Aufgabe 3: Konfetti

Teilnahme-ID: 77075

Bearbeiter/-in dieser Aufgabe:
Felix Du

27. Juni 2025

Inhaltsverzeichnis

1 Lösungsidee

1.1 Problemformulierung

Das Problem lässt sich zunächst einmal auf eine Matrix $M \in \{-1, 0, 1\}^{r \times c}$ abstrahieren, in der jede Spalte c einen Zeitslot und jede Zeile r eine Person beschreibt. Gefragt ist grundlegend nach der „Consecutive One's Property“ (abgk. C1P) von M; diese ist erfüllt, wenn es eine Permutation der Spaltenmenge C gibt, sodass die Einsen für alle Zeilen zusammenstehen. Das Problem wird üblicherweise als C1P-Problem bezeichnet. Die einzelnen Aufgabenteile spezifizieren und verändern das klassische C1P-Problem jedoch.

1.1.1 Aufgabe a)

Bei Aufgabe a) handelt es sich um das klassische C1P-Problem, wie jenes, das soeben beschrieben wurde.

1.1.2 Aufgabe b)

In Aufgabe b) wird erfragt, ob, falls Aufgabe a) lösbar ist, es nur diese mögliche Permutation aller Spalten gibt, oder ob mehr Lösungen existieren. In der Aufgabenstellung heißt es: „[…] Ändere dein Programm so, dass es diese Frage beantwortet“. In dem Kontext wird interpretiert, dass die Aufgabe fordert, einen `bool`-Wert zurückzugeben, ob denn mehrere Permutationen existieren. Dabei kann eine triviale Beobachtung gemacht werden. Sofern eine gegebene Permutation nicht gänzlich symmetrisch zur vertikalen Achse bei $\frac{|C|}{2}$ ist, ist die Umkehrung einer Permutation eine valide, andere Lösung des Problems. Da dies jdoch gewissermaßen eine Vereinfachung der Aufgabe darstellt, soll angenommen werden, dass jedwede Symmetrien der Permutation ignoriert werden sollen.

1.1.3 Aufgabe c)

In Aufgabe c) wird die erste Zeile vordefiniert und muss so in der Permutation auftreten, wie sie in der Eingabe formuliert ist. Je nach Lösungsmethode kann dieser Aufgabenteil verschiedenes bewirken. In der Aufgabenstellung heißt es: „Was ändert sich, wenn Anna sich doch wieder an ihre Zeile erinnert? Ändere dein Programm entsprechend“. Da Programm muss nur insofern verändert werden, dass diese Einschränkung mitimplementiert und berücksichtigt wird. Die Anzahl der möglichen Permutationen ändert sich in dem Sinne, dass eine zusätzliche Einschränkung nie die Menge an möglichen Permutationen erweitern wird. Stattdessen ist die Menge an Permutationen, die diese Einschränkung berücksichtigen, eine echte Teilmenge aller Permutationen für eine gegebene Matrix. Somit muss der Prozess der Erstellung der Permutationen nicht zwingend verändert werden, lediglich das Auslesen der Permutation muss jene Zeile berücksichtigen. Wie genau jedoch diese Einschränkung implementiert wird, hängt vom Algorithmus bzw. von der Lösungsmethode ab.

1.1.4 Aufgabe d)

In Aufgabe d) wird ein neues Zeichen „?“ in der Binärmatrix eingeführt. Der Algorithmus soll die Werte dieser Zeichen bestimmen und festlegen, sodass eine Permutation möglich ist. Diese zusätzliche Modifikation des Problems macht es deutlich komplexer. Einerseits können die „?“ als Hilfestellung wahrgenommen werden, denn es steht dem Algorithmus frei zu entscheiden, welches Zeichen hilfreich ist für eine lokale Entscheidung. Allerdings ist es trivial zu erkennen, dass eine Entscheidung, die lokal getroffen wird, zukünftig eine Permutation unmöglich macht. Wie komplex dieses Problem wirklich ist, wird noch folgend analysiert.

1.1.5 Aufgabe e)

Bisher waren alle Probleme Entscheidungsprobleme, die sich folgend formulieren lassen: *Gegeben eine Matrix M, gibt es eine Permutation der Spalten, sodass die Zeilen die C1P erfüllen?* Jede Modifikation durch die Aufgabenteile ist eine Modifikation jener Formulierung. In Aufgabe e) kriegt das Problem eine andere Dimension. Vorausgesetzt, die gegebene Matrix M besitzt keine Permutation, die die C1P erfüllt und es gibt ggb. auch keine Zuweisung der Zeichen „?“, die dies ändert, verändere so wenig Elemente von M wie möglich, sodass die Matrix M' eine gültige Permutation besitzt, wobei immer noch eine Zuweisung der „?“ stattfinden muss. Jede Veränderung eines Elements kann als Bit-Switch betrachtet werden, da alle Elemente Binärwerte sind. Es wird also als Optimierungsproblem formuliert, wobei die Voraussetzung die Unlösbarkeit von M ist.

1.2 Betrachtung der Komplexitätsklasse

Die in den Aufgabenteilen a) - e) formulierten Probleme lassen sich in drei große Probleme zusammenfassen:

1. Gegeben sei eine Matrix $M \in \{0,1\}^{r\times c}$; gibt es eine Permutation der Spalten, sodass die Zeilen die C1P erfüllen? Bezeichnet wird es als C1P-Problem.

2. Gegeben sei eine Matrix $M \in \{0,1,?\}^{r\times c}$; gibt es eine Zuweisung aller „?“, sodass eine Permutation existiert, die die C1P erfüllt? Bezeichnet wird es als AC1P-Problem.

3. Gegeben sei eine Matrix $M \in \{0,1,?\}^{r\times c}$; verändere so wenig Elemente von M wie möglich, sodass eine Permutation der Spalten existiert, die die C1P erfüllt? Bezeichnet wird es als OC1P-Problem.

Diese Probleme sind unterschiedlich komplex und werden dementsprechend auch in ihrer Komplexitätsklasse getrennt betrachtet. Schon vor mehr als zwei Dekaden wurden Lösungsalgorithmen mit linearer Zeitkomplexität von Booth und Lueker [1] präsentiert. Über PQ-Bäume, eine Datenstruktur, die Permutationen mit Rücksicht auf Einschränkungen darstellt, präsentieren sie eine Lösung in polynomieller Zeit. Dementsprechend ist das C1P-Problem, ob eine gültige Permutation existiert, sodass M die C1P erfüllt, in P. Das AC1P-Problem hingegen ist deutlich komplexer und liegt wahrscheinlich nicht mehr in P. Dazu wird anschließend eine Reduktion über das Hamilton-Kreis-Problem durchgeführt:

Theorem 1.1. *Das AC1P-Problem ist NP-vollständig.*

Die Komplexitätsklasse des AC1P-Problems soll anschließend bewiesen werden. Dazu wird ein bereits bekanntes NP-vollständiges Problem auf unser Problem reduziert, um die NP-Schwere des Problems zu beweisen. Zusätzlich soll die Zugehörigkeit des Problems in NP bewiesen werden, damit das Problem als NP-vollständig deklariert werden kann.

