als Vorgänger von s

if $s = Z$ **then**

break

for r in $\{N, O, S, W\}$ **do**

$l = r(s)$

▷ Neue Positionen l nach Anweisung r

if l noch nicht besucht **then**

 füge $(l, s, c + 1, c + 1 + H(l))$ in N ein

2.4 Implementierungsdetails

Die korrekte Wahl der Datenstruktur für die Warteschlange und die Vorgänger ist entscheidend für die Effizienz des Algorithmus. Für die Warteschlange bietet sich ein Binary Heap an. (Sie-

he Wikipedia) Diese Datenstruktur ermöglicht es, das Element mit den niedrigsten Kosten in $O(\log n)$ zu entfernen und ein neues Element in $O(\log n)$ einzufügen. Würde eine unsortierte Liste verwendet werden, so würde das Finden des Elements mit den niedrigsten Kosten $O(n)$ dauern. Wird beim Einfügen neuer Elemente versucht die Liste sortiert zu halten, würde das Einfügen $O(n)$ dauern.

Die Vorgänger können als Dictionary (Map) gespeichert werden, wobei die Position als Schlüssel und der Vorgänger als Wert gespeichert werden. Dies hat den Vorteil, dass nur die tatsächlich besuchten Positionen gespeichert werden. Ein Array würde hingegen alle Positionen speichern, was bei dem größten Labyrinth (Beispiel 6) $250^4 \cdot 2\text{byte} \approx 7.8\text{GB}$ Speicher benötigen würde.

2.5 Laufzeit A-Star

Die Laufzeit der Distanzberechnung ist $O(nm)$, da jede Position maximal einmal besucht wird. Da der A-Star Algorithmus im worst-case alle Positionen besucht, ist die Laufzeit proportional zur Anzahl möglicher Positionen. Zusammen mit dem Einfügen und Entfernen von Elementen aus der Prioritätswarteschlange ist die Laufzeit des A-Star Algorithmus $O((nm)^2 \log((nm)^2))$. Die worst-case Laufzeit ist somit schlechter als die der Breitensuche, jedoch ist der A-Star Algorithmus aufgrund der gerichteten Suche in der Praxis deutlich schneller.

2.6 Anzahl der Lösungen

Eine interessante Frage ist, wie viele optimale Lösungen es für ein gegebenes Labyrinth gibt. Wenn statt nur einem Vorgänger alle Vorgänger mit den gleichen Kosten gespeichert werden, können alle optimalen Lösungen rekonstruiert werden. Die Anzahl der optimalen Lösungen kann nun dynamisch bestimmt werden. Hierfür wird die Anzahl der Lösungen für jede Position gespeichert, beginnend mit einer 1 bei der Zielposition. Beginnend mit der Zielposition wird nun eine Breitensuche über die Vorgänger durchgeführt und die Anzahl der Lösungen für alle Vorgänger aufsummiert.

Algorithmus 5 LösungZählen

$S = ((0, 0), (0, 0))$	▷ Startposition
$Z = ((n - 1, m - 1), (n - 1, m - 1))$	▷ Zielposition
$V = \text{Vorgängermap}$	▷ Aus der A-Star-oder Breitensuche
$L = \text{Anzahlmap}$	▷ Anzahl der Lösungen für jede Position
$L[Z] = 1$	▷ Zielposition hat eine Lösung
$N = [Z]$	▷ Warteschlange
while N nicht leer do	
$s = N.\text{pop}()$	▷ Entferne Element mit niedrigsten erwarteten Kosten
if s bereits besucht then	
continue	
markiere s als besucht	
for p in $V[s]$ do	
$L[p] += L[s]$	
füge p in N ein	
return $L[S]$	

Das folgende Beispiel zeigt einen möglichen gerichteten Graphen und die Anzahl der möglichen Wege vom Ziel (Blau) zu jedem Knoten. Jede gerichtete Kante zeigt hier auf die Vorgängerposition.

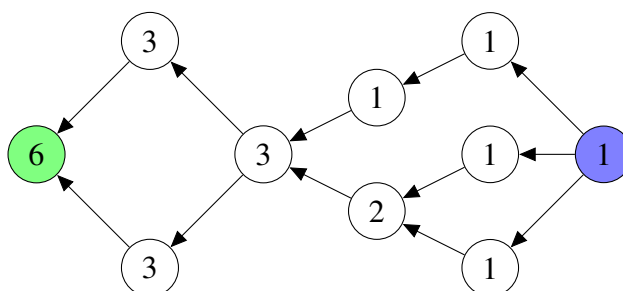


Abbildung 2.4: Anzahl der Pfade von der Zielposition zu jeder anderen Position

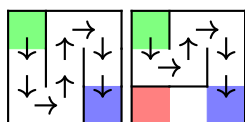
2.7 Beispiele

In der folgenden Tabelle sind die Laufzeiten und Anzahl der Anweisungen des A-star Algorithmus und der Breitensuche abzulesen.

Beispiel	Breite	Höhe	# Anweisungen	A-Star	Breitensuche
0	3	3	8	< 1 ms	< 1 ms
1	5	4	31	< 1 ms	< 1 ms
2	10	10	65	< 1 ms	1 ms
3	10	50	164	10 ms	18 ms
4	101	101	14384	5 s	8 s
5	151	151	1308	19 s	45 s
6	250	250	1844	188 ms	3 s
7	30	10	-	10 ms	4 ms
8	150	150	472	9 s	45 s
9	170	170	1012	9 s	94 s

In den folgenden Abschnitten wird für alle Beispiele die Anzahl der Lösungen und eine mögliche Anweisungsliste angegeben.

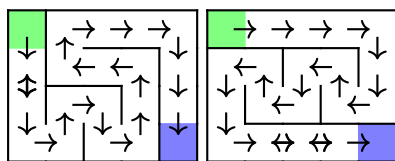
Beispiel 0



Anzahl optimaler Lösungen: 2

Anzahl Anweisungen: 8

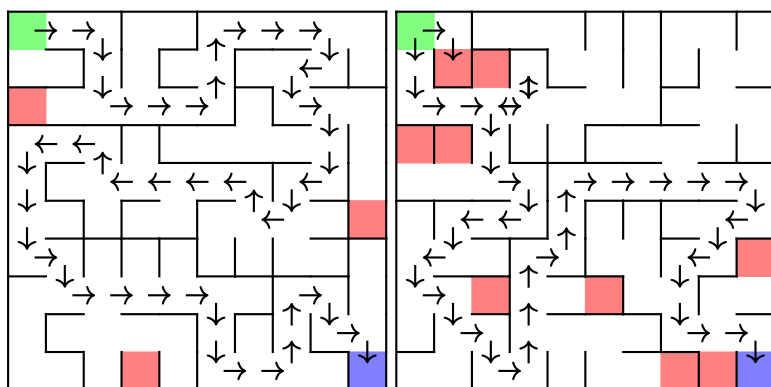
↓↓ → ↑↑ → ↓↓

Beispiel 1

Anzahl optimaler Lösungen: 1

Anzahl Anweisungen: 31

→ → → → ↓ ↓ ← ↑ ← ↓ ← ↑ ← ↓ ↓ → ↑ → ↓ → ↑ ↑ ← ← ↑ → → → ↓ ↓ ↓

Beispiel 2

Bei dieser Lösung wird die Grube direkt beim Start des rechten Labyrinths verwendet, um mit einer Süd-Anweisung zurück zum Startpunkt zu gelangen.

Anzahl optimaler Lösungen: 2

Anzahl Anweisungen: 65

→ → ↓ ↓ ↓ → → → ↑ ↑ → → → ↓ ← ↓ ↓ → ↓ ↓ ← ↓ ← ↑ ← ← ← ← ↑ ← ← ↓ ↓ ← ↓ → ↓
 → → → → ↓ ↓ → → ↑ ↑ ↑ → ↑ ↑ → → → → ↓ ← ↓ ← ↓ ↓ → → ↓

2.8 Beispiel 3

Anzahl optimaler Lösungen: 216

Anzahl Anweisungen: 164

↓ ↓ ↓ ↓ → ↓ ↓ → ↑ → → ↓ → → → ↓ ↓ ↓ ↓ → ↓ ← ↓ ← ↓ ↓ ↓ ↓ ↓ ← ← ↓ ← ↓ ← ↓ ↓ ↓ ↓ →
 → ↓ ↓ → → ↓ ↓ → ← ↓ ↓ ↓ ← ↓ → ↓ ← ← ↓ ← ↑ ← ↓ ↓ → ↓ ↓ → → → ↓ ← ↓ ↓ → → → →
 ↓ ↓ → → ↓ → → ↓ → ↓ ↓ ← ← ↓ ↓ ← ↓ ← ↓ ↓ → ↑ → ↓ ↓ ← ↓ ← ↓ ← ↑ ← ↑ ← ↑ ← ↓ ← ↓
 → → ↓ ↓ ↓ → → ↑ → ↓ ↓ ↓ → ↑ ← ↓ ← ← ← ↑ → → ↑ ← ↑ ↓ → ↓ ↓ ↓ ← ↓ ↓ ↓ → ↑ → ↓ →
 ↓ → ↓ → → → → →

Beispiel 4

Aufgrund der Größe des Beispiels werden nur die ersten und letzten 200 Anweisungen angezeigt. Die volle Ausgabe findest du im Materialteil auf der Webseite.

Anzahl optimaler Lösungen $\approx 8.7 \cdot 10^{408}$

Anzahl Anweisungen: 14384

→ → → ↓ ↓ ← ↑ ← ← ↓ → ↓ ← ↓ → ↓ ← ↓ → → ↓ → ↑ ↑ ← ↑ ↑ → ↓ → ↑ → ↓ ← ↓ ← ↓ ↓
 ↓ ← ← ← ↑ ← ↓ ↓ ↓ → ↑ → → → ↓ ↓ ↓ → ↑ ↑ → → ↑ ← ← ↑ → → → → ↑ ↑ → ↓ ← ↑ ↑ ↑
 ↑ ↑ ← ↓ ← ← ↓ → ↓ ← ← ↓ → → → → ↓ ← ← ← ↑ → ↑ ↑ ↑ → → ↑ ← ← ↓ ← ↑ ↑ → →
 ↓ ↓ → → ↓ ↓ → ↑ → → → → ↓ → → → ↓ ← ↑ ← ↑ ↑ ← ← ← ↓ ↓ ← ↓ → → ↓ → ↑ ↑ ←
 ← ↑ → ↓ → → ↓ → ↑ → ↓ ↓ ← ← ↓ ← ↓ ← ↓ → → → ↑ ← ← ↑ ↑ ↑ → ↓ ↓ → ← ↓ ↓ ↓ → ↑
 → → ↑ ← ← ← ↓ ← ↑ → ↑ →

...

→ → ↓ ↓ → ↑ → → ↑ ← ← ← ↑ → → → ↑ ↑ → ↓ ↓ → ↓ ← ↓ → → ↑ ↑ → → ↓ ← ← ← ←
 ← ← ↓ → ↓ ↓ → ↓ → ↑ ↑ ↑ ↑ → ↑ ← ← ↑ → → → → ↓ ← ↓ → ↓ → ↑ ↑ ↓ ← ↑ ← ↓ ↓ ← ↑ ↑
 ↑ ← ↓ ← ↓ → ↓ ↓ → → → ↓ ← ← ← ↓ ← ↑ → → ↑ ← ← ← ↓ → → ↓ ← ↓ → → ↑ ← ← ↑
 → ↓ → → → ↑ ← ← ↑ → → → → ↓ → ↑ ← ↑ → ↑ ← ↑ ← ↓ ↓ ↓ ← ↑ ↑ ← ↑ → ↑ ↑ → ↓
 → ↑ → → ↓ ↓ ← ↓ → ↑ ↑ → ↓ ↓ ↓ ↓ ↓ ← ← ← ← ↓ → → → → ↓ ← ← ↓ → → ↓ ← ↓ → ↓
 ← ← ↑ ↑ ← ↓ ↓ ← ← ↓ → → → → →

Beispiel 5

Aufgrund der Größe des Beispiels werden nur die ersten und letzten 200 Anweisungen angezeigt. Die volle Ausgabe findest du im Materialteil auf der Webseite.

Anzahl optimaler Lösungen $98673594646431832473600 \approx 9.9 \cdot 10^{22}$

Anzahl Anweisungen: 1308

→ → ↓ ↓ → → ↑ → ↑ → → ↓ ← ← ↓ ↓ ← ← ↓ → → ↓ ← ↓ → ↓ → → ↓ ← ← ← ↓ → ↓ ←
 ↓ → ↓ ↓ ← ↓ ↓ → → ↓ → ↓ → ↑ → ↓ ↓ ↓ ← ↓ → → ↓ → → ↓ ↓ ↓ ↓ ↓ ← ← ↓ → → → ↓ →
 → → ↑ → ↓ → ↓ ↓ ↓ ← ← ↓ ↓ ↓ ↓ → ↓ ↓ ← ↓ ↓ ↓ → ↓ → → ↑ ↑ → ↑ ↑ ↑ → → ↑ → → ↓ → ↑
 → → ↑ → ↓ → ↑ → ↑ ↑ → → ↓ ↓ ↓ ↓ → ↓ ↓ ↓ ↓ ↓ ↓ → ↓ → ↑ → ↓ ↓ ↓ ↓ ↓ ← ← ← ← ↓
 ← ← → ↓ → ↓ ← ← ↓ → → ↓ ↓ ↓ → → ↓ → ↑ ↑ → → ↓ ↓ ↓ ↓ ← ↓ ← ← ↓ ↓ → → ↓ ↓ → →
 ↓ → ↓ → ↓ → → ↑

...

↑ ↑ → → ↑ ← ↑ ↑ → ↑ ↑ ← ↑ ↑ → → ↓ → → ↑ → ↑ → → ↑ ↑ → → → → ↓ ↓ → ↓ ↓ → → ↓
 → ↑ → → ↓ ↓ → → ↓ ← ↓ → → ↓ → → ↓ ↓ → ← ↓ ← ↓ ↓ ↓ ← ← ← ↓ ↓ ↓ → ↓ ← ← ↑ ← ↓
 ↓ ← ↓ ← ↓ → → ↓ ↓ ↓ → ↑ → → ↓ → ↑ → ↓ → ↓ ↓ ← ↓ ← ↓ → → ↓ → ↓ → ↑ ↑ ↑ → ↑ ↑ →
 ↑ ← ↑ → → ↑ → ↑ → → → ↓ ↓ → ↑ → ↓ → → ↑ → ↓ → → ↑ ↑ ↑ → → → → → → → ↓ ↓ ↓
 ↓ ↓ → ↓ → ↓ → → ↓ → ↓ ← ↓ ↓ ← ← ← ↓ → ↓ → → → → ↓ ← ↓ ↓ ↓ ↓ ← ↓ → ↓ ↓ → → →
 → ↓ ↓ → → ↓ ↓ → → ↓

Beispiel 6

Aufgrund der Größe des Beispiels werden nur die ersten und letzten 200 Anweisungen angezeigt. Die volle Ausgabe findest du im Materialteil auf der Webseite.

Anzahl optimaler Lösungen: 1

Anzahl Anweisungen: 1844

```

→ → → ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ → → ↓ → ↓ ↓ → → → ↓ → ↓ → → ↑ → → ↓ ↓ → → → ↓ ↓ ↓ ↓ ←
← ↓ ← ↓ ↓ → ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ → ↓ → ↓ ↓ ← ↓ ↓ → → → ↑ ↑ → → → ↓ → → → → → → → ↑ →
→ ↓ → → ↑ → ↑ → → ↓ → → → → → → ↓ ↓ ↓ ↓ → → → ↓ ↓ → → → → → ↓ ↓ ↓ ↓ ←
↓ ← ← ↓ ← ↓ ↓ → → ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ → → ↓ ↓ → → ↓ → → → → ↓ → → → → ↓ ↓ → → ↓ → →
→ ↑ ↑ ← ↑ ↑ ↑ ↑ → → ↑ ↑ ↑ ↑ → → → → → ↓ → → ↓ → ↓ → → → ↓ ↓ → → → ↑ → ↑ → ↑
→ → ↑ → → → → ↓ → ↓ → ↓ → ↓

```

...

