Aufgabe 3: Sudokopie

Inhaltsverzeichnis

Lösungsidee……..1 - 3

Umsetzung……..3 - 6

Beispiele……..7 - 10

Quellcode……..10 - 17

INFO: Neben den Begriffen aus der Aufgabenstellung verwende ich hier den Begriff „Sektor“, um jeweils einen der neun 3x3-Blöcke im Sudoku zu benennen.

Mit einem „Bereich“ meine ich einen Ausschnitt mit beliebigen Dimensionen aus einem Sudoku.

Lösungsidee

Das Sudoku kann in vier Dimensionen betrachtet werden: Zeilenblock, Spaltenblock, Zeile und Spalte. So bezeichnet die Position (0, 1, 2, 1) die Ziffer im ersten Zeilenblock, dem zweiten Spaltenblock, der dritten Zeile im Zeilenblock und der zweiten Spalte im Spaltenblock.

Eigenschaften der Permutationen

Wenn man die ersten vier Sudoku-Permutationen betrachtet, die in der Aufgabenstellung genannt sind, fällt auf, dass sie bestimmte Eigenschaften des Sudokus nicht verändern können. Das Vertauschen der Zeilenblöcke z.B. ändert zwar die Reihenfolge der Sektoren in einem Spaltenblock, aber nicht, aus welchen Sektoren die Spalte besteht. Dasselbe überträgt sich auf die Beständigkeit der Zeilenblöcke beim Tauschen der Spaltenblöcke.

Dieses Prinzip ist äquivalent, wenn man die Auswirkung des Spalten- und Zeilen-Vertauschens auf die Konsistenz der Zeilen und Spalten in einem Bereich betrachtet.

Wenn außerdem eine Ziffer in eine andere umbenannt wird, muss die zweite Ziffer auch umbenannt werden usw., sodass sich nichts daran ändert, wie viele Ziffern wie häufig in einem Bereich des Sudokus vorkommen.

Zuletzt die Rotation um 90 Grad. Eine Rotation **nach rechts** bzw. 90° im Uhrzeigersinn macht die Zeilenblöcke zu Spaltenblöcken und die Spalten- zu Zeilenblöcken in umgekehrter Reihenfolge (von rechts nach links). Eine Rotation **nach links** bzw. 3 mal 90° Macht die Spalten- zu Zeilenblöcken und die Zeilen- zu Spaltenblöcken in umgekehrter Reihenfolge (von unten nach oben). Und eine Rotation **um 180°** bzw. 2 mal 90° invertiert die Reihenfolge der Zeilen- **und** Spaltenblöcke und die von deren Zeilen, Spalten und einzelnen Ziffern.

Lösungs-Algorithmus

Deduktive Iteration möglicher Anordnungen der 8 Ebenen:

Das erste Sudoku aus der Textdatei ist das, das evtl. zum anderen umgeformt wird.

Wenn die Zeilenblöcke und die Zeilen in einem Block nach Index permutiert werden, dann hat es einen Einfluss auf das Ergebnis, welche der beiden Arten von Permutation zuerst geschieht. Dasselbe gilt für Spaltenblöcke und Spalten. Ich lege mich also fest, die Permutationen in dieser Reihenfolge zu iterieren: Zeilenblöcke, Spaltenblöcke, Zeilen für jeden Zeilenblock, Spalten für jeden Spaltenblock.

Insgesamt sind das $1+1+3+3=8$ Verkapselungen, was ohne zusätzliche Maßnahmen im Worst-Case 6^8, also $1\,679\,616$ Durchläufe wären, und das nur bei festgelegter Ziffernumbenennung.

Es ist also sinnvoll, nach einer Permutation auf einer beliebigen Ebene zu prüfen, ob mit dieser die insgesamt positionell festgelegten Bereiche sich in ihren möglichen Ziffern-Umbenennungen mit denen **von der vorangehenden Ebene** und **miteinander** überschneiden, und nur dann in die nächste Ebene gehen.

Die möglichen Ziffern-Umbenennungen für einen Bereich des Sudokus können ermittelt werden, indem zunächst in beiden Sudokus für diesen Bereich die Häufigkeit jeder Ziffer betrachtet wird. Wenn eine Ziffer in Sudoku 1 genauso oft vorkommt wie eine andere in Sudoku 2, dann wurde sie möglicherweise zu dieser zweiten Ziffer umbenannt.

Überschneiden sich nach einer Permutation jeweils die 9 Mengen in den möglichen Umbenennungen der vorher permutierten Ebene und der drei Elemente der aktuellen Ebene jeweils in mindestens einer Ziffer, werden die Umordnungen der nächsten Ebene durchlaufen, ansonsten wird die nächste Permutation auf die aktuelle Ebene angewendet.

Ein Beispiel dafür: Die Spaltenblöcke werden nach einer bestimmten Permutation umgeordnet. Dann wird überprüft, ob es Überschneidungen zwischen den möglichen Umbenennungen von jedem der einzelnen Sektoren **und** den Umbenennungen gibt, die zuvor beim Festlegen der Zeilenblock-Reihenfolge ermittelt wurden (ebenfalls durch Überschneidungen).