Lemma 1. *Das AC1P-Problem liegt in NP.*

Beweis. Damit gezeigt werden kann, dass das AC1P-Problem in NP liegt, wird demonstriert, dass ein Zertifikat für eine „Ja-Instanz“ in polynomieller Zeit verifiziert werden kann. Ein Zeritifkat bestehend aus:

- Ein Zuweisung $A :? \mapsto \{0,1\}$ für alle „?“-Zeichen.
- Eine Permutation π der Spalten.

Ein Verifikationsalgorithmus funktioniert folgendermaßen: A soll auf der Matrix M angewendet werden, sodass eine Binärmatrix M' entsteht. Die Spalten werden gemäß der Permutation π

geordnet und die Matrix M_π entsteht. Für jede Zeile r in M_π wird überprüft, ob die Einsen in einem Block zusammenstehen, sodass niemals ein Muster in Form von „1,0,...,1“ für irgendeine Zeile r entsteht. Da jedes Element einmal überprüft werden muss, ist die Zeitkomplexität des Algorithmus $O(m \cdot n)$, wenn m die Anzahl der Zeilen und n die Anzahl der Spalten ist. Somit ist das AC1P-Problem in NP. □

Lemma 2. *Das Hamilton-Kreis-Problem ist in polynomieller Zeit auf das AC1P-Problem reduzierbar.*

Beweis. Das Hamilton-Kreis-Problem, das NP-vollständig ist[1], soll anschließend auf das AC1P-Problem reduziert werden.

Definition 1 (Hamiltonian-Kreis-Problem)**.** Gegeben sei ein ungerichteter Graph $G = (V, E)$. Gibt es einen geschlossenen Pfad in G, der jeden Knoten in V genau einmal enthält?

Mit dem Graphen $G = (V, E)$ und $n = |V|$ Knoten soll eine Matrix M konstruiert werden:

- $M = \{m_{i,j} \mid m_{i,j} \in \{0, 1, ?\}\}$
- M hat n Spalten, eine für jeden Knoten $v \in V$.
- M hat $|E|$ Reihen, eine für jede Kante in $e \in E$.

Für jede Kante $e = (u, v)$ wird eine Zeile $r(e)$ erstellt, in welcher:

- $r(e)_u = 1$ (1 in Spalte u von $r(e)$)
- $r(e)_v = 1$ (1 in Spalte v von $r(e)$)
- $r(e)_w = ?$ (für alle $w \neq u, v$)

Diese Modellierung läuft trivialerweise in polynomieller Zeit, da für die Matrix M gilt: $M = \{0, 1, ?\}^{r \times c}$. Jedes einzelne Element kann in $O(1)$ erstellt werden. □

Lemma 3. *Gegeben sei ein Graph G und eine Matrix M. Der Graph G besitzt nur dann einen Hamilton Kreis, wenn es auch eine Zuweisung für alle „?“ aus M gibt, sodass M' die C1P durch eine Permutation erfüllen kann.*

Beweis. Beide Richtungen der Äquivalenz sollen bewiesen werden. Angenommen G hat einen Hamilton Pfad $P = (v_1, v_2, \ldots, v_n)$. Nunsoll daraus eine Lösung für das AC1P-Problem konstruiert werden. Die Spalten von M sollen nach der Reihenfolge in P permutiert werden, sodass die Permutation π entsteht. Für jede Zeile $r(e)$ mit $e = (v_i, v_j)$ sollen den „?“-Zeichen folgendermaßen Werte zugeordnet werden:

- Wenn v_i und v_j adjazent in P ($|i - j| = 1$) sind:
  - Alle $? = 0$ für die Zeile $r(e)$.
- Ansonsten:
  - $? = 1$ für alle Spalten v_k mit $\min(i, j) < k < \max(i, j)$ in Zeile $r(e)$.
  - $? = 0$ anderweitig.

Es muss bewiesen werden, dass π die C1P erfüllt. Für jede $r(e)$ mit $e = (v_i, v_j)$:

- Wenn $|i - j| = 1$ gilt, dann hat die Zeile genau zwei Einsen, dessen Spalten nebeneinander liegen, sodass die C1P offensichtlich für $r(e)$ erfüllt ist.

[1] Quelle

- Wenn $|i-j| > 1$ gilt, dann hat die Zeile Einsen in den entsprechenden Spalten $v_i, v_{i+1}, \ldots, v_j$ (angenommen $i < j$). Diese Einsen sind, entsprechend den Indices von v, in einem Block und erfüllen somit auch die C1P.

Somit hat M eine Konfiguration der „?“-Zeichen, sodass eine Permutation π existiert, die die C1P erfüllt, wenn der Graph G einen Hamilton Pfad P besitzt.

Angenommen M hat eine Konfiguration aller „?“-Zeichen und eine Permutation π, die die C1P erfüllt. Dann soll π die Reihenfolge der Knoten v mit $v'_1, v'_2, \ldots, v'_n$ definieren. Für eine Kante $e = (u, v) \in E$ hat die dazugehörige Zeile $r(e)$ nur Einsen in den Spalten u und v. Diese Einsen müssen durch die Permutation π in einer Sequenz erscheinen. Das heißt, dass für jede Kante $(u, v) \in E$ gilt, dass sich die Knoten u und v in den Positionen i und j in π befinden müssen, sodass sich entweder keine anderen Knoten zwischen ihnen befinden und $|i - j| = 1$ gilt oder alle Knoten zwischen ihnen eine 1 zugewiesen bekommen. Allerdings kann beobachtet werden, dass eine allen „?“ in $r(e)$ eine 0 zugewiesen werden kann, außer denen, die sich zwischen u und v in π befinden. Dies wird die C1P nicht verletzen, solange sie davor schon erfüllt war. Tatsächlich können wir eine neue Zuweisung für die entsprechende Zeile $r(e)$ für alle „?“ erstellen, in der gilt:

- $? = 0$, wenn $|i - j| = 1$ gilt.
- Ansonsten, $? = 1$ für alle ? zwischen u und v in π, 0 anderweitig.