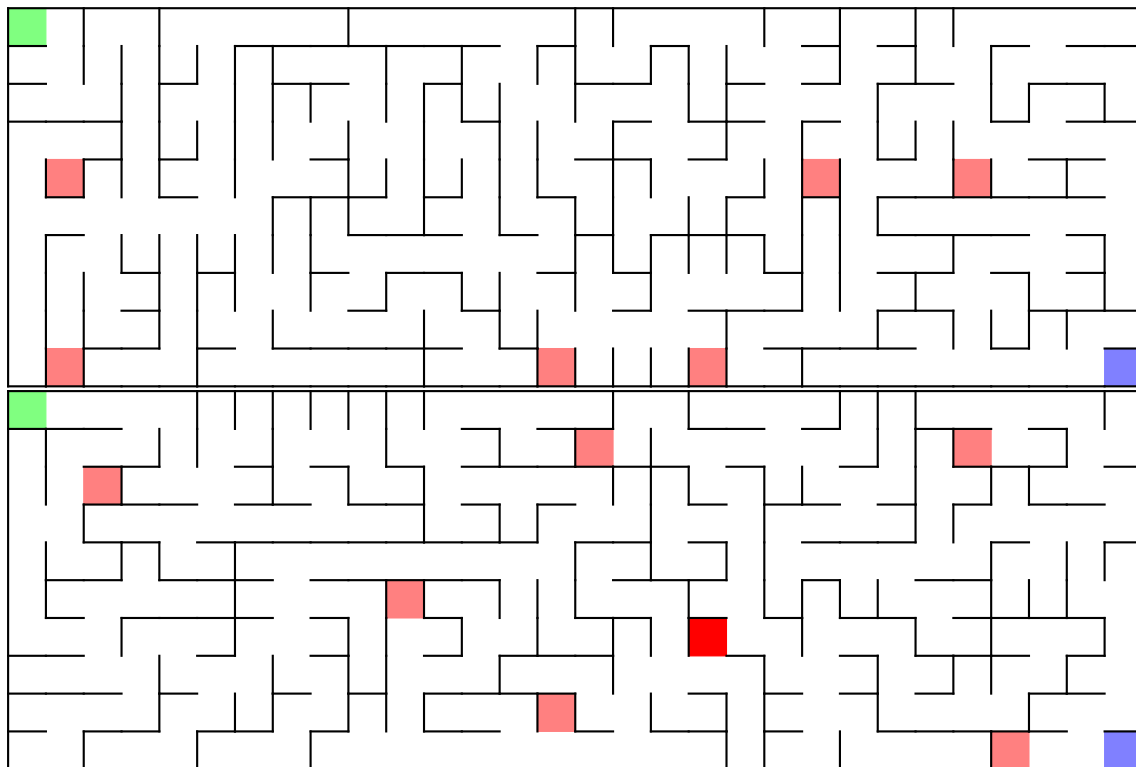
```

← ↓ ↓ ↓ ← ↓ ↓ ↓ ↓ ← ← ← ↑ ← ← ← ← ← ↓ ↓ ↓ ↓ ← ↓ ↓ ← ↓ ↓ ↓ ↓ ← ← ↑ ← ← ← ← ↓
← ↓ ← ↓ ↓ → ↓ → ↓ → ↓ ↓ ← ← ↓ ← ← ← ↑ ← ← ↑ ← ← ↓ ← ↓ ↓ ← ↓ ↓ ↓ ↓ → → ↑ → →
→ ↓ → → ↓ ↓ → → ↓ ↓ ← ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ → → → → → ↓ → → → → → → ↑ ↑ → ↑ ↑
→ ↑ → → ↓ ↓ → ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
→ → → → ↓ ↓ → → ↓ ↓ ← ↓ ↓ ↓ ← ↓ ↓ ↓ ← ↓ ↓ → → ↓ ↓ ← ← ← ← ← ↓ ↓ → → ↓ → →
→ ↓ → → ↑ ↑ → → → ↓ ↓

```

Beispiel 7

Bei diesem Beispiel gibt es keine Lösung, da das zweite (untere) Labyrinth nicht lösbar ist aufgrund einer Grube bei (14,8).



Beispiel 8

Anzahl optimaler Lösungen: $24831442944000 \approx 2.5 \cdot 10^{13}$

Anzahl Anweisungen: 472

```

↓ → ↓ ↓ ↓ → ↓ ↓ → ↓ ↓ ↓ → ↓ ↓ → ↓ → → → ↓ → → → ↓ → → ↑ → ↓ → ↓ ← ↓ → → ↓ →
↓ ↓ ↓ → ↓ ↓ ↓ → → ↓ ↓ ↓ → → ↑ → → ↓ → ↑ → ↓ → ↓ ↓ → ↑ → → → → → → → ↓ ↓ → → →

```

→ → ↓ → ↓ → → ↓ → → ↓ ↓ ↓ → ↓ → ↓ ↓ → ↓ → → ↓ → ↓ ← ↓ ← ↓ ↓ → ↓ ↓ ↓ → → ↓ →
 ↓ ↓ → ↓ → → → ↓ → ↓ ↓ ↓ → → → ↓ ↓ → ↓ ↓ ↓ → → ↓ ↓ ↓ → ↓ → ↓ ↓ ← ↓ → → ↓ → ↓ →
 ↓ → → ↓ → ↑ → → ↓ → → → → → ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ → ↓ ↓ → ↑ → ↓ ↓ → ↓ →
 ↓ ↓ → ↑ → ↓ → → ↓ → ↑ ↑ → ↑ ↑ ← ← ↑ ↑ ↑ ← ↑ → ↑ → ↑ → → ↑ → ↓ ↓ ↓ ↓ → ↓ ↓ ↓ →
 ↓ ↓ ↓ ↓ ↓ → → ↓ → ↓ → → → → ↑ → → → → ↓ → → → ↓ ↓ → ↓ ↓ ↓ ↓ ↓ ↓ ← ↓ ↓ → →
 ↓ → → ↓ ↓ ← ← ↓ ↓ → → → → ↑ ↑ → → → → ↑ ↑ → ↓ → → ↓ → ↑ → → → → → ↑ → →
 ↓ → → → ↓ ↓ ↓ ↓ → ↓ ↓ ↓ ↓ → ↓ ← ← ← ← ← ↓ ↓ ← ← ← ↓ → ↓ ← ← ↓ ↓ ← ↓ ↓ ↓ → ↓ ↓
 ↓ ↓ → → ↓ → → ↑ ↑ → → → ↑ → ↓ → ↓ ↓ ↓ → → → ↓ → ↑ → → ↓ ↓ → ↓ ↓ ↓ ↓ ↓ → → →
 → ↓ → ↓ → → → ↓ ↓ ↓ ↓ ← ← ↑ ← ← ↓ ↓ ↓ ← ← ↓ → ↓ → → ↓ ↓ ← ← ↓ ← ← ↓ ↓ ← ←
 ← ↓ → → ↓ ↓ → ↓ ↓ → → ↑ → → → → → ↓ → ↓ → → ↓ ↓ → ↓ → ↓ ← ← ↓ ↓ ↓ ↓ ↓ →
 → → → → → ↓ ↓ → ↑ → ↓ ↓ → ↓

Beispiel 9

Aufgrund der Größe des Beispiels werden nur die ersten und letzten 200 Anweisungen angezeigt. Die volle Ausgabe findest du im Materialteil auf der Webseite.

Anzahl optimaler Lösungen: 124797196418678784 $\approx 1.2 \cdot 10^{17}$

Anzahl Anweisungen: 1012

↓ → ↓ → → ↓ → → ↑ → → ↓ ↓ ↓ → ↑ → ↑ → → ↓ ↓ ↓ ↓ → → ↑ ↑ → → ↓ → ↓ → ↓ ↓ → ↓ ↓
 ← ↓ ← ↓ ← ↓ ↓ → ↓ ↓ → ↓ → → → ↑ → ↓ ↓ ↓ ↓ ↓ ↓ ← ↓ ← ↓ → ↓ ↓ → ↓ → ↑ → → ↑ → →
 ↑ ↑ → ↑ → ↑ ↑ ← → → ↓ ↓ → ↓ ↓ → → ↑ → → ↓ ↓ → ↓ ↓ → → → ↓ ← ↓ ← ← ← ↑ ↓ → ↑
 → ↓ → ↓ ← ← ← ↓ → → → ↓ ← ↓ ← ← ↓ → ↓ ← ↓ ← ← ↑ ↑ ← ← ↓ ← ↓ ↓ → ↓ ↓ → ↓ ←
 ↓ ↓ ↓ ↓ → ↓ ↓ → → ↓ ↓ ← ← ↓ ← ↓ ← ↓ ← ← ↑ ← ↓ ↓ ← ← ↑ ← ← ↓ ↓ ← ↓ ↓ → ↓ → → ↓
 ↓ ← ← ↓ ← ↓ ↓ →

...

↓ → ↑ → ↑ ↑ → ↓ → → ↑ ← ↑ → → ↓ → ↑ → → → → ↑ ↑ ↑ ← ↑ ↑ → ↑ → ↓ → → ↓ ↓ ↓ ↓ ↓
 → ↑ → → → → ↑ ↑ ↑ ↑ ↑ → ↓ ← ↓ ↓ ↓ ← ↓ ← ↓ ← ↓ ↓ → → ↓ → ↓ → → ↑ → ↑ → ↓ ↓ →
 ↓ ↓ ↓ ↓ → ↓ → → ↓ → → ↓ ← ↓ ↓ → ↑ → → → ↓ → → → ↓ ← ← ↓ ← ← ↑ ↑ ← ↑ ↑ ↑ ← ↓
 ← ↓ ↓ ↓ → ↓ ↓ ↓ ← ↓ ↓ ↓ → ↓ ← ↓ → ↓ ↓ ↓ ↓ ↓ ↓ ↓ → ↓ → ↑ → ↓ ↓ → ↓ ← ↓ → → ↓ → → ↑
 ↑ → ↑ → ↓ → ↓ → ↑ → ↓ → ↓ → → ↑ ↓ → ↑ ↑ ↑ → ↑ → ↓ → → ↓ ↓ ↓ ↓ ↓ → → → ↑ → → ↓
 ↓ ↓ ↓ ↓

2.9 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

- (1) *Problem adäquat modelliert*: Die Modellierung muss Folgendes sicherstellen:
 - Bewegen in alle vier Richtungen ist möglich.
 - Vor Wänden wird stehen geblieben.
 - Grubenfelder führen zurück zum Start.
 - Nach Erreichen eines Endfeldes stoppt die Bewegung.
- (2) *Verfahren nicht unnötig ineffizient*: Das Verfahren sollte Distanzen für gleiche Felder nicht mehrfach berechnen.
- (3) *Laufzeit des Verfahrens in Ordnung*: Falls mehrere Verfahren umgesetzt wurden, wird das Verfahren gewertet, welches die Endergebnisse produziert hat. Die effektive Laufzeit des Verfahrens sollte für alle Beispieleingaben im Minutenbereich liegen. Eine Breitensuche oder ein A* ohne weitere Optimierungen (z.B. effiziente Datenstrukturen, gute Distanzheuristik) ist in der Regel nicht ausreichend. Ein Verfahren mit schlechter worst-case Laufzeit aber besserer mittlerer Laufzeit (zum Beispiel A*) führt nur bei ungenügender Umsetzung zu Punktabzug. Können aufgrund der Laufzeit Ergebnisse nur für die ersten vier Beispieleingaben (bis `labyrinth3.txt`) berechnet werden, werden 6 Punkte abgezogen. Werden insbesondere die großen Beispiele `labyrinth4.txt` oder `labyrinth9.txt` nicht geschafft, so können bis zu 4 Punkte abgezogen werden.
- (4) *Speicherbedarf in Ordnung*: Der Speicherbedarf darf nicht größer als $O((nm)^2)$ sein.
- (5) *Verfahren mit korrekten Ergebnissen*: Liegt die Anzahl der Anweisungen unter dem *Optimum*, werden 4 Punkte abgezogen.

Dateiname	<i>Optimum</i>	<i>okay</i>	Anmerkung
<code>labyrinth0.txt</code>	8	10	Beispiel aus der Aufgabenstellung
<code>labyrinth1.txt</code>	31	35	Nicht quadratisch; Alle Felder müssen besucht werden (klein)
<code>labyrinth2.txt</code>	65	90	Für die optimale Lösung muss eine Grube verwendet werden
<code>labyrinth3.txt</code>	164	230	Extreme Seitenverhältnisse ohne Gruben
<code>labyrinth4.txt</code>	14384	18000	Alle Felder müssen besucht werden (groß)
<code>labyrinth5.txt</code>	1308	1800	Großes Beispiel mit Gruben
<code>labyrinth6.txt</code>	1844	2000	Keine Wände, nur Gruben
<code>labyrinth7.txt</code>	-	-	Keine Lösung (Grube im Weg)
<code>labyrinth8.txt</code>	472	660	Zyklen und nicht erreichbare Felder
<code>labyrinth9.txt</code>	1012	1300	Breitensuche wesentlich schlechter als A*

- (6) *Verfahren mit guten Ergebnissen*:
 - Sind die Ergebnisse von höchstens vier Beispielen größer als *okay*, werden 2 Punkte abgezogen, bei mehr Beispielen bis zu 4.
 - Ergebnisse kleiner als *okay* sind akzeptabel.
 - Für optimale Ergebnisse können bis zu 2 Pluspunkte vergeben werden.

2. Theoretische Analyse

- (1) *Verfahren / Qualität insgesamt gut begründet:* Die Wahl des Algorithmus und der Heuristik (falls vorhanden) muss begründet werden.
- (2) *Gute Überlegungen zur Laufzeit des Verfahrens:* Entspricht der Algorithmus einem Standardverfahren (z.B. Breitensuche, Dijkstra, A*), so kann die Laufzeit des Algorithmus ohne Herleitung angegeben werden. Andernfalls muss die Laufzeit des Algorithmus erklärt werden.
- (3) *Analysen bei mehreren Verfahren:* Wenn die Ergebnisse mehrerer Verfahren kommentarlos angegeben werden, können bis zu 2 Punkte abgezogen werden. Für eine Einsendung, die mehrere Heuristiken vergleicht und Begründungen für deren Qualität liefert, können Bonuspunkte vergeben werden.

3. Dokumentation

- (3) *Vorgegebene Beispiele dokumentiert:* Es müssen alle 10 Beispiele dokumentiert werden: `labyrinth0.txt` bis `labyrinth9.txt`. Ist kein Weg ins Ziel möglich, muss dies dokumentiert werden. Für die Anweisungsfolgen bei `labyrinth4.txt`, `labyrinth5.txt`, `labyrinth6.txt` und `labyrinth9.txt` kann auf eine externe Datei verwiesen werden.
- (5) *Ergebnisse nachvollziehbar dargestellt:* Für jedes Beispiel muss die Mindestanzahl für Anweisungen bis ins Ziel angegeben werden. Außerdem wird eine lesbare Anweisungsfolge erwartet. Eine Visualisierung der Labyrinth ist nicht nötig. Ist nur der Weg (und nicht die Anzahl der Anweisungen) gegeben, wird 1 Punkt abgezogen. Ist nur die Anzahl an Anweisungen (und nicht der Weg) gegeben, werden 2 Punkte abgezogen.

Aufgabe 3: Konfetti

In dieser Aufgabe geht es darum, die Spalten einer gegebenen Tabelle so zu permutieren, dass eine *zulässige* Anordnung entsteht. Die Einträge der Tabelle sind dabei entweder Kreuze oder Haken oder ab Teilaufgabe d) auch Fragezeichen. Im Folgenden werden wir zur einfacheren Darstellung statt \times 0 und statt \checkmark 1 verwenden.

Eine Anordnung wird *zulässig* genannt, wenn in jeder Zeile alle 1en hintereinander stehen (also keine 0 zwischen zwei 1en steht). Das bedeutet im Sachzusammenhang, dass alle Zeitblöcke zusammenhängen, zu denen eine Person Zeit hat. In Teilaufgabe d) können zusätzlich statt 0 und 1 Fragezeichen auftreten, diese müssen passend durch 0 oder 1 ersetzt werden, bevor die entstehende Tabelle permutiert wird. In der letzten Teilaufgabe können zusätzlich Einträge verändert werden, die in der ursprünglichen Tabelle durch 0 oder 1 vorgegeben waren. Hierbei sollen möglichst wenige Einträge verändert werden, damit eine zulässige Permutation entstehen kann. Es ist auch interessant, welche Einträge verändert werden müssen.

Um dies etwas zu formalisieren, definieren wir zunächst folgende Begriffe:

Sei eine Tabelle T definiert als eine Folge $(col_i)_{i \in \{1..c\}}$ ⁷ von c Spalten, wobei jede Spalte $col_i \in \Sigma^r$ eine Folge von r Einträgen ist. Die Einträge seien Teil der Menge der in der Tabelle vorkommenden Zeichen Σ , für Teilaufgabe a), b), c) ist $\Sigma := \{0, 1\}$, für d) und e) $\Sigma := \{0, 1, ?\}$.