Ist das der Fall, so können als nächstes die Zeilen von Zeilenblock 1 permutiert werden, ansonsten muss das Programm die Spaltenblöcke anders umordnen.

In den letzten 3 Ebenen, also denen der Spalten, müssen die genauen Positionen der Ziffern berücksichtigt werden, weil diese nach einer Spaltenpermutation im entsprechenden Spaltenblock festgelegt sind. Wenn an einer Position in einer der jeweils 3 Zeilen die Ziffer von Sudoku 2 in den möglichen Umbenennungen für die Ziffer von Sudoku 1 ist, wird sie aus den möglichen Umbenennungen aller anderen Ziffern ausgeschlossen. Dieser Vergleich wird für alle Ziffern in den jeweils 3 Spalten gemacht, und wenn er für eine Ziffer nicht passt, wird zur nächsten Permutation übergegangen.

Wenn es auch in der letzten Ebene, also den drei Spalten von Spaltenblock 3, Überschneidungen der Umbenennungen gibt, wurde eine mögliche Lösung gefunden, die aus den aktuellen Permutationen für jede Ebene und einem beliebigen (in diesem Fall dem

ersten) Element des kartesischen Produktes aus den finalen möglichen Umbenennungen besteht.

Ermittlung der günstigsten Rotation des Sudokus:

Sudoku 1 wird mit jeder der vier Rotationen (0°, 90°, 180°, 270°) rotiert, es wird mit obiger Methode eine Lösung gesucht und das Sudoku wieder zurück rotiert. Am Ende gibt es entweder keine, zwei oder vier Lösungen (für die Lösung einer Rotation gibt es immer eine Lösung für deren „auf den Kopf gedrehte“ Version). Davon wird diejenige ausgesucht, die (inklusive der Rotation) am wenigsten Schritte hat.

Inversion einer Lösung, um Sudoku 2 zu 1 umzuformen:

Die Inversion einer Lösung geschieht nach folgenden Regeln:

Wenn die Lösung eine Rotation um 1-mal oder 3-mal 90° beinhaltet, wird die neue Rotation aus (n + 2) % 4 berechnet, wobei n die Anzahl an 90°-Rotationen im Uhrzeigersinn bezeichnet.

Falls vorher die Rotation vor den Umordnungen angewendet wurde, soll sie in dieser Lösung danach angewendet werden und umgekehrt.

Jede der 8 gespeicherten Umordnungen wird durch die Indexe von den ursprünglichen Permutationsindexen (0, 1 und 2) in der jeweiligen Umordnung ersetzt. Dadurch wird eine relative (den letzten Zustand als Ausgangspunkt betrachtende) Permutation rückgängig gemacht. Wurden z.B. die Zeilenblöcke nach [1, 2, 0] umgeordnet, so wird für 0, 1 und 2 nacheinander die Position in dieser Umordnung ermittelt. Das Ergebnis davon ist [2, 0, 1], was sich mit der „Hin-Umordnung“ zum Ursprungszustand [0, 1, 2] ergänzt.

Ein ähnliches Prinzip wird auf die Umbenennung der Ziffern angewandt. Für die Ziffern von 1 bis 9 wird nacheinander deren Index in der Umbenennungsliste angegeben (und mit 1 addiert), was sich auch hier mit der „Hin-Umbenennung“ zu gar keiner Umbenennung ergänzt.

Umsetzung

Um meinen Algorithmus umzusetzen, habe ich Python in der Version 3.10.7 verwendet.

Die einbezogenen Module sind folgende:

- ***numpy*** zum Erstellen, Verwalten und Untersuchen von n-dimensionalen Arrays
- ***os*** zum Auflisten der verfügbaren Dateien unter einem Pfad
- ***itertools*** zum effizienten Iterieren von Permutationen und kartesischen Produkten

Um ein Sudoku darzustellen und zu verwalten habe ich eine Klasse *sudoku* erstellt, die das Sudoku als 4D-Array der Form 3x3x3x3 speichert, alle Permutations-Methoden, eine Ausgabemethode und Methoden zur Erfassung von wichtigen Zifferndaten enthält.

Die Klasse *sudoku_solution* repräsentiert die „Lösung“, um ein Sudoku in ein anderes umzuformen. Ihre Attribute sind die Rotation, die Umordnungen und die Umbenennungen und ein boolean-Wert, der angibt, ob die Rotation vor den Umordnungen geschehen soll oder nicht. Sie hat eine Methode *inversed*, die nach obigen Regeln die invertierte Version der Instanz zurückgibt, eine Methode *total_steps*, die die Gesamtzahl an Schritten zählt und

zurückgibt und eine Methode *instructions*, die die eine String-Repräsentation der Lösungsdaten zurückgibt.