Mit dieser Zuweisung für jede Zeile $r(e)$ der Kante $e \in E$ ist die C1P erfüllt, da alle Einsen in einer Reihe $r(e)$ einen Block bilden. Jetzt gilt: wenn es einen Pfad gemäß der Permutation π gibt, wäre dies ein Hamilton Pfad. $P = \{v'_1, v'_2, \ldots, v'_n\}$ soll solch ein Pfad sein. Für aufeinanderfolgende Knoten v'_i und v'_{i+1} in P muss es eine Kante $(v'i, v'i + 1) \in E$ geben. Wenn es nicht eine derartige Kante gäbe, dann gäbe es die entsprechende Zeile $r(e)$ in M nicht und alle Kanten, die diese Kante überqueren, würden in ihren Zeilen $r(e)$ nicht genau einen Block an Einsen haben, welches der C1P widerspricht. Somit ist $P = \{v'_1, v'_2, \ldots, v'_n\}$ ein Hamilton Pfad in G. □

Beweis. Nach Lemma 1 ist das AC1P-Problem in NP. Nach Lemmata 2 und 3 existiert eine Reduktion vom Hamilton-Kreis-Problem auf das AC1P-Problem in polynomieller Zeit, die die Lösung beibehält. Demnach ist das AC1P-problem NP-vollständig. □

Das OC1P-Problem scheint für Instanzen mit „?“ eine Erweiterung des AC1P-Problem sein, da eine Lösung für OC1P nur gefunden werden soll, wenn AC1P keine valide Lösung hat. Da dies jedoch an sich schon, wie soeben bewiesen, NP-vollständig ist, ist die Feststellung der Bedingung für OC1P nicht in polynomieller Zeit möglich und somit OC1P auch NP-schwer. Dies trifft jedoch keine Aussage darüber, wie schwer das Kernproblem OC1P ist. Daher nehmen wir folgend an, dass bekannt ist, das für eine Matrix M keine Zuweisung A existiert, sodass eine Permutation die C1P erfüllt oder keine „?“ enthalten sind und setzen diese Bedingung voraus. Es wird also nur das Problem betrachtet, die minimale Anzahl an Bit-Switches in M zu finden, sodass eine valide Permutation existiert. Folgend soll bewiesen werden, dass auch dieses Problem NP-schwer ist. Dazu soll das Problem zunächst einmal formalisiert werden:

Definition 2 (OC1P)**.** Gegeben sei eine Matrix:

$$M \in \{0,1\}^{m \times n}$$

Gesucht ist eine Matrix $M' \in \{0,1\}^{m \times n}$ und eine Spaltenpermutation π der n Spalten, sodass M'_π die *Consecutive-Ones-Property* (C1P) besitzt, wobei die Hamming-Distanz[2] $\Delta(M, M') = \#\{(i,j) \mid M_{ij} \neq M'_{ij}\}$ minimiert wird.

Theorem 1.2. *Das OC1P-Problem ist NP-schwer.*

[2]Die Anzahl unterschiedlicher Binaries in zwei gleichgroßen Binarystrings.

Lemma 4. *Das AC1P-Problem ist in polynomieller Zeit auf das OC1P-Problem reduzierbar.*

Beweis. Gegeben sei eine Instanz $M \in \{0, 1, ?\}^{m \times n}$ des AC1P-Problems . Wir konstruieren in polynomieller Zeit eine äquivalente Instanz (M', k) des OC1P-Problems wie folgt:

1. Für jede Zeile i und jede Spalte j setzen wir:

$$M'_{i,j} = \begin{cases} M_{i,j}, & \text{falls } M_{i,j} \in \{0, 1\} \\ 0, & \text{falls } M_{i,j} = ? \end{cases}$$

2. Sei q die Anzahl der „?“ in M. Es gilt $k = q$.

Die Konstruktion kann offensichtlich $O(m \cdot n)$ durchgeführt werden, da für jeden der $m \cdot n$ Einträge der Matrix M eine einfache Überprüfung und ggf. eine Zuweisung vorgenommen wird. Zusätzlich wird einmal durch die Matrix iteriert, um q zu bestimmen, was ebenfalls in $O(m \cdot n)$ Zeit möglich ist. □

Lemma 5. *Eine Instanz M von AC1P hat genau dann eine Lösung, wenn auch die konstruierte Instanz (M', k) von OC1P eine Lösung hat.*

Beweis. Die Äquivalenz soll in beide Richtungen gezeigt werden. Angenommen, M hat eine gültige Zuweisung A, sodass die resultierende Matrix M^* eine C1P-Permutation besitzt. Nun wird eine Menge von Bit-Switches für M' definiert: Für jedes Element (i, j) mit $M_{i,j} =?$ und $M^*_{i,j} = 1$ ändern wir das entsprechende Bit in M' von 0 auf 1. Für alle anderen Elemente bleibt M' unverändert. Sei M'' die nach diesen Änderungen aus M' resultierende Matrix. Es wird gezeigt, dass M'' identisch mit M^* ist:

- Für alle Elemente (i, j) mit $M_{i,j} \in \{0, 1\}$ gilt: $M'_{i,j} = M_{i,j} = M^*_{i,j}$ und diese Elemente werden nicht geändert, also $M''_{i,j} = M^*_{i,j}$.
- Für alle Elemente (i, j) mit $M_{i,j} = ?$ gilt:
  - Wenn $M^*_{i,j} = 0$, dann wurde in der Konstruktion $M'_{i,j} = 0$ gesetzt und dieser Wert bleibt unverändert, also $M''_{i,j} = 0 = M^*_{i,j}$.
  - Wenn $M^*_{i,j} = 1$, dann wurde in der Konstruktion $M'_{i,j} = 0$ gesetzt und dieser Wert wird auf 1 geändert, also $M''_{i,j} = 1 = M^*_{i,j}$.

Somit ist $M'' = M^*$. Da M^* per Annahme eine C1P-Permutation besitzt, hat auch M'' eine solche Permutation. Die Anzahl der vorgenommenen Bit-Switches ist gleich der Anzahl der Elemente (i, j) mit $M_{i,j} =?$ und $M^*_{i,j} = 1$. Diese Anzahl ist offensichtlich höchstens gleich der Anzahl aller „?“ in M, also $q = k$. Somit können wir M' mit höchstens k Bit-Switches in eine Matrix umwandeln, die eine C1P-Permutation besitzt. Die konstruierte Instanz (M', k) hat also eine Lösung. Anschließend soll der selbe Beweis in die entgegengesetzte Richtung geführt: angenommen, es gibt eine Menge von höchstens $k = q$ Bit-Switches für M', sodass die resultierende Matrix M'' eine C1P-Permutation besitzt. Es wird eine Zuweisung für die „?“ in M wie folgt konstruiert:

- Für jede Position (i, j) mit $M_{i,j} = ?$ wird der Wert auf $M''_{i,j}$ gesetzt.
- Alle anderen Einträge in M bleiben unverändert.