Der Eintrag einer Spalte col_i in Zeile j sei $col_i[j]$, $j \in \{1 .. r\}$.

Eine Tabelle T mit $\Sigma = \{0, 1\}$ heißt *zulässig*, wenn in jeder Zeile alle 1en in aufeinanderfolgenden Spalten liegen. Für eine einzelne Zeile nennen wir diese Eigenschaft die *Zeilenbedingung* dieser Zeile.

Eine Tabelle T mit $\Sigma = \{0, 1, ?\}$ heißt *zulässig*, wenn jedes „?“ so durch „0“ oder „1“ ersetzt werden kann, dass die entstehende Tabelle mit nur „0“, „1“ *zulässig* ist. Es ist zu beobachten, dass dies genau dann der Fall ist, wenn in der Zeile, die man erhält, wenn man alle „?“ weglässt, alle 1en aufeinanderfolgen.

Formal wird in den ersten Teilaufgaben also eine Permutation der Spalten gesucht, also $\pi : \{1 .. c\} \rightarrow \{1 .. c\}$ bijektiv⁸, sodass die permutierte Tabelle $T' = (col_{\pi(i)})_{i \in \{1..c\}}$ *zulässig* ist. Gibt es eine solche Permutation für eine Tabelle T , so heißt T *zulässig permutierbar*.

Das dazugehörige Entscheidungsproblem⁹ sei TABLE REORDER WITH/WITHOUT ? (abgekürzt „TR“/„TR?“) mit folgender Definition: Gegeben eine Tabelle T mit $\Sigma = \{0, 1\}$ / $\Sigma = \{0, 1, ?\}$, ist T *zulässig permutierbar*?

3.1 Teilaufgaben a)-c): PQ-Trees

Unser Ziel ist es, eine Datenstruktur zu entwickeln, welche alle möglichen Permutationen der Spalten enthält, für welche die Zeilenbedingungen erfüllt sind. Dazu lassen sich zunächst folgende Beobachtungen machen:

⁷ $\{i .. j\}$ ist eine verkürzte Schreibweise für $\{i, i+1, \dots, j\}$

⁸Jedes Element wird genau einmal „zugeordnet“.

⁹Ein Entscheidungsproblem ist ein Problem, bei dem die Antwort einfach entweder „ja“ oder „nein“ ist. Formell kann man ein Entscheidungsproblem auch als Menge aller Eingaben definieren, für die die Antwort „ja“ ist. Ist X ein Entscheidungsproblem, bedeutet $x \in X$ also einfach, dass die Antwort für die Eingabe x „ja“ ist.

```

1 1 1 1 1 1 1 1 0 0 0 | 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 | 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 | 1 1 1 1 1

```

Abbildung 3.1: Eine Tabelle mit zwei unabhängigen Abschnitten.

- Hat man eine *zulässige* Permutation gefunden und „dreht“ die gesamte Tabelle um 180°, so erhält man wieder eine *zulässige* Permutation.
- Zulässige Tabellen kann man oft in Abschnitte unterteilen, wobei eine Zeile immer nur innerhalb eines Abschnitts 1en enthält. Die Reihenfolge dieser Abschnitte in der Endanordnung spielt keine Rolle, sie können beliebig permutiert werden.
- Lässt sich die Tabelle nicht in dieser Weise unterteilen, so gibt es „Abschnitte“, deren Reihenfolge nicht verändert werden darf (bis auf das Umdrehen der gesamten Tabelle).
- In beiden Fällen gelten für diese (Teil-)Abschnitte ebenfalls die beiden vorherigen Bedingungen.

Es lässt sich also eine rekursive Struktur des Problems feststellen, daher liegt es nahe, die Datenstruktur ebenfalls rekursiv zu konstruieren. Nun benötigt man natürlich noch einen „Base Case“ (Blatt des späteren Baumes), hierfür verwenden wir die einzelnen Spalten der ursprünglichen Tabelle.

Analog zu den Fällen oben wollen wir nun zwei Fälle für unsere Datenstruktur, welche einen Abschnitt beschreibt, unterscheiden:

1. Die „Kindabschnitte“ (Abschnitte, aus denen dieser Abschnitt besteht) sind beliebig permutierbar.
2. Die Kindabschnitte kommen immer in einer festen Reihenfolge vor. Die einzige Möglichkeit, eine andere Spaltenreihenfolge (abgesehen von Änderungen innerhalb der Kindabschnitte) zu erhalten, ist, den gesamten Abschnitt umzudrehen.

Die genannten Eigenschaften lassen sich hervorragend mit einem PQ-Tree umsetzen. Die inneren Knoten eines PQ-Trees sind immer entweder P- oder Q-Nodes. Bei P-Nodes, dürfen die Kindknoten beliebig permutiert werden, dies beschreibt also den ersten beobachteten Fall. Bei Q-Nodes hingegen ist die Reihenfolge der Kindknoten fest, sie kann lediglich vollständig umgedreht werden.

Jeder Abschnitt der Tabelle, wird nun durch PQ-Nodes modelliert (also einem P- oder einem Q-Node).

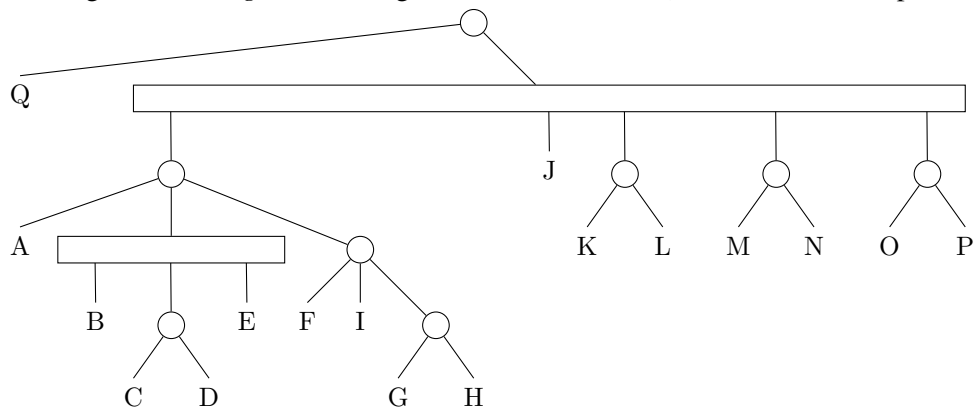
Ein PQ-Tree besteht aus mehreren solchen PQ-Nodes, wobei einer davon die Wurzel ist und die anderen PQ-Nodes nur indirekt über diese Wurzel referenziert werden. Einen PQ-Tree stellen wir also als Baum dar, wobei die Spalten die Blätter und P- und Q-Nodes innere Knoten sind. P-Nodes stellen wir als Kreis, Q-Nodes als Rechteck dar (Abbildung 3.2).

Man beachte, dass jede PQ-Node wieder als Wurzel eines PQ-Trees interpretiert werden kann: Als Wurzel eines Teilbaums des gesamten PQ-Trees. Im Folgenden wird mit einer PQ-Node daher manchmal auch der dazugehörige Teilbaum bezeichnet.

Als erste Optimierung fügen wir ein, dass alle PQ-Nodes, die nur einem Kindknoten haben, einfach durch diese ersetzt werden können. Alle inneren Knoten haben also mindestens zwei Kindknoten.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0
3	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
4	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0

(a) Eine mögliche *zulässige* Anordnung für `konfetti4.txt` (Die Zeilen wurden permutiert).



(b) Ein PQ-Tree, der alle möglichen Permutationen von `konfetti4.txt` darstellt. Die Blätter des PQ-Trees sind die Spalten der oberen Tabelle. Kreise stehen für P-Nodes, Rechtecke für Q-Nodes.

Abbildung 3.2: Ein Beispiel für einen PQ-Tree.

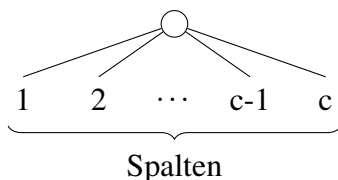


Abbildung 3.3: Initialer PQ-Tree

Um eine mögliche Permutation aus einem PQ-Tree abzulesen, nutzen wir die Reihenfolge der Blätter bei einer Traversierung des Baumes. Dabei dürfen wir die Kinder von P-Nodes in beliebiger Reihenfolge besuchen (Permutation der Kindabschnitte), die Kinder von Q-Nodes jedoch nur in der angegebenen Reihenfolge oder genau umgekehrt.

Es lässt sich beobachten, dass die einzelnen Zeilenbedingungen prinzipiell unabhängig¹⁰ voneinander sind. Wir wollen die Zeilenbedingungen daher nacheinander behandeln. Weiterhin lässt sich jede Zeilenbedingung so umformulieren, dass die Menge von Spalten mit einer 1 in dieser Zeile hintereinander kommen müssen.

Die Idee ist nun, dass man den Baum Stück für Stück aufbaut, indem für jede Zeile nacheinander dafür gesorgt wird, dass die Zeilenbedingung erfüllt bleibt. Dazu unterstützen wir folgende Operation: Gegeben ist eine Menge an Spalten (Blättern), der Baum wird nun so verändert, dass die gegebenen Spalten aufeinander folgen. Dabei sollen jedoch alle schon im Baum kodierten Zusammenhänge beibehalten werden. Diese Operation nennen wir $\text{REDUCE}(T, S)$, wobei T der PQ-Tree und S die Spaltenmenge ist. Sie gibt den veränderten PQ-Tree zurück. REDUCE wird nun Zeile für Zeile mit der Menge der Spalten, die in dieser Zeile eine 1 haben, aufgerufen, denn genau diese müssen nebeneinander liegen. Es kann vorkommen, dass es keinen solchen Baum gibt, in welchem Fall die Funktion fehlschlägt.

Auf diese Weise codiert der PQ-Tree nach der i -ten Iteration alle möglichen Permutationen, sodass die Zeilenbedingungen der ersten i Zeilen erfüllt sind. Vor der ersten Iteration gibt es keine Bedingungen, die gelten sollen, also müssen alle Permutationen möglich sein. Dies erreichen wir durch einen PQ-Tree, dessen Wurzel eine P-Node ist und welche alle Spalten als direkte Kinder erhält. (s. Abbildung 3.3)

Algorithmus 6 FIND PERMUTATION

```

function FIND PERMUTATION(Table T)
   $T \leftarrow$  P-Node with all columns as direct children
  for do  $j = 1, 2, \dots, r$ 
     $S \leftarrow \{col_i \mid col_i[j] = 1\}$        $\triangleright$  Die Menge aller Spalten, die in Zeile  $j$  eine 1 haben.
     $T \leftarrow \text{REDUCE}(T, S)$ 
    if REDUCE failed then
      Report that there is no solution, break.
  Extract a solution from  $T$ .
```

Um REDUCE zu implementieren, müssen wir für einen PQ-Node wissen, welche der Kinder Spalten enthalten, die eine 1 bzw. eine 0 in der betrachteten Zeile haben. Bevor wir die eigentlichen Veränderungen durchführen, teilen wir also alle PQ-Nodes in eine der folgenden Kategorien ein: ONLY0, ONLY1, BOTH.

¹⁰Die Zeilenbedingungen können zwar neue Zusammenhänge und potenzielle Konflikte bilden, hier geht es jedoch darum, dass für die Erfüllung einer einzelnen Zeilenbedingung nur die entsprechende Zeile relevant ist.

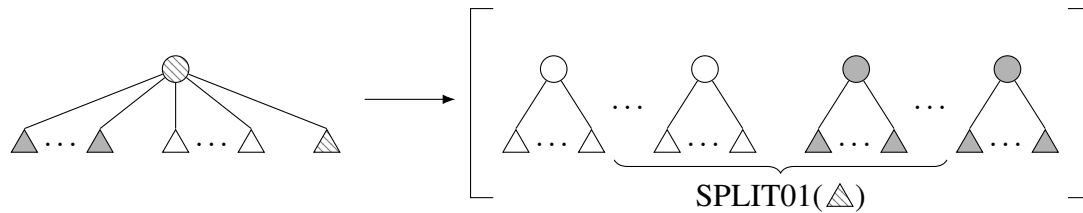


Abbildung 3.4: SPLIT01 auf P-Nodes (einziger erfolgreicher Fall)

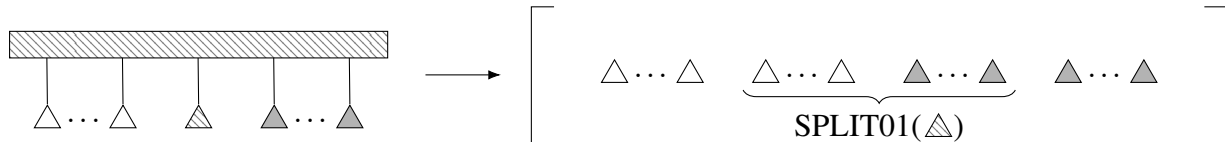


Abbildung 3.5: SPLIT01 auf Q-Nodes (einziger erfolgreicher Fall)

- Eine PQ-Node mit ONLY0 enthält nur Spalten mit einer 0 in der betrachteten Zeile.
- Eine PQ-Node mit ONLY1 enthält nur Spalten mit einer 1 in der betrachteten Zeile.
- Eine PQ-Node mit BOTH enthält sowohl Spalten mit dem Wert 0, also auch Spalten mit dem Wert 1 in der betrachteten Zeile.

In den folgenden Abbildungen werden nun PQ-Nodes mit ONLY0 weiß, PQ-Nodes mit ONLY1 grau, und PQ-Nodes mit BOTH schraffiert dargestellt.

Wenn alle Spalten mit dem Eintrag 1 nebeneinander liegen sollen, können PQ-Nodes mit BOTH nur am Rand dieser Einerblöcke vorkommen. Um die Spalten innerhalb der BOTH PQ-Nodes so anzuordnen, dass alle 0en und 1en gebündelt vorliegen, benötigen wir eine weitere Funktion. Wir implementieren die Funktion $SPLIT01(T, S)$, welche eine Liste von PQ-Nodes zurückgibt. Diese PQ-Nodes enthalten zusammen genau die Spalten des PQ-Trees T . In der Liste kommen zuerst alle PQ-Nodes, welche ONLY0 sind, dann alle mit ONLY1. In dieser Liste sind keine PQ-Nodes mit BOTH enthalten, werden sie also in der genau dieser Reihenfolge in einen Q-Node eingefügt¹¹, ist in diesem Bereich die Zeilenbedingung erfüllt.

Betrachten wir nun, wie wir SPLIT01 implementieren können.

Haben wir einen P-Node, so können wir die Kindknoten beliebig permutieren. Wir gruppieren diese also zunächst nach ONLY0, ONLY1 und BOTH. Es ist zu beachten, dass es maximal ein Kind mit BOTH geben darf; sonst ist es nicht möglich, nur einen Wechsel von 0 nach 1 zu haben (und keinen zurück nach 0). Ist dies nicht der Fall, so brechen wir ab, es existiert keine Lösung. Ansonsten müssen wir beachten, dass zunächst alle Kindknoten mit ONLY0 kommen müssen. Da wir jedoch deren Reihenfolge nicht wissen, erstellen wir als erstes Element der Liste einen P-Node, welcher alle Kindknoten mit ONLY0 enthält. Analog erstellen wir als letztes Listenelement einen P-Node mit allen ONLY1-Kindknoten. Haben wir einen Kindknoten mit BOTH, so müssen wir diesen nun ebenfalls aufteilen, wir rufen also auch SPLIT01 auf dem Knoten auf. Die dadurch erhaltenen Elemente sind die mittleren Elemente unserer Liste; diese fügen wir direkt ein, ohne sie in eine andere PQ-Node zu „verpacken“. Hier ist es nämlich weder möglich, die Elemente zu permutieren, oder den Bereich umzudrehen, da dann die Zeilenbedingung nicht mehr erfüllt wäre. (s. Abbildung 3.4)

Liegt jedoch eine Q-Node vor, so dürfen wir die Reihenfolge der Kindknoten nicht ändern. Hier gilt daher, dass in der Reihenfolge der Kindknoten bereits alle ONLY0-Knoten zuerst, dann

¹¹Wir geben eine Liste zurück statt einen Q-Node, da wir diesen Bereich nicht einfach umdrehen können.

Algorithmus 7 SPLIT01

```

function SPLIT01(T, S)
  if T is a P-Node then
     $zeros \leftarrow$  set of ONLY0 child nodes
     $ones \leftarrow$  set of ONLY1 child nodes
     $boths \leftarrow$  set of BOTH child nodes
    if  $|boths| > 1$  then
      Report that there is no solution, break.
    else if  $|boths| = 1$  then
      Let  $B$  be the only element of  $boths$ .
      return  $[P - Node(zeros)] \circ SPLIT01(B, S) \circ [P - Node(ones)]$ 
    else return  $[P - Node(zeros), P - Node(ones)]$ 
  else if T is a Q-Node then
    if children match  $[ONLY1^*, BOTH?, ONLY0^*]$  then
       $\triangleright$  * arbitrary number, ? at most one
      reverse children
    if children match  $[ONLY0^*, BOTH?, ONLY1^*]$  then
      Replace the BOTH child  $B$  with  $SPLIT01(B, S)$  in children (if it exists)
      return children
    else
      Report that there is no solution, break.
  else return  $[T]$ 
 $\triangleright T$  is a single Column

```

maximal ein Knoten mit BOTH und schließlich alle ONLY1-Knoten kommen müssen. Der einzige andere mögliche Fall ist der genau umgekehrte, dann können wir die Reihenfolge umdrehen und weiter genauso wie im ersten Fall verfahren. In jedem anderen Fall (insb. auch jeder mit mehr als einem BOTH-Knoten) ist es nicht möglich, die Zeilenbedingung zu erfüllen und wir müssen abbrechen. Die einzige Änderung, die wir an der Liste der Kindknoten vornehmen müssen, ist, dass wir die BOTH-Knoten durch die Elemente der „Aufteilung“ des Knotens ersetzen indem wir SPLIT01 aufrufen (s. Abbildung 3.5).

Wir erhalten also Algorithmus 7.

Nun können wir die Funktion SPLIT01 nutzen, um REDUCE zu implementieren.

Liegt eine einzelne Spalte vor, müssen wir wie bei SPLIT01 nichts verändern, wir geben die Spalte selbst zurück. Ansonsten können wir feststellen, dass wir Knoten, welche ONLY0 sind, nicht verändern müssen.

Weiterhin müssen wir, wenn es nur genau einen Kindknoten mit 1en gibt (ONLY1 oder BOTH), auch nur diesen verändern. Unabhängig davon, ob es sich um eine P- oder Q-Node handelt, müssen wir diese Kindknoten C lediglich durch $REDUCE(C, S)$ ersetzen. Hier ersetzt das Ergebnis jedoch nur den Kindknoten, wir „gliedern“ nichts in den Knoten ein (es ist uns schließlich egal, ob der reduzierte Teilbaum umgedreht eingefügt wird oder nicht; er enthält alle 1sen der Zeile).

Bei einer P-Node müssen wir dafür sorgen, dass alle Kindknoten mit einer 1 nebeneinander gelangen; die Reihenfolge der anderen Knoten soll jedoch nicht eingeschränkt werden. Daher entfernen wir alle Kindknoten mit einer 1 aus dem P-Node und ersetzen diese durch einen einzigen neuen Q-Node. In dieser müssen die maximal zwei (der entstehende 1en-Block hat zwei „Ränder“) Kindknoten mit BOTH am Anfang und Ende kommen; damit der Block auch zusammen-

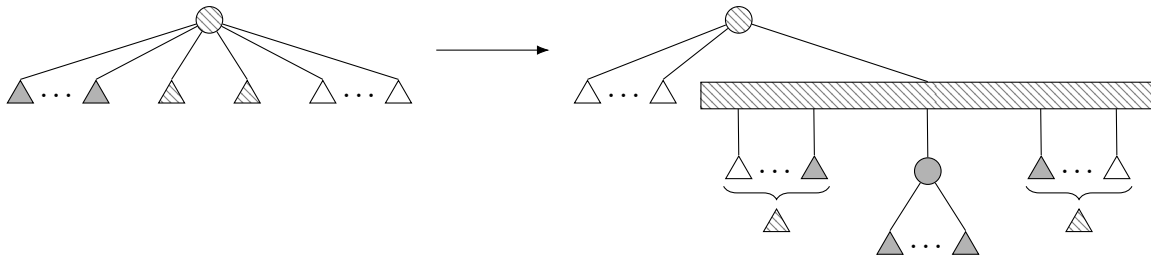


Abbildung 3.6: REDUCE auf P-Nodes (einziger erfolgreicher Fall)

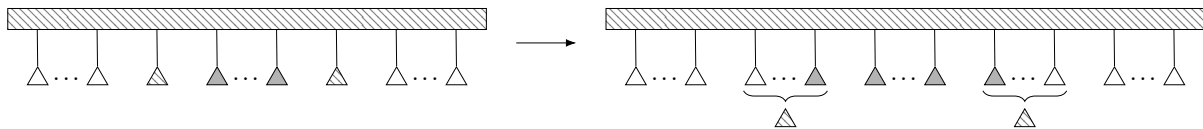


Abbildung 3.7: REDUCE auf Q-Nodes (einziger erfolgreicher Fall)

hängend ist, müssen sie zudem mit SPLIT01 aufgeteilt werden (bei dem zweiten Kindknoten müssen wir die von SPLIT01 zurückgegebene Liste noch umdrehen). Zwischen diesen „Enden“ müssen dann alle Kindknoten mit ONLY1 eingefügt werden. Da wir deren Reihenfolge nicht wissen, fügen wir stattdessen eine P-Node ein, welche all diese enthält. (s. Abbildung 3.6).

Bei einer Q-Node muss die Liste der Kindknoten folgendem Muster entsprechen: $[ONLY0^*, BOTH?, ONLY1^*, BOTH?, ONLY0^*]$. Wir müssen dann lediglich den ersten BOTH-Node mit SPLIT01 aufteilen und durch das Ergebnis ersetzen; den zweiten BOTH-Node ersetzen wir durch das umgedrehte Ergebnis („SPLIT10“) (s. Abbildung 3.7).

Wir erhalten also Algorithmus 8.

Hiermit können wir nun also wie in Algorithmus 6 beschrieben einen PQ-Tree erstellen, der alle möglichen Permutationen der Tabelle codiert.

Tiefe des PQ-Trees

Für die Laufzeitanalyse ist es essentiell zu analysieren, wie oft die Funktionen SPLIT01 und REDUCE aufgerufen werden. Da diese Aufrufe von der Wurzel bis zu den Blättern „laufen“, ist auch die Tiefe des PQ-Trees interessant. Daher werden wir diese im Folgenden beschränken.

Es lässt sich feststellen, dass durch das Ersetzen eines Teilbaums T durch $SPLIT01(T, S)$ der Baum nie tiefer wird. SPLIT01 kann zwar neue Knoten erstellen, diese werden bei der Ersetzung jedoch maximal auf der gleichen Tiefe wie der ursprüngliche Knoten T eingefügt.

Zudem lässt sich beobachten, dass nur auf einem Knoten eine „richtige“ REDUCE-Operation ausgeführt wird (je Aufruf auf der Wurzel), also eine REDUCE-Operation, welche nicht einfach den Baum unverändert lässt oder REDUCE auf genau einem Kindknoten aufruft. Dieser „richtige“ REDUCE-Aufruf ruft selbst nur noch SPLIT01 auf (und nicht mehr REDUCE), weshalb je Aufruf auf der Wurzel nur ein Knoten so bearbeitet wird. Findet dieser Aufruf auf einer P-Node statt, so erhöht sich die Tiefe des Baums um 2 (vgl. Abbildung 3.6); handelt es sich um eine Q-Node, so wird die Tiefe nicht größer.

Da insgesamt r REDUCE-Aufrufe stattfinden, lässt sich die Höhe des PQ-Trees also durch $\mathcal{O}(r)$ beschränken.

Algorithmus 8 REDUCE

```

function REDUCE(T, S)
  if T is a single column then return T
  else if T is ONLY0 then return T
    ▷ only possible when T is the complete tree and we have a only-0 row
  else if T has only one non-ONLY0 child node C then
    Replace C with REDUCE(C, S) in children
    return T
  else if T is a P-Node then
    zeros  $\leftarrow$  set of ONLY0 child nodes
    ones  $\leftarrow$  set of ONLY1 child nodes
    boths  $\leftarrow$  set of BOTH child nodes
    if |boths| > 2 then
      Report that there is no solution, break.
      Let  $B_1$  and  $B_2$  be the nodes in boths.
      ▷ If they do not exist, ignore the terms containing them.
    return P-Node(zeros  $\cup$  {Q-Node(SPLIT01( $B_1$ , S)  $\circ$  [P-Node(ones)]  $\circ$  reverse SPLIT01( $B_2$ , S))})
  else if T is a Q-Node then
    if children match [ $ONLY0^*$ ,  $B_1 := BOTH^*$ ,  $ONLY1^*$ ,  $B_2 := BOTH^*$ ,  $ONLY0^*$ ] then
      Replace  $B_1$  with SPLIT01( $B_1$ , S) in children
      Replace  $B_2$  with reverse SPLIT01( $B_2$ , S) in children
    return T

```

Weiterhin lässt sich beobachten, dass SPLIT01 nur maximal einen rekursiven Aufruf auf einem Kindknoten machen kann. Zudem tätigt REDUCE entweder maximal einen REDUCE-Aufruf auf einem Kindknoten oder maximal zwei SPLIT01-Aufrufe auf zwei Kindknoten. Insgesamt finden je „Ebene“ also maximal zwei REDUCE oder SPLIT01-Aufrufe statt, also insgesamt maximal $\mathcal{O}(r)$ Aufrufe.

Laufzeit

Wir betrachten nun die Laufzeit einer REDUCE-Operation und der dafür nötigen Vorberechnung.

Sowohl Algorithmus 7 als auch Algorithmus 8 sind (abgesehen von den anderen Funktionsaufrufen) in Linearzeit in der Anzahl der Kindknoten implementierbar¹² (Jeder einzelne Fall ist in Linearzeit implementierbar). Ebenso ist die Berechnung, ob eine PQ-Node ONLY0, BOTH oder ONLY1 ist, linear in der Anzahl der Kindknoten, wenn für diese schon bestimmt wurde, ob sie ONLY0, ONLY1 oder BOTH sind. Insgesamt wird für jeden Knoten also eine Laufzeit linear in der Anzahl seiner Kinder benötigt.

Die Anzahl der PQ-Nodes ist durch $2 \cdot c \in \mathcal{O}(c)$ beschränkt, da der Baum c Blätter hat und alle inneren Knoten mindestens zwei Kinder (sonst ersetzen wir ihn durch das Kind). Also ist unsere Laufzeit beschränkt durch $2 \cdot c + \sum_{\text{node } v} \langle \text{number of children of } v \rangle$.¹³ Wir können zudem beobachten, dass jede der PQ-Nodes bis auf die Wurzel ein Kind von genau einer anderen

¹²Die zu konkatenierenden Listen können zwar mehr Elemente haben, mit Linked-Lists ist dies aber in $\mathcal{O}(1)$ möglich.

¹³Die $2 \cdot c$ stehen für die konstanten Kosten je Knoten

PQ-Node ist. Jede PQ-Node kommt also in maximal einer der Kindknotenmengen vor. Also ist $\sum_{\text{node } v} \langle \text{number of children of } v \rangle \leq \langle \text{number of nodes} \rangle \leq 2 \cdot c$. Die Laufzeit einer REDUCE-Operation (und der Vorberechnung) ist also insgesamt auch durch $\mathcal{O}(c)$ beschränkt.

Da wir diese beiden Operationen für jede Zeile genau einmal aufrufen, ist die Gesamtlaufzeit unseres Algorithmus 6 durch $\mathcal{O}(r \cdot c)$ beschränkt. Dies ist für uns nicht zu verbessern¹⁴, da wir diese Laufzeit bereits zum Einlesen der $r \cdot c$ Tabelleneinträge benötigen.

Teilaufgabe b) (Lösung eindeutig?)

Zunächst einmal gibt es zwei verschiedene Möglichkeiten, wie wir die Eindeutigkeit einer Tabelle definieren können. Diese unterscheiden sich lediglich darin, ob das Vertauschen zweier identischer Spalten zu einer gleichen Tabelle führt. Im Sachzusammenhang sollte es Anna eigentlich egal sein, in welcher Reihenfolge identische Spalten auftauchen. Tabellen, welche sich nur durch die Reihenfolge ihrer identischen Spalten unterscheiden (wenn die Spaltenindizes nicht angegeben werden, also gar nicht), betrachten wir also als gleich.

Es lässt sich feststellen, dass alle identischen Spalten im PQ-Tree immer Kinder der gleichen PQ-Nodes sein werden. Ferner ist dieser Elternknoten immer eine P-Node, da die Reihenfolge der Spalten nicht relevant ist. Daher werden wir gleiche Spalten vor Anwendung des obigen Algorithmus zusammenfassen; wir müssen lediglich bei den Blättern zusätzlich speichern, wie oft die Spalte vorkommt (wie viele Spalten also zusammengefasst wurden).

Wenn eine (mehrfache) Spalte nun das Kind einer P-Nodes ist, so müssen wir beim Finden einer Permutation noch beachten, dass alle diese Spalten nicht notwendigerweise direkt nacheinander kommen müssen. Es könnten auch noch andere Kindknoten der P-Nodes zwischen den identischen Spalten vorkommen.

Ist sie das Kind eines Q-Nodes, so müssen die Spalten direkt hintereinander kommen.

Haben wir identische Spalten zusammengefasst, so haben wir nun den Vorteil, dass eine Eindeutigkeit ohne Betrachten der Reihenfolge identischer Spalten in der ursprünglichen Tabelle das Gleiche ist wie Eindeutigkeit in der komprimierten Tabelle. Da es nun aber keine doppelten Spalten mehr gibt, müssen wir die „Arten“ der Eindeutigkeit nicht mehr unterscheiden.

Wir können nun beobachten, dass es eine eindeutige Permutation genau dann gibt, wenn wir nur eine (komprimierte) Spalte vorliegen haben (also alle Spalten gleich sind). Liegen mehrere Spalten vor, so können wir nämlich in jedem Fall eine weitere Permutation konstruieren. In dem Fall, dass die Lösung nicht symmetrisch ist, können wir die gesamte Tabelle einfach umdrehen und erhalten eine andere Tabelle. Ist die Lösung jedoch symmetrisch, so sind die erste und letzte Spalte gleich, wir können die erste Spalte entfernen und am Ende anfügen. Also erhalten wir in beiden Fällen eine weitere Lösung.

Die Eindeutigkeit einer Lösung wird daher erst dann interessant, wenn eine Zeile gegeben ist. Das ist in Teilaufgabe c der Fall: Hier erinnert sich Anna an die ursprünglich erste Zeile des Zettels.

Teilaufgabe c) (gegebene Zeile)

Zunächst einmal lässt sich feststellen, dass die Anzahl der 1en in der ersten Zeile der potenziellen gültigen Anordnung mit der Anzahl der 1sen in der gegebenen Zeile übereinstimmen

¹⁴bezogen auf die asymptotische Laufzeit

muss. Da es sonst keine Lösung geben kann, gehen wir im Folgenden davon aus, dass diese Bedingung zutrifft.

Nun lässt sich die erste Zeile eindeutig durch den Index der ersten 1 in der Zeile beschreiben, da danach der 1en-Block mit der richtigen Anzahl 1sen folgen muss.

Statt nur zu berechnen, ob es eine Permutation mit dem gleichen Startindex wie die gegebene erste Zeile gibt, berechnen wir diese Information für alle möglichen Startindizes. Diese Information werden wir zudem für alle PQ-Nodes unseres PQ-Trees berechnen (außer sie sind ONLY0 oder ONLY1 bezüglich der ersten Zeile). Dabei berechnen wir die möglichen Startindizes innerhalb des Teils der Tabelle, welcher durch den Teilbaum der PQ-Node beschrieben wird. Im Wurzelknoten ist dies die gesamte Tabelle, wir können dann also überprüfen, ob die erste Zeile am gesuchten Index (der gegebenen Zeile) beginnen kann. Es wird sich feststellen lassen, dass wir im Algorithmus ohne viel Mehraufwand auch gleich mitberechnen können, ob es mehrere Möglichkeiten für einen Index gibt. Wir berechnen also statt der einfachen Information, ob ein Index möglich ist, die Information, ob ein Index unerreichbar, eindeutig möglich oder mehrdeutig möglich ist.

Es lässt sich beobachten, dass nach einer REDUCE-Operation für eine Zeile für jeden Node einer der folgenden drei Fälle gilt:

1. Der Knoten enthält keine Spalte mit einer 1 in dieser Zeile (ONLY0).
2. Der Knoten enthält nur Spalten mit einer 1 in dieser Zeile (ONLY1).
3. Sonst (BOTH) enthält der Knoten alle Spalten mit einer 1 in dieser Zeile; es gibt auf jeder Ebene des Baums maximal einen BOTH-Knoten.