Sei M^* die resultierende Matrix nach dieser Zuweisung. Es wird gezeigt, dass es eine optimale Lösung für das Problem OC1P-Problem gibt, die nur Bits an Elementen ändert, die „?“ in M entsprechen. Angenommen, es gibt eine optimale Lösung, die ein Bit an einer Position (i, j) ändert, für die $M_{i,j} \in \{0, 1\}$ gilt. Nach der Konstruktion ist $M'_{i,j} = M_{i,j}$. Betrachten wir nun

eine alternative Lösung, die dieses Bit nicht ändert. Falls diese Lösung keine gültige C1P-Matrix erzeugt, würde dies bedeuten, dass die ursprüngliche Instanz M keine Lösung hat, da keine Zuweisung der „?“ die C1P-Eigenschaft erreichen kann, ohne den Wert an Position (i, j) zu ändern. Dies steht im Widerspruch zu unserer Annahme, dass die ursprüngliche Instanz lösbar ist. Somit gibt es eine optimale Lösung, die nur Bits an Elementen ändert, die „?“ in M entsprechen. Wir können dadurch annehmen, dass M'' aus M' nur durch Änderungen an Elementen entsteht, die „?“ in M entsprechen. Da M'' eine C1P-Permutation besitzt und M^* an diesen ?-Positionen mit M'' übereinstimmt, hat auch M^* eine C1P-Permutation. Die Anzahl der benötigten Bit-Switches entspricht der Anzahl der Elemente (i, j), für die $M_{i,j} = ?$ und $M'_{i,j} \neq M''_{i,j}$ gilt. Nach unserer Konstruktion ist $M'_{i,j} = 0$ für alle Elemente mit $M_{i,j} =?$. Somit werden genau die „?“ geändert, die in M'' den Wert 1 haben. Da die Gesamtzahl der vorgenommenen Änderungen höchstens $k = q$ beträgt und wir annehmen können, dass nur Elemente mit „?“ geändert werden, ist unsere Zuweisung gültig. Somit hat die ursprüngliche Instanz M eine Lösung für das AC1P-Problem. □

Beweis. Es wurde in Lemma 4 eine Reduktion in Polynomialzeit vom AC1P-Problem auf das Problem OC1P-Problem konstruiert und in Lemma 5 gezeigt, dass die ursprüngliche Instanz genau dann eine Lösung hat, wenn die konstruierte Instanz eine Lösung hat. Da AC1P NP-vollständig ist, folgt, dass OC1P NP-schwer ist. □

1.3 Modellierung als IP

Für alle Aufgabenteile lassen sich IP-Modelle formulieren. Je nach Inputgröße sind diese jedoch nur mehr oder weniger feasible in angemessener Zeit[3], da IP-Modelle für Permutationen immer ein gleich näher erklärtes Indexset fordern, dass alle möglichen Permutation als Entscheidungsvariablen modelliert. Anschließend werden für verschiedene Aufgabenteile IP-Modelle bzw. zusätzliche Constraints aufgeführt.

1.3.1 Feasibility-Modell (a)

Der gegebenen Matrix M werden zwei Spalten hinzugefügt, die ausschließlich Nullen enthalten und die Matrix M' entsteht. Jedes Element, das den Wert „?“ hat, wird mit einer Variable $m'_{r,c}$ beschrieben, wobei r die Zeile und c die Spalte darstellt. Außerdem ein Permutation-Indexset I, eine quadratische Matrix mit Dimensionsgröße $|C| + 2$, als $x_{i,j}$ mit i als Spalte und j als Position der Spalte in der Permutation. Außerdem ist ggbf. ein Array l der Länge $|C|$ gegeben, das die erste Zeile beschreibt. Es wird keine klare Zielfunktion formuliert, da es lediglich um eine valide Permutation geht, die alle gegebenen Constraints erfüllen muss, jedoch kein Ziel der Minimierung oder Maximierung aufweist. Dementsprechend lassen sich folgende Constraints

[3] So wie auf der Bwinf Seite gefordert wurde.

formulieren:

$$x_{1,1} = 1 \tag{1}$$

$$x_{c+2,c+2} = 1 \tag{2}$$

$$\sum_{n=1}^{c+2} x_{n,j} = 1 \qquad \forall\, j \tag{3}$$

$$\sum_{n=1}^{c+2} x_{i,n} = 1 \qquad \forall\, i \tag{4}$$

$$y_{r,j} = \sum_{n=1}^{c+2} m'_{r,n}\, x_{n,j} \qquad \forall\, r, \forall j \tag{5}$$

$$\sum_{n=1}^{c+1} (y_{r,n} + y_{r,n+1} - 2\, y_{r,n}\, y_{r,n+1}) \leq 2 \qquad \forall\, r \tag{6}$$

$$\text{oder äquivalent: } \sum_{n=1}^{c+1} (|y_{r,n} - y_{r,n+1}|) \leq 2 \qquad \forall\, r \tag{7}$$

Matrix M' ist eine durch zwei „0-Spalten“ modifizierte Matrix von M. Dies wird getan, um die C1P besser beschreiben zu können. Denn wenn jede Zeile von Nullen „umrandet“ wird, dann muss für eine gültige Zeile r^* folgendes gelten: für zwei aufeinanderfolgende Elemente in einer Permutation $y_{r^*,j}$ und $y_{r^*,j+1}$ dürfen diese sich nur insgesamt entweder zweimal oder gar nicht unterscheiden, falls eine Zeile nur Nullen enthält. Ein Einserblock wird immer von Nullen umrandet und somit gibt es an den Rändern jeweils ein Bitflip zwischen zwei aufeinanderfolgenden Elementen, also insgesamt 2. Da also gilt:

$$|y_{r^*,j} - y_{r^*,j+1}| = 1 \iff y_{r^*,j} \neq y_{r^*,j+1} \tag{8}$$

wird Gleichung (7) aufgestellt, die Gleichung (8) als Constraint implementiert, oder äquivalent dazu eine UND-Verknüpfung, modelliert als Constraint in Gleichung (6). Die Gleichungen (1) und (2) legen fest, dass die Hilfsspalten an den Rändern der Matrix M bleiben. Gleichung (3) und (4) bestimmen die Struktur von I, denn eine Spalte darf in einer Permutation nur in einer Position auftauchen. Gleichung (5) liest die Elemente der Permutation aus, sodass die Permutationsmatrix Y entsteht.

1.3.2 Multiple Lösungen (b)

Um zu wissen, ob es mehrere Lösungen gibt, soll versucht werden, eine zweite Lösung zu finden. Dabei muss für den zweiten Durchlauf des IP eine Constraint eingeführt werden, sodass die erste Permutation nicht länger möglich ist. Dazu muss zunächst das IP ein erstes Mal gelöst werden. Die korrespondierenden Variablen $x_{i,j}$ zur Permutation π werden dann in einer Menge $I_\pi, I_\pi \subset$ gespeichert, um folgende Constraint einzuführen:

$$\sum_{x_{i,j} \in I_\pi} x_{i,j} \leq n - 1 \tag{9}$$

Das bedeutet, dass die Summe aller Permutationsvariablen von π in der nächsten Permutation nicht gleich der Spaltenanzahl sein darf, denn sonst wäre es die selbe Permutation π.