Zudem lässt sich feststellen, dass, sobald es irgendwo im Baum einen ONLY0- oder eine ONLY1-Knoten gibt, die Lösung auf keinen Fall eindeutig sein kann (wenn eine existiert). Im komprimierten PQ-Tree muss ein P- oder Q-Node verschiedene Kinder enthalten (immer mindestens zwei Kinder, gleiche Spalten zusammengefasst). Also ist der Teil der Permutation, der einem ONLY0- oder ONLY1-Knoten entspricht, nicht eindeutig (aus PQ-Nodes mit verschiedenen Kindern lassen sich immer mehrere Permutationen konstruieren). Da dieser Teil durch die erste Zeile zudem nicht eingeschränkt wird, gibt es also auch mehrere Permutationen für die gesamte Tabelle. Abgesehen von diesem Fall kann die Lösung nur mehrdeutig werden, wenn es mehrere Möglichkeiten gibt, die Kinder der BOTH-Knoten so anzuordnen, dass die erste Zeile stimmt.

Wir betrachten nun den untersten BOTH-Knoten im PQ-Tree. Hier muss man bestimmen, an welchem Index der erste Kindknoten mit 1sen stehen kann. Dies ist äquivalent zu der Frage, welche Längen von Präfixen¹⁵ mit ONLY0-Knoten man bilden kann.

Alle weiteren BOTH-Knoten haben genau einen anderen BOTH-Knoten als Kind. Hier muss man bestimmen, welche Präfixlängen vor Beginn dieses Kindknotens möglich sind. Dies kann man dann mit den möglichen Startindizes innerhalb des Kindnotens kombinieren, um die möglichen Startindizes zu erhalten.

Betrachten wir zunächst das Problem, die möglichen Präfixe (bzw. Präfixlängen) zu bestimmen. Dieses Problem entspricht bei P-Nodes dem (NP-vollständigen) SUBSET-SUM-Problem¹⁶.

¹⁵Anfangssequenz der Permutation, siehe Aufgabe 1

¹⁶Wir werden hier dennoch für unseren Fall einen Polynomialzeitalgorithmus angeben. Dieser entspricht einem pseudopolynomiellen Algorithmus von SUBSET SUM, d.h. seine Laufzeit ist polynomiell in der Eingabegröße und der größten binär codierten Zahl der Eingabe (also hier des größten Wertes). Dies ist kein Polynomialzeitalgorithmus, da dafür die Laufzeit polynomiell in der Eingabegröße sein müsste; dies ist jedoch nicht der Fall, da die größte Eingabezahl exponentiell in der Anzahl der Bits sein kann.

Algorithmus 9 Pseudopolynomieller Algorithmus für SUBSET SUM.

```

function SUBSET SUM( $(a_i)_{i \in \{1..n\}}$ ,  $max$ )
  Let  $reachable[0..max] \leftarrow false$ .
   $reachable[0] \leftarrow true$ .
  for  $i \in \{1 .. n\}$  do
    for  $j \in \{max - a_i .. 0\}$  do  $\triangleright$  iterate backwards to avoid using already changed values
      if  $reachable[j]$  then  $reachable[j + a_i] \leftarrow true$ .
  return  $reachable$ .

```

Hierbei ist die Frage, ob man aus einer gegebenen Menge von n natürlichen Zahlen $(a_i)_{i \in \{1..n\}}$ eine Teilmenge auswählen kann, deren Summe gleich einer anderen gegebenen Zahl ist. Wir möchten effektiv dieses Problem für jeden möglichen Index lösen, unsere a_i sind die Anzahlen der Spalten in den Kindknoten.

Die Idee der Lösung ist, dass wir nacheinander jeweils ein a_i „dazunehmen“ und eine Fallunterscheidung durchführen, ob wir das a_i als Summanden für unsere Summe wählen oder nicht. Welche Zahlen erreichbar sind, speichern wir in einem Bitset (Größe: Summe aller Zahlen. Bei uns ist das maximal die Anzahl der enthaltenen Spalten). Fügen wir nun eine Zahl a_i zur Menge der nutzbaren Zahlen hinzu, so ändert sich die Menge der erreichbaren Zahlen nicht, wenn wir den neuen Wert nicht verwenden. Verwenden wir ihn jedoch, wird zusätzlich zu jeder erreichbaren Summe s auch $s + a_i$ erreichbar. Als Basisfall verwenden wir, dass 0 immer erreichbar ist. Wir erhalten also Algorithmus 9.

Bei Betrachten der Spalten-Kindknoten müssen wir beachten, dass die Spalten nicht notwendigerweise direkt nacheinander kommen müssen. Hier machen wir statt der Fallunterscheidung, ob wir den Wert nehmen, die Unterscheidung, wie viele der Spalten wir nehmen.

Wollen wir nun noch wissen, wie wir eine Summe erreichen können, speichern wir bei jeder möglichen Summe *ein* a_i ¹⁷, mit dem wir als letzten Summanden diese Summe erreicht haben. Möchten wir nun wissen, wie wir die Summe s erreicht haben, und haben wir a_i als möglichen letzten Summanden gespeichert, so wissen wir, dass a_i als Summand in der Summe vorkommen kann. Wir speichern also, dass a_i verwendet wurde, und fahren bei $s - a_i$ fort. Hier speichern wir erneut einen Summanden usw., bis wir bei 0 angekommen sind (Dies nennt sich „backtracking“).

Um zu berechnen, ob die Präfixlängen auch eindeutig sind, müssen wir nun noch berechnen, ob wir Zahlen mehrmals erreichen können. Wir gehen davon aus, dass Permutationen der einzelnen zu wählenden „Blöcke“ schon behandelt wurden (ONLY0, ONLY1 wie oben beschrieben), wir müssen also nur berechnen, ob es mehrere Möglichkeiten gibt, eine Präfixlänge zu erreichen. Hierfür speichern wir statt der Information, ob ein Wert erreichbar ist, entweder „nicht erreichbar“, „eindeutig erreichbar“ oder „mehrdeutig erreichbar“. Statt einfach den nächsten Wert $s + a_i$ auf „true“ zu setzen, kombinieren wir also die Information bei s mit der von $s + a_i$.

Die Laufzeit bleibt auch mit diesen Veränderungen $\mathcal{O}(n \cdot max) \subseteq \mathcal{O}(c \cdot max)$ (Anzahl der ONLY0-Kindknoten \cdot Anzahl der Spalten innerhalb dieser Kindknoten).

Wir müssen nur noch potenziell die möglichen Startindizes der BOTH-Kindknoten berücksichtigen. Dafür addieren wir auf alle möglichen Präfixlängen einfach diese Startindizes; der einzige

¹⁷Haben wir das erste Mal eine Möglichkeit für eine Summe s gefunden, überschreiben wir diesen Summanden nicht mehr. Ersetzen wir dies nämlich durch ein erst später betrachtetes a_i , so würde der Algorithmus für $s + a_i$ ausgeben, dass wir zweimal a_i verwendet haben (da wir die eigentliche Lösung überschrieben haben).

Unterschied zu oben ist, dass einer der Startindizes gewählt werden muss. Dies benötigt eine Laufzeit von $\mathcal{O}(\text{Anzahl Präfixlängen} \cdot \text{mögliche Startindizes}) \subseteq \mathcal{O}(\max \cdot c)$, was quadratisch in c ist.

Uns fehlt nur noch die Berechnung für Q-Nodes. Hier gibt es genau zwei mögliche Präfixlängen, und zwar die Anzahl der Spalten vor dem BOTH-Knoten (oder dem ersten ONLY1-Knoten), und die danach (Umdrehen der Q-Node). Die restliche Berechnung erfolgt wie bei P-Nodes.

Da die Berechnung bei einer PQ-Node $\mathcal{O}(c \cdot \max)$ Zeit benötigt, und \max jeweils die maximale Präfixlänge, also die Summe der Längen der ONLY0-Kindknoten ist; und jede dieser Knoten daher nur einmal betrachtet wird, liegt die Gesamtlaufzeit in $\mathcal{O}(c^2)$.

Wir berechnen also für den Wurzelknoten des PQ-Trees, an welchen Indizes die erste Zeile beginnen kann, und überprüfen, ob dies eben am benötigten Index möglich ist. Zudem haben wir direkt mitberechnet, ob es mehrere Möglichkeiten gibt (dies ist der Fall, wenn es irgendeine ONLY0- oder ONLY1-Knoten gibt, oder der berechnete Wert dies angibt). Wir können das gesamte Problem also in der Laufzeit von $\mathcal{O}(r \cdot c + c^2)$ lösen.

Eine alternative (etwas schnellere¹⁸) Implementierung von PQ-Trees wird in folgendem Paper beschrieben: Kellogg S. Booth, George S. Lueker, Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, Journal of Computer and System Sciences, Volume 13, Issue 3, 1976, Pages 335-379, ISSN 0022-0000, [https://doi.org/10.1016/S0022-0000\(76\)80045-1](https://doi.org/10.1016/S0022-0000(76)80045-1). (<https://www.sciencedirect.com/science/article/pii/S0022000076800451>)

3.2 Teilaufgabe a): Graphen

Anmerkung 1: Dieser Algorithmus geht an mehreren Stellen davon aus, dass alle Zeilen paarweise verschieden sind. Daher müssen vor Aufrufen des Algorithmus doppelte Zeilen entfernt werden. Da diese keine weiteren Informationen über die Reihenfolge der Spalten liefern, ist dies ohne Probleme möglich.

Anmerkung 2: Die in diesem Abschnitt dargestellten Tabellen wurden bereits zulässig permutiert, damit die Zusammenhänge zwischen den Zeilen besser erkennbar sind. Dies ist jedoch nicht nötig, um die Zusammenhänge zu bestimmen.

Bei diesem Ansatz versuchen wir, Zusammenhänge zwischen den einzelnen Zeilen zu analysieren und daraus zu folgern, in welcher Reihenfolge die Zeilenblöcke (1sen einer Zeile) beginnen und enden.

Es lässt sich feststellen, dass zwei Zeilen folgende Zusammenhänge haben können:

- Die Zeilen „überlappen“ sich (nicht-leere Schnittmenge, diese ist aber eine echte Teilmenge beider Zeilen). Dies ist der Fall, wenn es sowohl Spalten mit 0/1, 1/1, und 1/0 in diesen Zeilen gibt.

```
1 1 1 1 1 0 0 0 0 0
0 0 1 1 1 1 1 1 0 0
```

- Eine Zeile ist in einer anderen „enthalten“. Dies ist der Fall, wenn es Spalten mit 1/0 und 1/1 in diesen Zeilen gibt, aber nicht 0/1.

¹⁸Der dort beschriebene Algorithmus benötigt $\mathcal{O}(r + c + f)$ Zeit, wobei f die Anzahl der 1en ist. Bei uns ist die Laufzeit jedoch aufgrund des Eingabeformats schon durch $\Omega(r \cdot c)$ beschränkt.

```
0011111100
0000110000
```

- Die Zeilen sind „disjunkt“ / „unabhängig“. Hier gibt es keine Spalten 1/1.

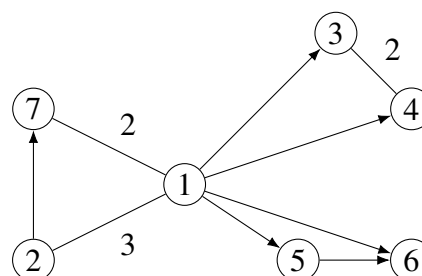
```
1111000000
0000001111
```

Wir erstellen nun einen „Zeilengraphen“. Dieser enthält einen Knoten für jede Zeile der Matrix, eine ungerichtete Kante zwischen sich überlappenden Zeilen, und eine gerichtete Kante von einer Zeile zu einer zweiten, wenn die erste die zweite enthält.

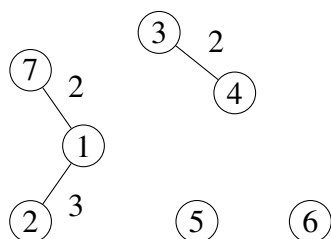
Als Teilgraph des Zeilengraphen betrachten wir nun den „Überlappungsgraphen“, welcher nur die ungerichteten Kanten des Zeilengraphen enthält. Hier gewichten wir zusätzlich die Kanten: Als Gewicht einer Kante verwenden wir die Anzahl der Spalten, die in beiden Zeilen enthalten sind. In diesem Graphen bestimmen wir die Zusammenhangskomponenten (mittels Tiefensuche (DFS) oder Breitensuche (BFS)).

```
1: 11111111111100000
2: 00000000001111110
3: 01110000000000000
4: 00111000000000000
5: 00000111100000000
6: 00000011000000000
7: 00000000001111000
```

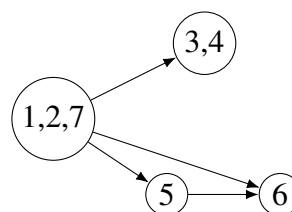
(a) Die Tabelle



(b) Der Zeilengraph



(c) Der Überlappungsgraph



(d) Der Komponentengraph

Es lässt sich feststellen, dass in einer Zusammenhangskomponente durch die Überlappungen (und nicht-Überlappungen) sowie deren Gewichtungen eindeutig bestimmt ist, in welcher Reihenfolge die einzelnen Zeilenblöcke (= 1sen der Zeile) beginnen und enden. Diese „Anfänge“ und „Enden“ der Zeilenblöcke bezeichnen wir als „Events“, ihre Reihenfolge als Eventreihenfolge. Die einzige „Operation“, mit der wir diese Eventreihenfolge verändern können, ist, indem wir die gesamte Eventreihenfolge (und damit einen Abschnitt der Tabelle) umdrehen¹⁹ (vgl. Q-Node im PQ-Tree).

Um eine solche Eventreihenfolge zu erstellen, müssen wir innerhalb der Komponente die Knoten so geordnet betrachten, dass neu zur Eventreihenfolge hinzugefügte Knoten adjazent (benachbart) zu anderen schon hinzugefügten Knoten sind. Für die ersten beiden Zeilen ist frei wählbar, in welcher Reihenfolge sie sich überlappen (welcher Zeilenblock zuerst beginnt), da die gesamte Eventreihenfolge umkehrbar ist. Für jeden neuen Knoten betrachtet man einen

¹⁹und Start- und End-Events vertauschen, damit Blöcke immer erst beginnen und danach enden

beliebigen adjazenten Knoten und berechnet mithilfe der Informationen, welche anderen Zeilen die Zeile noch überlappen muss bzw. nicht überlappen darf. Damit kann man eindeutig (in Linearzeit) herausfinden, in welcher Reihenfolge die Anfänge und Enden der Zeilenblöcke stattfinden.

Der einzige nicht triviale Fall ist der, dass sich drei Zeilen paarweise überlappen (s. Abbildung 3.9). Hier kann man feststellen, dass immer zwei der Zeilen „zusammen“ die dritte enthalten; für die Reihenfolge ist relevant zu wissen, welche dies ist. Dies ist der Fall genau dann, wenn die Anzahl der gemeinsamen Spalten der ersten beiden Zahlen kleiner ist als sowohl die Anzahl der gemeinsamen Spalten der ersten und dritten als auch der zweiten und dritten Zeile (s. Fulkerson und Gross (1965)). Mit den Gewichtungen (Anzahl der gemeinsamen Spalten) lässt sich diese Bedingung leicht überprüfen.

```

00111111110000000
0000001111111100
0000011111111000

```

Abbildung 3.9: Drei sich paarweise überlappende Zeilen.

Den Bereich zwischen zwei benachbarten Events bezeichnen wir als „Segment“. In diesen Segmenten ist eindeutig bestimmt, welche Werte in den im Segment enthaltenen Spalten stehen (zumindest bezüglich der Zeilen, die in der Komponente enthalten sind). Für jede Belegung dieser Zeilen gibt es ebenso nur ein mögliches Segment, weshalb sich alle Spalten in eines der Segmente einordnen lassen²⁰.

Fassen wir im Zeilengraphen alle Knoten einer Komponente zu einem Knoten zusammen, erhalten wir den „Komponentengraphen“. Es lässt sich beobachten, dass dieser azyklisch und transitiv²¹ ist (Beweis s. Fulkerson und Gross (1965)). Ebenso hat jede Komponente nur einen „direkten“ Vorgänger²². In diesem ist die gesamte Komponente enthalten, genauer gesagt, sogar in einem einzigen Segment des direkten Vorgängers (dieses lässt sich aus den Zeilen des Vorgängers bestimmen, in welchen die gesamte Komponente enthalten ist).

Wir können weiterhin beobachten, dass sich jede Spalte genau einem Segment in irgendeiner Komponente zuordnen lässt. Insgesamt erhalten wir also eine Menge „erster“ Komponenten (sie besitzen keinen Vorgänger), diese bestehen aus verschiedenen Segmenten. Jedes dieser Segmente kann eine Menge an Spalten und weiterer Komponenten enthalten (die Reihenfolge dieser ist beliebig). In Abbildung 3.8d ist (1,2,7) die einzige „erste“ Komponente, der direkte Vorgänger von z.B. (6) ist (5).

Es ist auffällig, dass diese Struktur sehr den PQ-Trees des ersten Ansatzes ähnelt. Die Komponenten entsprechen den Q-Nodes, die Segmente (und die Menge der ersten Komponenten) den P-Nodes.

Dieser Algorithmus benötigt $\mathcal{O}(r^2 \cdot c)$ Zeit, allein schon zum Berechnen des gesamten Zeilengraphen mit Gewichtungen.

Die weiteren Teilaufgaben b) und c) werden somit analog zu PQ-Trees gelöst.

²⁰Das stimmt nicht so ganz für die Belegung mit nur 0en, wenn alle Zeilen der Komponente in einer anderen Zeile enthalten sind; dazu kommen wir aber später noch.

²¹Gibt es eine Kante $i \rightarrow j$ und eine Kante $j \rightarrow k$, dann gibt es auch eine Kante $i \rightarrow k$

²²Der direkte Vorgänger ist ein Vorgänger, welcher in allen anderen Vorgängern enthalten ist.

Eine Beschreibung dieses Verfahrens findet man auch in: D. R. Fulkerson, O. A. Gross "Incidence matrices and interval graphs," Pacific Journal of Mathematics, Pacific J. Math. 15(3), 835-855, (1965), (<https://projecteuclid.org/journals/pacific-journal-of-mathematics/volume-15/issue-3/Incidence-matrices-and-interval-graphs/pjm/1102995572.full>)

Dieser Algorithmus benötigt $\mathcal{O}(r^2 \cdot c)$ Zeit, da er den gesamten Zeilengraphen explizit berechnet. Es ist tatsächlich möglich, dies auf $\mathcal{O}(r \cdot c)$ zu verbessern, wie im folgenden Paper beschrieben wird: Hsu, WL. (1992). A simple test for the consecutive ones property. In: Ibaraki, T., Inagaki, Y., Iwama, K., Nishizeki, T., Yamashita, M. (eds) Algorithms and Computation. ISAAC 1992. Lecture Notes in Computer Science, vol 650. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-56279-6_98, (https://link.springer.com/chapter/10.1007/3-540-56279-6_98)

3.3 Teilaufgabe d)

NP-Vollständigkeit

Anmerkung: Dieser Abschnitt ist relativ theoretisch und für eine vollständige Lösung der Aufgabe nicht notwendig.

Das Entscheidungsproblem TR?, ob eine Tabelle mit Fragezeichen zulässig permutierbar ist, ist NP-vollständig.²³

Es ist zunächst offensichtlich, dass $TR? \in NP$ gilt, da für eine gegebene Permutation in Polynomialzeit überprüfbar ist, ob die permutierte Tabelle zulässig ist.²⁴

Nun reicht es aus, dass wir noch eine Karp-Reduktion von einem anderen NP-vollständigen Problem auf TR? angeben.²⁵ Dazu verwenden wir das Entscheidungsproblem BETWEENNESS²⁶.

Gegeben ein positives $n \in \mathbb{N}$ und k Tripel $((a_i, b_i, c_i))_{i \in \{1..k\}}$ von Zahlen aus $\{1 .. n\}$, gibt es eine Permutation von $\{1 .. n\}$, in der für jedes Tripel (a_i, b_i, c_i) b_i zwischen a_i und c_i steht?²⁷

Eine Instanz dieses Problems formulieren wir in eine Tabelle für TR? um, indem wir jeder Zahl in $\{1 .. n\}$ eine Spalte zuordnen (n Spalten) und für jedes Tripel zwei Zeilen verwenden. Die erste Zeile hat eine 1 in den Spalten a_i und b_i und eine 0 in der Spalte c_i , die zweite Zeile eine 0 in der Spalte a_i und 1en in den Spalten b_i und c_i . Alle restlichen Einträge sind Fragezeichen.

	a_i		b_i		c_i	
?...?	1	?...?	1	?...?	0	?...?
?...?	0	?...?	1	?...?	1	?...?

Es lässt sich leicht feststellen, dass in jeder validen Permutation der Tabelle die Spalte b_i zwischen a_i und b_i stehen muss. Wegen der Fragezeichen entspricht zudem jede valide Permutation

²³siehe auch <https://tryalgo.org/en/data%20structures/2024/01/03/pc-trees/#a-generalization-to-partially-defined-matrices>

²⁴Für eine etwas ausführlichere und formellere Version siehe auch Abschnitt 3.4

²⁵Für den formalen Hintergrund siehe Abschnitt 3.4

²⁶siehe auch https://en.wikipedia.org/wiki/Betweenness_problem

²⁷Wir können für die Laufzeitanalyse davon ausgehen, dass n polynomiell in der Eingabegröße beschränkt ist. Wir können nämlich annehmen, dass jede Zahl in $\{1 .. n\}$ in mindestens einem der Tripel vorkommt. Ansonsten können wir diese Zahl ohne Probleme aus der Menge der betrachteten Zahlen entfernen.

		0 0 1 1 1 ? 0	
		1 1 1 1 1 0 0	
0 0 1 1 ? 0 0			
0 1 ? ? 1 1 0	(a) Ist die erste Zeile in der zweiten enthalten?	(b) Überlappen sich die Zeilen (? muss 1 sein) oder ist die erste in der zweiten enthalten (? muss 0 sein)?	

Abbildung 3.10: Problematische Situationen mit Zusammenhängen zwischen den Zeilen.

der BETWEENNESS-Instanz einer validen Permutation der Tabelle. Also ist die Tabelle valide permutierbar genau dann, wenn die BETWEENNESS-Instanz lösbar ist.

Zudem lässt sich beobachten, dass die Reduktion in Polynomialzeit berechenbar ist, es handelt sich also um eine Karp-Reduktion $BETWEENNESS \leq_p TR?$.

Da BETWEENNESS NP-vollständig ist, ist demnach auch $TR?$ NP-vollständig. q.e.d.

Potenzielle Schwierigkeiten

In diesem Abschnitt werden zunächst Probleme aufgezeigt, die bei einer Verallgemeinerung der Ansätze für die ersten Teilaufgaben auftreten können.

Bei PQ-Trees ist das offensichtlichste Problem zunächst, dass diese bei einer REDUCE-Operation erfordern, dass man die Spalten in die Kategorien „0“ und „1“ einteilen kann. Dies ist bei „?“ offensichtlich (i.d.R.) unmöglich, oder zumindest schwierig, weshalb man den Algorithmus nicht so weiterverwenden kann. Eine potenzielle Idee wäre, in einem solchen Fall die Spalte in zwei Spalten aufzuteilen (mit jeweils unterschiedlichem Wert), hier könnte man sich jedoch später evtl. entscheiden müssen, welche Spalte man behält, da manche in Kombination mit anderen möglichen Spalten in Konflikt stehen. Man könnte auch versuchen, diese Aufteilung erst möglichst spät vorzunehmen, es kann jedoch erzwungen werden, dass Spalten in verschiedenen Kindknoten sein könnten. Außerdem müsste man bei der Bestimmung der Möglichkeiten für die erste Zeile berücksichtigen, dass bestimmte Spalten in mehreren der Kindknoten sein können, und die mögliche Größe eines zu wählenden Knotens von den schon gewählten Knoten abhängt. Zudem ist nicht einfach eindeutig bestimmbar, welche Spalten der ersten Zeile die 1sen enthalten.

Bei dem Ansatz mit Graphen ist das Problem, dass sich die Zusammenhänge zwischen den Zeilen nicht immer eindeutig bestimmen lassen (s. Abbildung 3.10).

Im Folgenden werde ich noch kurz zwei Ansätze vorstellen, die man versuchen könnte (natürlich neben vielen weiteren (optimierten) Brute-Force-Lösungen).

Greedy (Heuristik)

Anmerkung: Der folgenden Algorithmus kann für alle der lösbaren Beispieleingaben (bis auf `konfetti00.txt`) in einer angemessenen Zeit eine valide Permutation finden. Es ist jedoch auch relativ leicht, Beispiele zu finden / zu konstruieren²⁸, für die der Algorithmus eine existierende Lösung nicht findet; er ist also eigentlich nicht korrekt.

²⁸durch Duplizieren von Zeilen / Spalten

Man kann auch versuchen, die Aufgabe mit Heuristiken (hier zur Wahl der nächsten Spalte) zu lösen. Der hier vorgestellte Ansatz wählt so zu jedem Zeitpunkt greedy die nächste Spalte der Permutation aus.

Als aktuellen Zustand bezeichne man die Belegung von 0en/1sen der zuletzt gewählten Spalte. Hat diese in einer Zeile ein '?', so wird die Belegung der letzten Spalte, welche in dieser Zeile kein '?' hat, übernommen. Der initiale Zustand enthält in jeder Zeile eine 0. Als restliche Spalten bezeichnen wir die Spalten, welche noch nicht gewählt wurden.

Um die Menge der möglichen nächsten Spalten einzuschränken, werden folgende Regeln verwendet:

- Eine Spalte, welche eine 1 des aktuellen Zustands in eine 0 ändern (den Zeilenblock beenden) würde, darf nicht gewählt werden, wenn es noch eine restliche Spalte gibt, welche eine 1 in dieser Zeile hat.
- Eine Spalte, welche in zwei Zeilen die Werte $1/1^{29}$ hat, darf nicht gewählt werden, wenn es noch restliche Spalten mit den Werten $0/1$ und eine mit den Werten $1/0$ gibt. Die Spalte mit $1/1$ muss nämlich in der Permutation zwischen denen mit $0/1$ und $1/0$ stehen.
- Alle anderen Spalten kann der Algorithmus als nächste wählen. *Achtung: Dies bedeutet nicht, dass es auch eine valide Permutation gibt, in der diese Spalten als nächstes kommen!*

Aus allen möglichen nächsten Spalten wird eine nach den folgenden Regeln gewählt.

1. Die gewählte Spalte sollte im aktuellen Zustand möglichst wenige Änderungen von 0 nach 1 verursachen (möglichst wenig neue Blöcke beginnen).
2. Bei Gleichstand sollten möglichst viele 1sen im aktuellen Zustand beibehalten werden.

Speichert man einige Werte (z.B. wie viele restliche Spalten eine 1 in einer bestimmten Zeile haben, sowie wie viele restliche Spalten die Werte $1/0$ in einem Zeilenpaar haben), so lässt sich dieser Algorithmus leicht in $\mathcal{O}(r^2c^2)$ implementieren (es geht sicher auch noch schneller), was auch für die größten Beispielergebnisse in angemessener Zeit läuft.

SAT-Solver

Ein weiterer Ansatz, die Aufgabe zu lösen, könnte darin bestehen, eine Instanz als 3-SAT-Formel (in konjunktiver Normalform) zu formulieren. Diese besteht aus einer Menge an Klauseln (hier Disjunktion von maximal 3 booleschen Variablen), welche alle erfüllt werden müssen. Ein SAT-Solver kann dann verwendet werden, um eine erfüllende Belegung der Variablen zu finden (oder zu bestimmen, dass es keine solche gibt).

Hierbei verwenden wir für jedes Paar von Spalten mit den Indizes i, j eine boolesche Variable $prev_{i,j}$, welche genau dann *true* sein soll, wenn $\pi(i) < \pi(j)$ ist, also Spalte i in der Permutation vor Spalte j kommt.

Da entweder Spalte i vor Spalte j kommt oder umgekehrt, muss gelten $less_{i,j} = \neg less_{j,i}$. Dies lässt sich über die Klauseln $less_{i,j} \vee less_{j,i}$ und $\neg less_{i,j} \vee \neg less_{j,i}$ erzwingen.

Nun müssen wir die Zeilenbedingungen codieren. Dafür können wir beobachten, dass die Zeilenbedingung einer Zeile zu folgender Aussage äquivalent ist: Für jede 0 in einer Zeile gilt, dass

²⁹Wert der ersten betrachteten Zeile / Wert der zweiten betrachteten Zeile

ihre Spalte entweder vor allen Spalten mit einer 1 in dieser Zeile oder nach all diesen Spalten kommt. Fragezeichen müssen hier nicht betrachtet werden, da für diese je nach Position ein passender Wert wählbar ist.

Dies wiederum ist damit gleichbedeutend, dass für jeden Eintrag $col_i[j] = 0$ alle Variablen $prev_{i,i'}$ mit einem Spaltenindex i' mit $col_{i'}[j] = 1$ äquivalent sind. Diese Äquivalenzen müssen nicht paarweise codiert werden, es genügt, eine der Variablen als „Referenzvariable“ zu wählen und für alle restlichen Variablen eine Äquivalenz zu dieser Referenzvariable zu codieren³⁰ (ausreichend, da Äquivalenz transitiv ist).

Zuletzt müssen wir noch sicherstellen, dass $less_{i,j}$ eine transitive Relation codiert. Gibt es also Spalten i, j, k , sodass Spalte i vor Spalte j kommt ($less_{i,j} = true$) und sodass Spalte j vor Spalte k kommt ($less_{j,k} = true$), dann muss auch Spalte i vor Spalte k kommen ($less_{i,k} = true$). Da dies durch unsere Klauseln bisher noch nicht codiert wurde, müssen wir diese Bedingung $less_{i,j} \wedge less_{j,k} \Rightarrow less_{i,k} = \neg less_{i,j} \vee \neg less_{j,k} \vee less_{i,k}$ noch für jedes Spaltentripel (i, j, k) einfügen.³¹

Nachdem wir mithilfe eines SAT-Solvers eine mögliche Belegung der Variablen bestimmt haben, müssen wir nur noch die Reihenfolge der Spalten rekonstruieren. Hierfür wählen wir einfach iterativ diejenige Spalte, welche (nach Belegung der $less$ -Variablen) vor allen anderen verbleibenden Spalten kommen muss.

Potenzielle Optimierungen Die Größe unserer SAT-Instanz lässt sich noch deutlich verringern. So benötigen wir nur eine der Variablen $less_{i,j}$ und $less_{j,i}$, die jeweils andere können wir immer durch eine Negation der ersteren ersetzen.

Weiterhin lässt sich beobachten, dass wir ansonsten neben den Transitivitätsklauseln nur Äquivalenzen vorliegen haben. Daher kann man vor dem Hinzufügen der Klauseln zur Instanz die Mengen der äquivalenten Klauseln bestimmen und diese durch jeweils eine einzige Variable ersetzen.

3.4 Teilaufgabe e)

NP-Vollständigkeit

Anmerkung 1: Dieser Abschnitt ist vergleichsweise theoretisch und ist für das Verstehen sowie eine vollständige Lösung der Aufgabe nicht notwendig, jedoch ist er sehr interessant.

Anmerkung 2: Wenn man die NP-Vollständigkeit von d) bereits gezeigt hat, ist die Karp-Reduktion von d) auf e) trivial³², da e) eine Verallgemeinerung von d) ist.

Da NP-Vollständigkeit nur für Entscheidungsprobleme definiert ist, müssen wir zunächst ein passendes Entscheidungsproblem definieren:

TABLE REORDER CHANGES (TRC). Gegeben eine Tabelle bestehend aus 0, 1 und ? und $k \in \mathbb{N}$, ist es möglich, maximal k Einträge so zu ändern, dass die Tabelle valide permutierbar ist.

³⁰Um Äquivalenzen zu codieren, verwendet man folgende Umformung (x, y sind boolesche Variablen): $x \Leftrightarrow y = (x \Rightarrow y) \wedge (y \Rightarrow x) = (\neg x \vee y) \wedge (\neg y \vee x)$

³¹Diese Transitivitätsklauseln sind die einzigen Klauseln, welche aus 3 Literalen bestehen. Ohne diese hätten wir also eine 2-SAT-Instanz, welche in Linearzeit lösbar wäre. Da wir diese Klauseln jedoch benötigen, liegt eine 3-SAT-Instanz vor; 3-SAT ist ein NP-vollständiges Problem.