1.3.3 Vorgegebene erste Zeile (c)

Damit die vorgegebene erste Zeile in der Lösung des IP auftaucht, müssen wir die erste Zeile der Permutationsmatrix Y ebenso festlegen. So gilt also im Falle, dass l gegeben ist, zusätzlich:

$$y_{1,n} = l_n \quad \forall n \tag{10}$$

1.3.4 „?“-Zeichen (d)

Manche Elemente der Matrix $m'_{i,j}$ sind „?“-Zeichen. Im IP können diese ganz einfach als Entscheidungsvariablen modelliert werden. Für die Elemente, für die gelten $m'_{i,j} =?$, werden diese zu Entscheidungsvariablen initialisiert $m'_{i,j} \in \{0, 1\}$, sodass der Solver für eine Lösung diese festlegt.

1.3.5 OC1P-Constraints (e)

Aufgabenteil e) formuliert ein Optimierungsproblem; somit wird eine Zielfunktion eingeführt. Da das Ziel als die minimale Anzahl an Bitswitches, die eine Permutation mit der C1P erzeugt, formuliert werden kann, wird die Zielfunktion folgendermaßen aufgestellt:

$$\min \sum_{i=1}^{m} \sum_{j=1}^{n} b_{i,j} \tag{11}$$

Außerdem werden anstatt Gleichung (6) und (7) neue Gleichungen eingeführt, die eine neue Permutationsmatrix Y' einführt, in der Bitswitches stattfinden können. Es wird eine UND-Verknüpfung wie auch oben durchgeführt, um zu erkennen, wenn zwei Elemente y und y' sich voneinander unterscheiden. Dies wird auch zugleich der Wert für die Bitswitch-Variable b. Außerdem gilt die C1P Constraint nun für die Y', da die Bitswitches mit einbezogen werden müssen:

$$\sum_{n=1}^{c+2} \left(y_{r,n} + y'_{r,n} - 2 \cdot y_{r,n} \cdot y'_{r,n}\right) = b_{r,n} \qquad \forall\, r \tag{12}$$

$$\sum_{n=1}^{c+1} \left(y'_{r,n} + y'_{r,n+1} - 2 \cdot y'_{r,n} \cdot y'_{r,n+1}\right) \leq 2 \qquad \forall\, r \tag{13}$$

Da die Summe aller Variablen b minimiert wird und dies exakt der Zielfunktion der Aufgabenstellung entspricht, modelliert dieses IP somit genau das OC1P-Problem.

1.4 PQ-Tree

2 Umsetzung

Alle Algorithmen und Modelle werden in Python 3.11 umgesetzt. Sie werden auf einem MacOS-System mit einem Intel i7 2.6-GHz 6-Kern Prozessor und maximal 12 GB Hauptspeicher (RAM) berechnet.

2.1 IP-Modell

Das IP-Modell wird in der Python-Bibliothek `pyscipopt`[4] mit der Klasse `Model` implementiert. Da die Constraints 1 zu 1 in ein Programm umgesetzt werden, scheint es mir trivial, genauer zu erläutern, was jede Quellcode-Zeile tut. Stattdessen wird eine Analyse der Variablen- und Constraintentwicklung folgen, sprich, wie sich die Anzahl dieser im Verhältnis zu einer gegebenen Inputgröße verhält, da dies hauptsächlich die realistische Lösbarkeit eines Solvers bestimmt.
Zunächst betrachten wir das Permutations-Indexset I. Da die Permutation über die Spalten der Matrix M' stattfindet, gilt somit, wenn m die Zeilenanzahl und n die Spaltenanzahl darstellt:

$$O((n+2)^2) \implies O(n^2) \tag{14}$$

Für die Permutationsmatrix Y, die die Permutation π von I darstellt, gibt es in etwa so viele Variablen, wie in M.

$$O(m \cdot (n+2)) \implies O(mn) \tag{15}$$

[4] `https://pyscipopt.readthedocs.io/en/latest/whyscip.html`, zuletzt aufgerufen: 24. April 2025

Da es zur Gesamtgröße $M = n \cdot m$ vergleichsweise nur wenige „?“ gibt, werden diese in der Analyse vernachlässigt. Insgesamt ergibt dies für die Anzahl und somit auch Zeitkomplexität der Variablen:

$$O(n^2) + O(mn) = O(n^2 + nm) \tag{16}$$

Für die Constraints wird zunächst Gleichung (6) betrachtet. Für jede Zeile wird $n + 1$ mal darüber iteriert, sodass gilt:

$$O(m \cdot (n + 1)) \implies O(nm) \tag{17}$$

Wie in Gleichung (14) dargestellt wird, werden insgesamt $m(n + 2)$ Variablen für Y erstellt. Die Constraints für jene Variablen, die die Werte für die Variablen zuweisen, müssen über das gesamte Indexset I iterieren, sodass für die gesamte Zeitkomplexität gilt:

$$O(m \cdot (n + 2)^2) \implies O(mn^2) \tag{18}$$

Somit gilt insgesamt, für die Anzahl an Variablen und Constraints:

$$O(mn^2) + O(n^2) + O(nm) = O(mn^2 + n^2 + nm) \implies O(mn^2 + n^2) \tag{19}$$

Wie bereits gesagt sind die „?“-Zeichen für eine Laufzeitentwicklung unerheblich, gegebenenfalls eine vorgegebene festgelegte Zeile aus Aufgabenteil c) und die Constraints zum Verschränken einer ersten gefundenen Lösung aus Aufgabenteil b) auch. Nun bleibt noch die IP-Modifikation für das OC1P-Problem. Diese initialisiert eine neue Variable b, die in der Größe der Matrix M entspricht. Es gibt äquivalent dazu genauso viele Constraints, die die Werte für b festlegen. Somit gilt für das OC1P-Modell:

$$O(mn^2 + n^2) + O(nm) \implies O(mn^2 + n^2) \tag{20}$$

Somit ändert die Modifikation für das OC1P-Problem nichts für Zeitkomplexität für die Erstellung des Modells. Da das Integer-Programming-Problem generell NP-schwer ist, ist die Zeitkomplexität des Lösens auf jeden Fall nicht polynomiell. Eine genaue Analyse ist nicht möglich, da dies von der Methodik des Solvers abhängt. Allgemein lässt sich sagen, dass die Zeitkomplexität eines IP-Solvers exponentiell zur Anzahl der Binaries liegt. Folgend wird der Prozess des gesamten Algorithmus, der eine Eingabe identifiziert und dazu den passenden Aufgabenteil zugeordnet, vorgestellt:

Algorithm 1 Prozess des IP

```
1: function IPModel(M)
   model = BuildModel(n, m)                          ▷ Feasibility-Modell
   SolveModel(model)
2:     if model.status = „optimal“ then
3:         AddSolCons(model)                         ▷ Ausschließen der ersten Lösung
4:         SolveModel(model)
5:         sol = RetrieveModel(model)
6:         if model.status = „optimal“ then
7:             bool = True                           ▷ Es existiert eine zweite gültige Permutation
8:             return sol, True
9:         else
10:            bool = False                          ▷ Es existiert keine zweite Permutation
11:            return sol, False
12:        end if
13:    else
14:        model = BuildOptModel(M)                  ▷ Optimization-Modell; OC1P
15:        SolveModel(model)
16:        sol = RetrieveModel(model)
17:        return sol, False
18:    end if
19: end function
```

Die Permutation einer Lösung kann aus den Variablen von I ausgelesen werden. Für eine Variable $x_{i,j} = 1$ ist die i-te Zeile an der Position j. Anschließend sollen die verwendeten Funktionen der Klasse `Model` von `pyscipopt` zur Erstellung des IP-Modells kurz genannt werden:

1. Model.addVar()
   - Fügt dem Modell eine Variable hinzu.
2. Model.addCons()
   - Fügt dem Modell eine Einschränkung/Constraint hinzu.
3. Model.setObjective()
   - Fügt dem Modell die Zielfunktion hinzu. Falls es sich um ein Feasibility-Modell handelt, muss diese nicht hinzugefügt werden.
4. quicksum()
   - Summiert einen Ausdruck über eine for-Schleife auf. Dient nur zur übersicht und ist eine Kurzschreibweise einer normalen for-Schleife.
5. Model.freeTransform()
   - Setzt nach dem Lösen das Modell wieder in den Presolve-Zustand zurück, sodass für ein zweites Lösen zusätzliche Constraints hinzugefügt werden können.

3 Beispiele

Anschließend werden die gelösten Beispiele vorgestellt. Die Eingaben „konfetti05.txt“, „konfetti11.txt“, „konfetti12.txt“ sind aufgrund ihrer Größe nicht in angemessener Zeit für das IP-Modell lösbar. Wenn es für Eingabe keine gültige Permutation ohne Bit-Switches gab, wurde

automatisch das IP-Modell der Teilaufgabe e) verwendet. Die Koordinaten der „?“, die Koordinaten der Bit-Switches, sowie die Nummerierung der Spalten beginnt von 1. Koordinaten werden immer als Tupel (r, c) angegeben, wobei r die Zeile und c die Spalte beschreibt. Die Spaltennummerierung bezieht sich immer auf die Ursprungsreihenfolge in der Eingabe; die Permutation wird von links nach rechts angegeben. Die Koordinaten der „?“ bezieht sich auf die Koordinaten in der originalen Eingabe mit der ursprünglichen Permutation, während sich die Koordinaten der Bit-Switches immer auf die Stellen in der gegebenen Permutation beziehen. Dies schien sinnvoll, um Zuweisung der „?“ der Eingabe besser zuordnen zu können, während die Stellen der Bit-Switches ohnehin nur relevant in der jeweiligen Permutation sind. Falls es für eine Eingabe mehr als eine Permutation gibt, die gültig ist, wird eine alternative Permutation mit einer gegebenenfalls anderen Zuweisung der „?“ angegeben. Dies geschieht nur für lösbare Modelle, da eine alternative Permutation für eine Eingabe, die nur durch Bit-Switches lösbar gemacht werden kann, sinnfrei erscheint. BwInf selbst sieht nicht vor, dass für Eingaben mit „?“, Aufgabenteil b), ob es mehrere gültige Permutationen gibt, bearbeitet wird, da eine andere Zuweisung der „?“ genügen könnte. Für die alternativen Permutationen der Eingaben 06-13 wurde darauf geachtet, sofern sie lösbar sind, dass sie sich in ihrer Permutation und nicht nur in der Zuweisung unterscheiden müssen. Gelegentlich sind Werte wie -0.0 zu sehen; dies liegt am Lösungsverfahren des Solvers der gegebenenfalls eine LP-Relaxation durchführt, wodurch anschließend gerundet werden muss, wodurch wiederum Zahlen wie -0.0 entstehen. Es gilt $-0.0 = 0$.

3.1 konfetti00.txt

Dies ist das Beispiel auf dem Aufgabenblatt. Wie schon erwähnt, wurden auch für Eingaben mit „?“ alternative Permutationen berechnet. An diesem Beispiel sit gut zu sehen, dass dies nicht über die Zuweisung geschieht, sondern tatsächlich andere Permutationen berechnet werden.

Tabelle 1: `konfetti00.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt mehrere gültige Permutationen.
Permutation 1	[4, 1, 3, 2]
?-Zeichen 1	$\{\,(2,3) \mapsto 1.0\,\}$
Permutation 2	[2, 3, 1, 4]
?-Zeichen 2	$\{\,(2,3) \mapsto 1.0\,\}$
Laufzeit	0.0277 s

3.2 konfetti01.txt

Tabelle 2: `konfetti01.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt mehrere gültige Permutationen.
Permutation 1	[3, 9, 4, 2, 7, 6, 5, 8, 1]
Permutation 2	[1, 8, 5, 6, 7, 2, 4, 9, 3]
Laufzeit	0.2997 s

3.3 konfetti02.txt

Tabelle 3: `konfetti02.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt keine gültigen Permutationen.
Minimale Bit-Switches	2
Permutation	[3, 5, 4, 6, 2, 7, 1, 8, 9]
Bit-Switches	$[(1,1),(3,7)]$
Laufzeit	3.6038 s

3.4 konfetti03.txt

Hierbei ist interessant, dass jede mögliche Permutation eine gültige ist. So ist auch die sehr schnelle Laufzeit zu begründen.

Tabelle 4: `konfetti03.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt mehrere gültige Permutationen.
Permutation 1	[5, 4, 1, 3, 2]
Permutation 2	[5, 1, 2, 3, 4]
Laufzeit	0.0116 s

3.5 konfetti04.txt

Tabelle 5: `konfetti04.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt mehrere gültige Permutationen.
Permutation 1	[5, 15, 14, 3, 13, 9, 2, 7, 17, 4, 11, 6, 1, 10, 8, 16, 12]
Permutation 2	[12, 17, 13, 9, 2, 7, 3, 14, 5, 15, 4, 11, 6, 1, 10, 16, 8]
Laufzeit	5.4843 s

3.6 konfetti05.txt

Für dieses Beispiel wurde keine Lösung gefunden, da die Eingabe mit ca. 1000 Spalten und ca. 100 Zeilen zu groß für das IP-Modell. In dieser Größe würde das IP-Modell ca. $100 \cdot 1000^2 + 1000^2 \approx 10^8$ Variablen besitzen, welches auf einem normalen PC mit Standard-Solvern definitiv nicht lösbar wäre. Da diese Eingabe keine „?“ besitzt, hätte man dieses Problem mit einem PQ-Baum mit der Laufzeit $O(mn)$ lösen können.