³²übernehme die Tabelle unverändert, maximal 0 Veränderungen

Um zu zeigen, dass TRC NP-vollständig ist, müssen zwei Dinge gezeigt werden:

1. $\text{TRC} \in \text{NP}$: Hierzu ist zu zeigen, dass es einen Verifizierer gibt³³, der, gegeben eine Tabelle und ein sog. Zertifikat, nachweist, dass die Tabelle zulässig permutierbar ist. Hierfür kann man als Zertifikat einfach eine Permutation $\pi : \{1 \dots c\} \rightarrow \{1 \dots c\}$ nehmen, sodass $T' = (\text{col}_{\pi(i)})_{i \in \{1 \dots c\}}$ zulässig ist.

Es lässt sich relativ einfach in Polynomialzeit zeigen, dass π eine Permutation ist (überprüfe, dass es für jedes $i' \in \{1 \dots c\}$ genau ein $i \in \{1 \dots c\}$ gibt, sodass $\pi(i) = i'$). Es muss nur noch überprüft werden, dass T' zulässig ist, was, wie oben beschrieben, durch Wahl von Werten für die „?“ geschehen kann. Die Überprüfung, dass in der entstehenden Tabelle alle Zeilen zulässig sind, kann auch offensichtlich in Polynomialzeit erfolgen.

Also gibt es einen Verifizierer für TRC, es gilt $\text{TRC} \in \text{NP}$, q.e.d.

2. TRC ist NP-schwer, d.h. $\forall L \in \text{NP} : L \leq_p \text{TRC}$, TRC ist also mindestens so schwer wie jedes andere Problem in NP. Da \leq_p transitiv ist³⁴, reicht es, eine Karp-Reduktion $X \leq_p \text{TRC}$ anzugeben für ein (schon gezeigt) NP-vollständiges Problem X (für dieses wurde ja schon gezeigt, dass $L \leq_p X \quad \forall L \in \text{NP}$).

Im Folgenden wird eine Karp-Reduktion $\text{HAMILTONIAN PATH} \leq_p \text{TRC}$ angegeben³⁵.

Sei also $G = (V, E)$ eine Instanz von HAMILTONIAN PATH (HP). Seien $n := |V|, m := |E|$. Daraus konstruiere man nun eine Tabelle T mit n Spalten und m Zeilen.

Für jeden Knoten wird genau eine Spalte erstellt, die Reihenfolge der Spalten entspricht dann einer Permutation der Knoten. Kann man erzwingen, dass für jeweils zwei Spalten, die aufeinanderfolgen, eine Kante zwischen den entsprechenden Knoten existiert, so ist diese Permutation der Knoten offensichtlich ein Hamiltonpath.

Dafür konstruiert man für jede Kante $uv \in E$ eine Zeile in T , welche in genau den Spalten von u und v eine „1“ hat, in allen anderen eine „0“. Damit folgt trivialerweise, dass eine Zeile valide ist genau dann, wenn die beiden Knoten der Kante in dem Pfad aufeinanderfolgen. Damit die Permutation einem HP entspricht, muss dies für alle $n - 1$ Kanten des Pfades gelten. In den anderen Zeilen muss eine „Verletzung“ der Zeilenzulässigkeit ermöglicht werden, es ist zu beobachten, dass dies durch genau eine Änderung eines Eintrags möglich ist (ändere eine der 1sen zu einer 0).

Also lässt man $m - (n - 1) = m - n + 1$ ³⁶ Änderungen in der Tabelle zu, gibt also insgesamt die TRC-Instanz $(T, k = m - n + 1)$ zurück.

- a) Laufzeit: Obiger Algorithmus ist offensichtlich in Laufzeit $\mathcal{O}(n \cdot m)$ implementierbar, die Laufzeit der Reduktion ist also polynomiell in der Eingabegröße.
- b) $G = (V, E) \in \text{HP} \Rightarrow f(G) \in \text{TRC}$: Sei p ein HP in G (existiert wg. $G \in \text{HP}$). Dann ordne die Spalten von T in der Reihenfolge der entsprechenden Knoten in p an. Offensichtlich sind alle Zeilen, die einer der $n - 1$ Kanten auf p entsprechen, zulässig. In den anderen $m - (n - 1) = m - n + 1 = k$ Zeilen, ersetze eine der 1sen durch eine

³³Die Existenz eines Verifizierers ist äquivalent zur klassischen Definition über die Existenz einer nichtdeterministischen Turing-Maschine.

³⁴Aus $A \leq_p B$ und $B \leq_p C$ folgt auch $A \leq_p C$.

³⁵ HAMILTONIAN PATH . Gegeben ein einfacher ungerichteter Graph $G = (V, E)$, gibt es einen Pfad in G , der jeden Knoten genau einmal besucht?

³⁶Ist $m - n + 1$ negativ, so gilt $m < n - 1$. Da jeder HP jedoch genau $n - 1$ Kanten hat, gibt es keinen HP. Gib also eine bekannt unlösbare Instanz für TRC zurück.

0, wodurch diese nur noch eine 1 enthalten und damit auch zulässig sind. Die gesamte Tabelle ist also mit k Änderungen zulässig permutierbar, es gilt $f(g) \in \text{TRC}$, q.e.d.

- c) $f(G) \in \text{TRC} \Rightarrow G \in \text{HP}$: Betrachte eine Lösung von $f(G)$. In der Tabelle T muss es genau $n - 1$ unveränderte Zeilen geben. Gäbe es mehr unveränderte Zeilen, lägen mehr als $n - 1$ Kanten auf dem HP mit n Knoten ($\frac{1}{2}$) oder es gäbe Kanten „doppelt“ ($\frac{1}{2}$, G ist einfacher Graph). Gäbe es weniger unveränderte Zeilen, wären mehr als $m - (n - 1) = k$ Zeilen verändert, jede veränderte Zeile benötigt jedoch mindestens eine der k möglichen Veränderungen ($\frac{1}{2}$). Da es genau $n - 1$ unveränderte, zulässige Zeilen gibt, liegen die entsprechende Kante auf dem potenziellen Pfad, der der Spaltenpermutation entspricht. Da es $n - 1$ solche Kanten gibt und keine Kanten doppelt, muss es sich um einen tatsächlichen Pfad handeln. Dieser ist, da jeder Knoten genau einmal besucht wird, ein HP. Also ist $G \in \text{HP}$, q.e.d.

Also ist $\text{HAMILTONIAN PATH} \leq_p \text{TRC}$, TRC ist NP-vollständig, q.e.d.

Lösung mit ILP

Da TRC NP-vollständig ist, gibt es unter der Annahme, dass $P \neq \text{NP}$ ³⁷, keinen Algorithmus, der das Entscheidungsproblem, und auch das Suchproblem, in Polynomialzeit löst. Eine gute Möglichkeit, NP-vollständige Probleme zu lösen, ist mithilfe von sog. ILP-Solvern³⁸. Dazu muss man sein Problem jedoch zunächst als ILP (Integer Linear Program) formulieren. Ein ILP besteht aus einer Menge an ganzzahligen Variablen sowie einer Menge von linearen Ungleichungen³⁹ über dieser. Zusätzlich wird noch ein linearer Term in den Variablen angegeben⁴⁰, welcher entweder zu minimieren oder zu maximieren ist (unter Einhaltung der linearen Ungleichungen). Der Vorteil davon ist, dass moderne ILP-Solver sehr gut optimiert sind, und das Problem oft schneller lösen als andere (zudem aufwändigere) exponentielle Lösungsansätze.

Im Folgenden wird eine *mögliche* Formulierung des Problems als ILP dargestellt, es gibt aber garantiert auch andere mögliche Formulierungen.

Zunächst muss man die Permutation der Spalten formulieren. Dazu kann man boolesche⁴¹ Variablen $(\text{perm}_{i,j})_{i,j \in \{1..c\}}$ definieren. Hierbei ist $\text{perm}_{i,j} = 1$ genau dann, wenn die Spalte col_i in der permutierten Tabelle zur j -ten Spalte wird, also $\pi(i) = j$ gilt. Es muss nun noch erzwungen werden, dass tatsächlich eine Permutation codiert wird. Dazu muss einerseits $\pi(i)$ eindeutig sein, d.h. für jedes $i \in \{1..c\}$ darf es nur ein $j \in \{1..c\}$ geben, sodass $\text{perm}_{i,j} = 1$ (Gleichung (3.1)). π muss weiterhin bijektiv sein, d.h. für jedes $j \in \{1..c\}$ darf es nur ein $i \in \{1..c\}$ geben, für dass $\pi(i) = j \Leftrightarrow \text{perm}_{i,j} = 1$ (Gleichung (3.2)). In einem ILP lässt sich dies folgendermaßen codieren:

³⁷Die Frage, ob die Komplexitätsklassen P und NP gleich sind ($P \subseteq NP$ ist trivial), zählt zu einem der größten ungelösten Probleme der Informatik. Wäre dies der Fall, würde dies dazu führen, dass u.a. alle NP-vollständigen Probleme schnell lösbar wären; auch basiert fast die gesamte Kryptographie auf der Annahme, dass $P \neq NP$.

³⁸ILP ist selbst ein NP-vollständiges Problem

³⁹Ungleichungen der Form $a_1x_1 + a_2x_2 + \dots + a_nx_n \leq b$, wobei x_1, \dots, x_n Variablen und a_1, \dots, a_n, b Konstanten sind.

⁴⁰Term der Form $a_1x_1 + a_2x_2 + \dots + a_nx_n$, wobei x_1, \dots, x_n Variablen und a_1, \dots, a_n Konstanten sind.

⁴¹d.h., die Variablen können nur die Werte 0 und 1 annehmen; in einem ILP lässt sich dies für eine Variable x mit den Bedingungen $0 \leq x \leq 1$ erzwingen, viele ILP-Solver bieten jedoch auch die Möglichkeit, boolesche Variablen direkt zu definieren.

$$\sum_{j=1}^c perm_{i,j} = 1 \quad \forall i \in \{1 \dots c\} \quad (3.1)$$

$$\sum_{i=1}^c perm_{i,j} = 1 \quad \forall j \in \{1 \dots c\} \quad (3.2)$$

Um die weiteren Bedingungen zu codieren, ist es nützlich, Variablen zu haben, welche den Zustand der permutierten Tabelle (bzw. aller Einträge dieser) enthalten. Dafür verwenden wir die Variablen $(col'_i[j])_{i \in \{1..c\}, j \in \{1..r\}}$ ⁴². Diese Variablen könnte man nun so codieren, dass sie ,0‘, ,1‘, ,?’ darstellen können, jedoch reicht es aus, ,0‘ und ,1‘ darzustellen, und bei einem ,?’ den ILP-Solver direkt die Belegung wählen zu lassen (so, dass die anderen Bedingungen erfüllt sind). Also sind auch diese Variablen Booleans, 0 (FALSE) codiert ,0‘ (×) und 1 (TRUE) ,1‘ (✓).

Bevor man nun erzwingt, dass die Werte der Spalten auch richtig (entsprechend $perm$) in die $col'_i[j]$ übernommen werden, muss man noch berücksichtigen, dass auch Veränderungen von Werten möglich sind. Dafür definiere man weitere boolesche Variablen $(change_i[j])_{i \in \{1..c\}, j \in \{1..r\}}$, welche 1 sein sollen genau dann, wenn das Feld $col'_i[j]$ verändert wurde (im Vergleich zu dem Eintrag, den es nach $perm$ und der ursprünglichen Tabelle haben müsste). Offensichtlich ist dann die Summe aller $change_i[j]$ zu minimieren, die Zielfunktion lautet also:

$$\min : \quad \sum_{i=1}^c \sum_{j=1}^r change_i[j] \quad (3.3)$$

Man betrachte jetzt einen Eintrag $col_i[j]$ in der ursprünglichen Tabelle ($i \in \{1 \dots c\}, j \in \{1 \dots r\}$). Zudem betrachtet man den Eintrag $col'_{i'}[j]$, in den der Wert potenziell übertragen werden muss ($i' \in \{1 \dots c\}$). Zunächst ist zu beobachten, dass für $perm_{i,i'} = 0$ keine Abhängigkeit zwischen $col_i[j]$ und $col'_{i'}[j]$ besteht. Da $col_i[j]$ bekannt ist (gegeben), mache eine Fallunterscheidung bzgl. $col_i[j]$:

1. $col_i[j] = 0$: Wenn $perm_{i,i'} = 1$ (also die Spalte i in der Permutation an Stelle i' steht), muss der Wert von $col_i[j]$ korrekt in den Wert von $col'_{i'}[j]$ übernommen werden, also muss entweder $col'_{i'}[j] = col_i[j] = 0$ sein oder $change_{i'}[j] = 1$ (Wert verändert)⁴³. Wenn $perm_{i,i'} = 0$ (also die Spalte i sich irgendwo anders befindet), ist der Wert von $col'_{i'}[j]$ „egal“, der von $change_{i'}[j]$ ebenfalls⁴⁴. Also muss gelten: $perm_{i,i'} = 0 \vee col'_{i'}[j] = 0 \vee change_{i'}[j] = 1 \Leftrightarrow (1 - perm_{i,i'}) + (1 - col'_{i'}[j]) + change_{i'}[j] \geq 1 \Leftrightarrow -perm_{i,i'} - col'_{i'}[j] + change_{i'}[j] \geq -1$.
2. $col_i[j] = 1$: Analog folgt, dass gelten muss: $-perm_{i,i'} + col'_{i'}[j] + change_{i'}[j] \geq 0$.
3. $col_i[j] = ?$: Wenn $perm_{i,i'} = 1$, gibt es trotzdem keine Einschränkungen für $col'_{i'}[j]$. Der Wert von $change_{i'}[j]$ ist in jedem Fall 0, dies muss aber nicht explizit codiert werden, da der ILP-Solver die Summe der $change$ -Variablen minimiert. Wenn $perm_{i,i'} = 0$, gibt es

⁴²Für eine tatsächliche Implementierung wären i, j beides Indizes; diese Schreibweise wurde auch für die anderen Teilaufgaben verwendet, weshalb sie auch jetzt verwendet wird.

⁴³Theoretisch können auch beide Werte 1 sein, jedoch würde der ILP-Solver dies nicht tun, da in diesem Fall das Setzen von $change_{i'}[j]$ auf 0 die Zielfunktion um 1 verringern würde.

⁴⁴Diese Werte werden durch eine andere Bedingung (andere i, i') bestimmt.

wie oben auch keine Einschränkungen. Also sind in diesem Fall keine Bedingungen für das ILP nötig.

Es ergeben sich also folgende weitere Bedingungen:

$$-perm_{i,i'} - col'_{i'}[j] + change_{i'}[j] \geq -1 \quad \forall i, i' \in \{1 \dots c\}, j \in \{1 \dots r\}, col_i[j] = 0 \quad (3.4)$$

$$-perm_{i,i'} + col'_{i'}[j] + change_{i'}[j] \geq 0 \quad \forall i, i' \in \{1 \dots c\}, j \in \{1 \dots r\}, col_i[j] = 1 \quad (3.5)$$

Nun kann man die eigentlichen Bedingungen an die permutierte Tabelle implementieren. Dies ist zunächst die Bedingungen der zusammenhängenden Zeilenblöcke. Hier gibt es mehrere Möglichkeiten, eine davon ist zu erzwingen, dass es in einer Zeile maximal einen „Wechsel“ von 0 nach 1 gibt (also einen Blockanfang). Dafür führe man neue boolesche Variablen ein, $(start_i[j])_{i \in \{1 \dots c\}, j \in \{1 \dots r\}}$, welche 1 sein sollen genau dann, wenn das entsprechende Feld $col'_i[j] = 1$ ist und $col'_{i-1}[j] = 0$. Also muss gelten $col'_{i-1}[j] = 1 \vee col'_i[j] = 0 \vee start_i[j] = 1$ ⁴⁵. Dies ist äquivalent zu $col'_{i-1}[j] - col'_i[j] + start_i[j] \geq 0$. Es ist zu beachten, dass diese Bedingung nur für $i \geq 2$ funktioniert, da es sonst kein „linkes“ Feld gibt. Für $i = 1$ muss einfach nur gelten, dass $start_i[j] = 1$, wenn $col'_i[j] = 1$. Dies ist formulierbar als $-col'_i[j] + start_i[j] \geq 0$. Nun muss natürlich noch die Anzahl der „starts“ je Zeile beschränkt werden, also gilt noch Gleichung (3.8). Insgesamt kommen folgende Bedingungen dazu:

$$-col'_1[j] + start_1[j] \geq 0 \quad \forall j \in \{1 \dots r\} \quad (3.6)$$

$$col'_{i-1}[j] - col'_i[j] + start_i[j] \geq 0 \quad \forall i \in \{2 \dots c\}, j \in \{1 \dots r\} \quad (3.7)$$

$$\sum_{i=1}^c start_i[j] \leq 1 \quad \forall j \in \{1 \dots r\} \quad (3.8)$$

Bisher lässt sich mit dem ILP berechnen, wie viele Werte verändert werden müssen, sodass es eine Lösung gibt, eine solche wird auch gefunden. Es ist jedoch noch zu ergänzen, dass auch die Fragen aus b), c) beantwortet werden.

Ist die erste Zeile angegeben, ist es trivial, die Werte von $col'_i[1]$ zu erzwingen.

Um zu überprüfen, ob es eine weitere Lösung gibt, kann man zuerst eine optimale Lösung finden, dann Bedingungen hinzufügen, die erzwingen, dass mindestens ein $col'_i[j]$ anders ist, und das ILP erneut lösen. Zudem kann man eine Bedingung hinzufügen, die die Zielfunktion mit dem schon gefundenen optimalen Wert beschränkt. Dann gibt es eine weitere optimale Lösung genau dann, wenn der ILP-Solver für das erweiterte ILP eine Lösung findet.

Seien $sol_i[j]$ die Werte der $col'_i[j]$ für eine optimale Lösung. Dann kann man neue boolesche Variablen einführen, $(diff_i[j])_{i \in \{1 \dots c\}, j \in \{1 \dots r\}}$, welche 1 sein sollen, wenn sich $col'_i[j]$ (der neuen Lösung) und $sol_i[j]$ unterscheiden. Zunächst muss man die Bedingung einführen, dass die Summe aller $diff$ -Werte größer als 0 sein muss. Da dies eine untere Schranke für die Summe ist, ist hier relevant, dass man die $diff$ -Werte auf 0 zwingt, wenn sie 0 sein müssten⁴⁶. Da die $sol_i[j]$ bekannt sind, kann man wieder eine Fallunterscheidung durchführen:

1. $sol_i[j] = 0$: Ist $col'_i[j] = 0$ (gleich dem Wert der ersten Lösung), muss auch $diff_i[j] = 0$

⁴⁵Auch hier kann $start_i[j] = 1$ gesetzt werden, auch wenn dies nicht nötig ist. Da die $start$ -Werte gleich 1 jedoch durch eine obere Schranke beschränkt sind und anderweitig nicht mehr verwendet werden, ist dies unproblematisch.

⁴⁶Bisher haben wir immer nur Werte auf 1 gezwungen, da wir obere Schranken hatten. Analog ist hier das Zwingen auf 1 irrelevant.

gelten. Also $col'_i[j] = 1 \vee diff_i[j] = 0 \Leftrightarrow col'_i[j] - diff_i[j] \geq 0$ ($\forall i \in \{1 \dots c\}, j \in \{1 \dots r\}$).

2. $sol_i[j] = 1$: Ist $col'_i[j] = 1$, muss $diff_i[j] = 0$ gelten. Also $col'_i[j] = 0 \vee diff_i[j] = 0 \Leftrightarrow -col'_i[j] - diff_i[j] \geq -1$ ($\forall i \in \{1 \dots c\}, j \in \{1 \dots r\}$).

Insgesamt werden $r \cdot c <perm> + r \cdot c <col> + r \cdot c <change> + r \cdot c <start> + r \cdot c <diff> = 5r \cdot c$ Variablen erstellt. Die Anzahl der Bedingungen ist maximal $c + c + r \cdot c^2 + r \cdot c + r + r \cdot c \in \mathcal{O}(r \cdot c^2)$. Die Laufzeit zum Erstellen des ILPs ist also polynomiell.

Anmerkung: Formell ist unser Algorithmus eine Cook-Reduktion⁴⁷ von dem Problem der Teilaufgabe e) auf ILP.

3.5 Beispiele

Übersicht über die Beispiele

Dateiname	rows	cols	? ⁴⁸	Erste Zeile ⁴⁹	lösbar	eindeutig
konfetti00.txt	4	4	ja	nein	ja	nein
konfetti01.txt	5	9	nein	nein	ja	nein
konfetti02.txt	5	9	nein	nein	nein (2 Änderungen)	-
konfetti03.txt	3	5	nein	nein	ja	ja
konfetti04.txt	7	17	nein	nein	ja	nein
konfetti05.txt	110	1020	nein	nein	ja	nein
konfetti06.txt	9	21	nein	ja	ja	ja
konfetti07.txt	9	21	nein	ja	nein (2 Änderungen)	-
konfetti08.txt	9	21	ja	ja	ja	ja
konfetti09.txt	7	13	ja	nein	ja	nein
konfetti10.txt	7	13	ja	nein	nein (3 Änderungen)	-
konfetti11.txt	107	1016	ja	nein	ja	nein
konfetti12.txt	109	1013	ja	ja	ja	nein
konfetti13.txt	8	16	ja	nein	nein (4 Änderungen)	-

Mögliche Anordnungen

Anmerkung: In den folgenden Anordnungen wurden Fragezeichen nicht ersetzt, damit die Permutationen besser erkennbar sind.

Es wird zunächst die Folge der Spaltenindizes in der Permutation angegeben (1-indiziert), anschließend die entstehende Tabelle. Die erste Zeile ist immer Annas Zeile.

Fragezeichen werden grün hervorgehoben. War keine Permutation möglich, wird eine Permutation mit einer minimalen Anzahl an Veränderungen angegeben. Veränderungen werden rot hervorgehoben.

Zu manchen Beispieleingaben fehlt aus Platzgründen die Ausgabe. Du kannst sie im Materialteil auf der Webseite finden.

⁴⁷Link YouTube

⁴⁸Die Eingabe enthält Fragezeichen

⁴⁹Die erste Zeile ist gegeben.

konfetti00

Order: [2, 3, 1, 4]

```

1 0 0 0
1 ? 1 0
0 0 1 1
0 1 1 0

```

konfetti01

Order: [1, 5, 8, 6, 7, 2, 4, 9, 3]

```

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

```

konfetti02

nur mit 2 Veränderungen lösbar

Order: [9, 5, 3, 7, 8, 1, 2, 4, 6]

```

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

```

konfetti03

Order: [1, 2, 3, 4, 5]

```

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

```

konfetti04

Order: [12, 8, 16, 1, 10, 6, 11, 4, 17, 3, 14, 5, 15, 7, 2, 9, 13]

```

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

```

konfetti05*wegen Größe hier ausgelassen***konfetti06**

erste Zeile gegeben, eindeutig lösbar

Order: [5, 16, 10, 9, 14, 15, 12, 2, 4, 19, 18, 21, 7, 8, 20, 1, 3, 6, 11, 13, 17]

```

00000000111111000000000
0000000111111111111111
0000000000000111111110
0011111111111100000000
0011111111111111100000
1111111111111111000000
0001111000000000000000
0000001111110000000000
0011111111000000000000

```

konfetti07

erste Zeile gegeben, nur mit 2 Veränderungen lösbar

Order: [10, 9, 17, 7, 5, 11, 6, 4, 19, 15, 13, 21, 3, 20, 16, 1, 2, 14, 18, 8, 12]

```

000001111111111111000
1111111111111000000000
0000001100000000000000
111111111111111111110
1100000000000000000000
0001111111000000000000
1111111110000000000000
00000001111111111111
000111111110000000000

```

konfetti08

erste Zeile gegeben, eindeutig lösbar

Order: [18, 21, 16, 10, 14, 3, 8, 2, 7, 5, 13, 17, 19, 20, 9, 15, 4, 11, 12, 1, 6]

```

00000000011111111100
000000111?111?111111
1111111111111111110
11111111111111111111
001111111111000000000
000000011111?111110?0
00011111111111111100
1111?111?11111110000
00000000111111111111

```

konfetti09

Order: [8, 6, 7, 13, 1, 9, 10, 2, 3, 11, 12, 4, 5]

```

0111?11111111
1111111?00??0
0100000000000
00111111111?1
0011111100?00
?000011111110
0111111100000

```

konfetti10

nur mit 3 Veränderungen lösbar

Order: [9, 3, 11, 1, 10, 6, 5, 8, 4, 13, 2, 7, 12]

```

0111111110000
1111111111111
00000011?1000
000000001110

```

```

0000?11000?00
1111?1111000
00000?111100

```

konfetti11

wegen Größe hier ausgelassen

konfetti12

erste Zeile gegeben

wegen Größe hier ausgelassen

konfetti13

nur mit 4 Veränderungen lösbar

Order: [14, 9, 1, 15, 4, 8, 10, 7, 3, 11, 12, 2, 5, 6, 16, 13]

```

01?111111?111000
01111111000?0000
0000111111111111
0001110000000000
0111?000000?0000
1111111111111110
0000000000001111
?00000?111100000

```

3.6 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

- (1) *Problem adäquat modelliert*: Es müssen alle Werte aus der Tabelle (\times , \checkmark und $?$) abgebildet werden können. Permutationen müssen erzeugt und zusammenhängende Zeitblöcke müssen berücksichtigt werden können.
- (2) *Verfahren nicht unnötig ineffizient*: Mehrfache Berechnungen des gleichen Wertes an verschiedenen Stellen sind nicht akzeptabel.
- (3) *Laufzeit des Verfahrens in Ordnung*:
 - Bei den Teilaufgaben a) bis c) sollte ein Verfahren mit polynomieller Laufzeit verwendet werden. Sonst sind 2 Punkte abzuziehen.
 - Bei den Teilaufgaben d) und e) werden 2 Punkte abgezogen, falls ein Brute-Force-Verfahren nicht sinnvoll verbessert wird.
- (4) *Speicherbedarf in Ordnung*: Die verwendete(n) Datenstruktur(en) sollen mit maximal polynomiell viel Speicher auskommen.
- (5) *Ergebnisse von Teilaufgaben a) bis c) korrekt*:
 Die vom Verfahren ausgegebene Anordnung der Spalten (falls vorhanden) muss zulässig sein. Wenn für „lösbar“ Beispiele (solche, für die es eine zulässige Anordnung ohne Änderung der Einträge gibt) nur Lösungen mit Änderungen angegeben werden, evtl. unter Zuhilfenahme des Verfahrens aus Teilaufgabe e), werden 2 Punkte abgezogen. Weiterhin muss das Verfahren korrekt ermitteln, ob die gefundene Anordnung nach der Definition der Einsendung eindeutig ist.
 Die folgende Tabelle gibt die Ergebnisse der Beispiellösung an, mit deren Definition der Eindeutigkeit.

Dateiname	rows	cols	Erste Zeile	lösbar	eindeutig
konfetti01.txt	5	9	nein	ja	nein
konfetti02.txt	5	9	nein	nein	-
konfetti03.txt	3	5	nein	ja	ja
konfetti04.txt	7	17	nein	ja	nein
konfetti05.txt	110	1020	nein	ja	nein
konfetti06.txt	9	21	ja	ja	ja
konfetti07.txt	9	21	ja	nein	-

- (6) *Ergebnisse von Teilaufgabe d) korrekt*: Falls
 - eine zulässige Anordnung mit einer korrekten Ersetzung der $?$
 - unter Berücksichtigung der bekannten 1. Zeile (falls vorhanden) ermittelt und
 - deren Eindeutigkeit (im Sinne der Einsendung) korrekt angegeben wird,

dann können bis zu 2 Pluspunkte gegeben werden. Wird für lösbare (s.o.) Beispiele nur eine Anordnung mit Änderungen angegeben (hier zählt nicht das Ersetzen von $?$), wird 1 Punkt abgezogen.

Dateiname	rows	cols	?	Erste Zeile	lösbar	eindeutig
1. Zeile unbekannt						
konfetti09.txt	7	13	ja	nein	ja	nein
konfetti10.txt	7	13	ja	nein	nein	-
konfetti11.txt	107	1016	ja	nein	ja	nein
1. Zeile bekannt						
konfetti08.txt	9	21	ja	ja	ja	ja
konfetti12.txt	109	1013	ja	ja	ja	nein

- (7) *Ergebnisse von Teilaufgabe e) korrekt und gut:* Für die Eingaben konfetti02.txt, konfetti07.txt, konfetti10.txt und konfetti13.txt muss eine zulässige Anordnung unter Angabe der Änderungen gefunden werden, ansonsten werden 4 Punkte abgezogen.

Die Anzahl der Änderungen muss *akzeptabel* sein, sonst werden bis zu 2 Punkte abgezogen. Ist die Anzahl der Änderungen *optimal*, können bis zu 2 Bonuspunkte gegeben werden.

Dateiname	optimal	akzeptabel
konfetti02.txt	2	≤ 2
konfetti07.txt	2	≤ 6
konfetti10.txt	3	≤ 4
konfetti13.txt	4	≤ 6

- (8) *Alle Teilaufgaben bearbeitet:* Wird Teilaufgabe d) nicht bearbeitet, werden 2 Punkte abgezogen. Wird Teilaufgabe e) nicht bearbeitet, werden 4 Punkte abgezogen.

2. Theoretische Analyse

- (1) *Verfahren / Qualität insgesamt gut begründet:* Es sollte entweder die Korrektheit des Verfahrens begründet werden oder erkannt werden, wenn das Verfahren nicht immer korrekt ist (es sollte jedoch zumindest auf den gegebenen Beispielen korrekt sein). Aus der Dokumentation muss hervorgehen, wie „eindeutig“ definiert wird. Wenn die Ergebnisse mehrerer Verfahren kommentarlos angegeben werden, können bis zu 2 Punkte abgezogen werden. Für eine Einsendung, die mehrere Verfahren vergleicht und Begründungen für deren Qualität liefert, können Bonuspunkte vergeben werden.
- (2) *Schwierigkeit der Teilaufgabe d) gezeigt:* Es sollte gezeigt werden, dass ein effizientes Verfahren für Teilaufgabe d) sehr schwierig zu finden ist, und sich entsprechend kritisch mit dem eigenen Lösungsansatz auseinander gesetzt werden (bei effizienten, aber nicht immer korrekten Verfahren). Falls die NP-Schwere der Teilaufgabe d) erkannt und korrekt begründet wird, etwa durch eine (informelle) Reduktion von einem bekannten NP-schweren Problem, können Pluspunkte vergeben werden.
- (3) *Gute Überlegungen zur Laufzeit des Verfahrens:* Die Laufzeiten der Verfahren müssen nicht zwingend formal, aber nachvollziehbar und korrekt charakterisiert werden. Mit Teilaufgabe c) ist eine Zeile der Anordnung (Annas Zeile) bekannt. Es wird erwartet, dass sich Gedanken darüber gemacht werden, was sich daraus für die Laufzeit des Verfahrens ergibt; falls nicht, wird 1 Punkt abgezogen.
- (4) *NP-Schwere der Teilaufgabe e) gezeigt:* Falls die NP-Schwere der Teilaufgabe e) erkannt und korrekt begründet wird, etwa durch eine (informelle) Reduktion von einem bekannten NP-schweren Problem, können Pluspunkte vergeben werden.

3. Dokumentation

- (3) *Vorgegebene Beispiele dokumentiert:* Die Ergebnisse für alle Beispiele `konfetti01.txt` bis `konfetti13.txt` müssen dokumentiert werden. Bei den großen Beispieleingaben `konfetti5.txt`, `konfetti11.txt` und `konfetti12.txt` ist ein Verweis auf externe Dateien in Ordnung.
- (4) *Weitere aussagekräftige Beispiele:* Werden für Teilaufgabe d) oder e) Beispiele entworfen, die die Grenzen des eigenen Verfahrens gut aufzeigen, können bis zu 2 Pluspunkte vergeben werden.
- (5) *Ergebnisse nachvollziehbar dargestellt:* Für jedes Beispiel muss, wenn möglich, eine Anordnung angegeben werden. Weiterhin muss angegeben werden, ob die Anordnung „eindeutig“ ist.

Für ? muss angegeben werden, wie diese ersetzt werden müssen, oder es muss erkennbar bleiben, welche Zeichen ursprünglich ? waren.

Teilaufgabe e: Falls ein Verfahren gefunden und angewendet wurde, muss die Anzahl der Veränderungen angegeben werden. Wenn nicht erkennbar ist, welche Zeichen geändert werden, dann wird 1 Punkt abgezogen.

Wird die Permutation nur als Folge von Spaltenindizes ausgegeben, können aufgrund der schlechter Überprüfbarkeit der Ergebnisse 2 Punkte abgezogen werden.

Aus den Einsendungen: Perlen der Informatik

Der Tauchgang nach den Perlen der Informatik ist noch nicht abgeschlossen.