3.7 konfetti06.txt

Tabelle 6: `konfetti06.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt mehrere gültige Permutationen.
Permutation 1	[5, 16, 10, 9, 15, 14, 12, 2, 4, 19, 18, 21, 7, 8, 20, 1, 6, 3, 13, 11, 17]
Permutation 2	[5, 16, 10, 14, 9, 15, 12, 4, 19, 2, 18, 21, 7, 8, 20, 1, 6, 11, 3, 13, 17]
Laufzeit	0.7766 s

3.8 konfetti07.txt

Tabelle 7: `konfetti07.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt keine gültigen Permutationen.
Minimale Bit-Switches	2
Permutation	[17, 9, 10, 7, 5, 11, 6, 4, 19, 15, 21, 13, 1, 20, 16, 3, 2, 14, 18, 8, 12]
Bit-Switches	[(7, 20), (8, 20)]
Laufzeit	8.1856 s

3.9 konfetti08.txt

Tabelle 8: `konfetti08.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt mehrere gültigen Permutationen.
Permutation 1	[21, 18, 16, 3, 10, 14, 8, 2, 7, 13, 5, 17, 20, 9, 19, 15, 4, 11, 12, 1, 6]
?-Zeichen 1	$\{(2,5) \mapsto 1.0, (2,20) \mapsto 1.0, (6,1) \mapsto -0.0, (6,19) \mapsto 1.0, (8,7) \mapsto 1.0, (8,14) \mapsto 1.0\}$
Permutation 2	[18, 21, 16, 10, 3, 14, 8, 2, 7, 17, 13, 5, 19, 15, 9, 4, 20, 11, 12, 1, 6]
?-Zeichen 2	$\{(2,5) \mapsto 1.0, (2,20) \mapsto 1.0, (6,1) \mapsto -0.0, (6,19) \mapsto 1.0, (8,7) \mapsto 1.0, (8,14) \mapsto 1.0\}$
Laufzeit	0.3414 s

3.10 konfetti09.txt

Tabelle 9: `konfetti09.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt mehrere gültige Permutationen.
Permutation 1	[5, 4, 11, 12, 3, 9, 2, 10, 7, 1, 13, 6, 8]
?-Zeichen 1	$\{(1,1) \mapsto 1.0, (2,2) \mapsto 1.0, (2,4) \mapsto 0.0, (2,12) \mapsto 0.0, (4,4) \mapsto 1.0, (5,12) \mapsto 0.0, (6,8) \mapsto 0.0\}$
Permutation 2	[8, 6, 13, 7, 1, 10, 9, 2, 11, 12, 3, 4, 5]
?-Zeichen 2	$\{(1,1) \mapsto 1.0, (2,2) \mapsto 0.0, (2,4) \mapsto 0.0, (2,12) \mapsto 0.0, (4,4) \mapsto 1.0, (5,12) \mapsto 0.0, (6,8) \mapsto 0.0\}$
Laufzeit	4.8093 s

3.11 konfetti10.txt

Tabelle 10: `konfetti10.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt keine gültigen Permutationen.
Minimale Bit-Switches	3
Permutation	[12, 5, 10, 6, 11, 3, 9, 1, 4, 8, 13, 7, 2]
?-Zeichen	$\{ (3,4) \mapsto 1.0,\ (5,2) \mapsto 0.0,\ (5,10) \mapsto 1.0,\ (6,10) \mapsto 0.0,\ (7,5) \mapsto 0.0 \}$
Bit-Switches	$[(1,\ 7), (3,\ 2), (7,\ 12)]$
Laufzeit	103.1874 s

3.12 konfetti11.txt

Wie auch in Beispiel 05 konnte keine Lösung gefunden werden, da mit ca. 1000 Spalten und ca. 100 Zeilen das IP-Modell ca. 10^8 Variablen besitzt und somit deutlich zu groß ist. Da diese Eingabe zusätzlich noch „?“ besitzt, davon ca. 5500, genügt hierfür auch nicht länger ein PQ-Baum. Weiteres soll in Kapitel 4 diskutiert werden.

3.13 konfetti12.txt

Hier gilt gleiches wie für Beispiel 11.

3.14 konfetti13.txt

Tabelle 11: `konfetti13.txt`

Beschreibung	**Ausgabe**
Hinweis	Es gibt keine gültigen Permutationen.
Minimale Bit-Switches	4
Permutation	[14, 1, 9, 15, 4, 8, 7, 10, 3, 12, 11, 2, 5, 6, 13, 16]
?-Zeichen	$\{ (1,1) \mapsto 0.0,\ (1,11) \mapsto 1.0,\ (2,2) \mapsto 0.0,\ (5,2) \mapsto 0.0,\ (5,4) \mapsto 0.0,\ (8,10) \mapsto 1.0,\ (8,14) \mapsto 0.0 \}$
Bit-Switches	$[(5,\ 8), (6,\ 6), (6,\ 16), (8,\ 4)]$
Laufzeit	251.4963 s

4 Diskussion

Anschließend sollen kurz mögliche Lösungswege für die Eingaben 05, 11, 12 genannt und diskutiert werden, da diese nicht durch das hier aufgestellte IP-Modell gelöst wurden. Für das Beispiel 05 ohne „?“ können PQ-Bäume implementiert werden, die nach Booth und Lueker [1] eine lineare Zeitkomplexität haben. Pq-Bäume sind eine dynamische Datenstruktur, die Permutationen unter Einschränkungen darstellen können. Dabei stellt jedes Blatt des Baums ein zu permutierendes Element dar. Die Beispiele 11 und 12 sind jedoch komplizierter. Ein Brute-Force-Ansatz für die „?“ ist hierfür undenkbar; wenn man diesen mit den sehr schnellen PQ-Bäumen implementieren würde, hätte dieser Algorithmus immer noch eine Zeitkomplexität von $2^{|?|} \cdot O(mn) \implies O(2^{|?|})$ und wäre somit schon für kleine Mengen „?“ sehr ineffizient. Eine Möglichkeit wäre, Heuristiken für die Zuweisung der „?“ zu finden, diese dann durch PQ-Bäume überprüfen zu lassen und die „falschen“ Zuweisungen, die den PQ-Baum verhindern, mit einer Art Backtracking zu korrigieren. Dazu ist allerdings eine gute Heuristik nötig, da der Algorithmus durch das Backtracking in seiner Laufzeit explodiert.

5 Quellcode

```
def modify_matrix(matrix):
    for x in range(len(matrix)):
        matrix[x].append(0)
        matrix[x].insert(0, 0)
    return matrix

def feasibility_model(matrix, r, c, line, line_bool, filename):
    # Fügt die Hilfsspalten hinzu.
    mod_matrix = modify_matrix(matrix)
    model = Model(filename)
    # Liest ggb. "?" ein und ersetzt es durch eine Binärvariable.
    for i in range(r):
        for j in range(c+2):
            if mod_matrix[i][j] == -1:
                mod_matrix[i][j] = model.addVar(vtype="B", name=f"
                    mod_matrix_{i}_{j}")

    # Das Permutationsindexset wird erstellt.
    indexset = {}
    for i in range(c+2):
        for j in range(c+2):
            indexset[(i, j)] = model.addVar(vtype="B", name=f"indexset_{i}_
                {j}")

    # Jede Spalte darf nur einmal vorkommen.
    for n in range(c+2):
        model.addCons(quicksum(indexset[(n, j)] for j in range(c+2)) == 1)
        model.addCons(quicksum(indexset[(i, n)] for i in range(c+2)) == 1)
    # Die Hilfsspalten dürfen nur an den Rändern stehen.
    model.addCons(indexset[(0, 0)] == 1)
    model.addCons(indexset[(c+1, c+1)] == 1)
    # Aus dem Indexset wird die tatsächliche Permutationsmatrix erstellt.
    permutation = {}
    for i in range(r):
        for j in range(c+2):
            permutation[(i, j)] = model.addVar(vtype="B", name=f"
                permutation_{i}_{j}")
            model.addCons(
                permutation[(i, j)]
                == quicksum(mod_matrix[i][n] * indexset[(n, j)] for n in
                    range(c+2))
            )
    # Falls die erste Zeile vorgegeben …ist
    if line_bool:
        for j in range(c):
            model.addCons(permutation[(0, j+1)] == line[j])
    # C1P-Constraint
    for i in range(r):
        model.addCons(
            quicksum(
                permutation[(i, n)] + permutation[(i, n+1)]
                - 2 * permutation[(i, n)] * permutation[(i, n+1)]
                for n in range(c+1)
            )
            <= 2
        )
```

```
    return model, indexset

def optimization_model(matrix, r, c, line, line_bool, filename):
    mod_matrix = modify_matrix(matrix)
    model = Model(filename)

    for i in range(r):
        for j in range(c+2):
            if mod_matrix[i][j] == -1:
                mod_matrix[i][j] = model.addVar(vtype="B", name=f"
                    mod_matrix_{i}_{j}")
    indexset = {}
    for i in range(c+2):
        for j in range(c+2):
            indexset[(i, j)] = model.addVar(vtype="B", name=f"indexset_{i}_
                {j}")
    for n in range(c+2):
        model.addCons(quicksum(indexset[(n, j)] for j in range(c+2)) == 1)
        model.addCons(quicksum(indexset[(i, n)] for i in range(c+2)) == 1)
    model.addCons(indexset[(0, 0)] == 1)
    model.addCons(indexset[(c+1, c+1)] == 1)

    permutation = {}
    for i in range(r):
        for j in range(c+2):
            permutation[(i, j)] = model.addVar(vtype="B", name=f"
                permutation_{i}_{j}")
            model.addCons(
                permutation[(i, j)]
                == quicksum(mod_matrix[i][n] * indexset[(n, j)] for n in
                    range(c+2))
            )
    if line_bool:
        for j in range(c):
            model.addCons(permutation[(0, j+1)] == line[j])

    # Alternativmatrix und Bit-Switches
    mod_permutation = {}
    bit_change = {}
    for i in range(r):
        for j in range(c+2):
            mod_permutation[(i, j)] = model.addVar(vtype="B", name=f"
                mod_permutation_{i}_{j}")
            bit_change[(i, j)]   = model.addVar(vtype="B", name=f"
                bit_change_{i}_{j}")

    # Hilfsspalten dürfen nicht geändert werden
    for i in range(r):
        model.addCons(mod_permutation[(i,0)]   == 0)
        model.addCons(mod_permutation[(i,c+1)] == 0)

    # Initialisierung der Bit-Switch-Werte
    for i in range(r):
        for n in range(1, c+1):
            model.addCons(
                bit_change[(i, n)]
                == permutation[(i, n)]
                + mod_permutation[(i, n)]
```

```
                - 2 * permutation[(i, n)] * mod_permutation[(i, n)]
            )
    # C1P-Constraint für mod_permutation
    for i in range(r):
        model.addCons(
            quicksum(
                mod_permutation[(i, n)] + mod_permutation[(i, n+1)]
                - 2 * mod_permutation[(i, n)] * mod_permutation[(i, n+1)]
                for n in range(c+1)
            )
            <= 2
        )
    # Ziel: Minimierung der Bit-Switches
    model.setObjective(
        quicksum(quicksum(bit_change[(i, j)] for i in range(r)) for j in
            range(c+2)),
        "minimize"
    )
    return model, indexset

def solve(model: Model):
    model.optimize()
    status = model.getStatus()
    return status in ("optimal", "feasible")

def add_sol_cons(model, old_perm):
    lhs = quicksum(indexset[k] for k, _ in old_perm.items())
    model.addCons(lhs <= len(old_perm) - 1)

def feasibility_solution(model: Model, c: int, indexset):
    permutation = [0] * c
    dec_vars = {}
    for i in range(1, c+1):
        for j in range(1, c+1):
            if round(model.getVal(indexset[(i,j)])) == 1:
                permutation[j-1] = i
    all_vars = model.getVars()
    mod_matrix_vars = [v for v in all_vars if v.name.startswith("
        mod_matrix_")]
    for var in mod_matrix_vars:
        i, j = map(int, var.name.split("_")[2:])
        dec_vars[(i+1, j)] = model.getVal(var)
    return permutation, dec_vars

def optimization_solution(model: Model, r: int, c: int, indexset):
    bit_changes = []
    permutation, dec_vars = feasibility_solution(model, c, indexset)
    all_vars = model.getVars()
    bit_change_vars = [v for v in all_vars if v.name.startswith("
        bit_change_")]
    for var in bit_change_vars:
        if round(model.getVal(var)) == 1:
            i, j = map(int, var.name.split("_")[2:])
            bit_changes.append((i+1, j))
    return permutation, dec_vars, bit_changes
```

Listing 1: Feasibility- und Optimierungsmodelle in Python

Literatur

[1] Kellogg S. Booth und George S. Lueker. „Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms“. In: *Journal of Computer and System Sciences* 13.3 (Dez. 1976), S. 335–379. DOI: https://doi.org/10.1016/S0022-0000(76)80045-1.