Die Klasse „sudoku“:

Im Folgenden erkläre ich eine der Umordnungs-Methoden der Klasse *sudoku*, ihre restlichen erklärungsbedürftigen Methoden und die statische Konstante *INDICES*.

Die Konstante INDICES:

```
INDICES = [list(tuple([i]) for i in range(3)), list((rb, cb) for rb in range(3) for cb in
range(3))]
for i in range(3):
    INDICES.append(list(zip(3*[i], 3*[slice(None)], range(3))))
for i in range(3):
    INDICES.append(list(zip(3*[slice(None)], 3*[i], 3*[slice(None)], range(3))))
```

Diese Konstante beinhaltet in einer Liste der Länge acht nacheinander jeweils eine Liste mit den 3 Indexen der Zeilenblöcke, den 9 Indexen der Sektoren, 3 Indexen der jeweils 3 Zeilen in den Zeilenblöcken und der jeweils 3 Spalten in den Spaltenblöcken.

Der Konstruktor:

```
def __init__(self, data: np.ndarray):
```

Im Konstruktor wird ein neues Array aus den Daten erstellt und eine objekteigene konstante Liste *FUNC_BY_DEPTH* initialisiert, die je nach Rekursionstiefe bei den iterierten Permutationen die zugehörige Umordnungsmethode beinhaltet, also zuerst die für die Zeilenblöcke, dann die für die Spaltenblöcke usw.

Die Methode output:

```
def output(self, dig_spaces = 1, dig_breaks = 0, sec_spaces = 4, sec_breaks = 1):
```

Gibt eine gut lesbare String-Repräsentation des Sudokus zurück. *dig_spaces* und *sec_spaces* sind die Parameter für die Anzahl von Leerzeichen als Separator zwischen den Ziffern / Sektoren, *dig_breaks* und *sec_breaks* für die Anzahl an Leeren Zeilen zwischen den Ziffern / Sektoren.

Die Methode arrange_rows:

```
def arrange_rows(self, row_block, indices):
```

Dies ist eine der vier Umordnungs-Methoden. Diese hier ordnet die Zeilen im Block *row_block* nach den Indexen *indices* um.

Die Methode do_solution:

```
def do_solution(self, solution: sudoku_solution):
```

Übernimmt ein *sudoku_solution*-Objekt und formt das Sudoku anhand seiner Daten um.

Die Methode digit_distribution:

```
def digit_distribution(self, area = None):
```

Gibt für das ggf. angegebene Teilarray *area* in einem Dictionary an, welche Ziffern wie oft vorkommen. Das Dictionary verwendet die verschiedenen Häufigkeiten als Keys und als Werte Sets, die jeweils alle Ziffern beinhalten, die die jeweilige Häufigkeit haben.

Die Methode digit_assignments:

```
def digit_assignments(self, target, indices = tuple()):
```

Im angegebenen Bereich *indices* werden die Ziffern-Häufigkeiten mit denen vom Sudoku *target* verglichen (Siehe ***digit_distribution***). Wenn die Keys der beiden Dictionaries, also die vorhandenen Häufigkeiten, gleich sind, werden diese iteriert. Für jede Ziffer mit der Häufigkeit *k* werden in einem Set alle Ziffern eingetragen, die an der Stelle *k* im Dictionary vom Sudoku *target* stehen. Das resultierende Dictionary wird zurückgegeben.

Die Methode positional_digit_assignments:

```
def positional_digit_assignments(self, target, assignments, indices):
```

Wird nach jeder Spaltenpermutation in *find_rearrangements* aufgerufen und vergleicht an den Stellen *indices* die Daten vom Sudoku mit denen des Sudokus *target*, wobei *assignments* die übergebene Liste mit bisherigen möglichen Umbenennungen für jede Ziffer ist. Wenn eine Ziffer z1 mit einer anderen z2 verglichen wurde, wird z2 für alle Ziffern außer z1 aus den möglichen Umbenennungen ausgeschlossen. Gibt None aus, wenn nicht alle Ziffern von den Umbenennungen her miteinander übereinstimmen, ansonsten die resultierenden, neuen möglichen Umbenennungen. Für weiteren Kontext siehe Abschnitt mit dem gelb markierten Anfang auf Seite 2.

Die Funktion „sudoku_from_lines“:

```
def sudoku_from_lines(lines):
```

Gibt anhand einer Liste von Zeilen aus einer der Textdateien ein Sudoku zurück. Dafür werden die Ziffern zu Integer-Werten umgewandelt und die Listen-Struktur passend zu einem Pseudo-Array umgeformt, welches an den Konstruktor von *sudoku* übergeben wird.

Die Funktion „sudokus_to_textfile“:

```
def sudokus_to_textfile(s1: sudoku, s2: sudoku, name, path = ""):
```

Übernimmt zwei Sudoku-Objekte und speichert diese im vorgegebenen Format, also Zeile für Zeile mit Leerzeichen als Separator zwischen den Ziffern, in einer Textdatei unter dem angegebenen Pfad und Namen.

Die Funktion „sudokus_from_textfile“:

```
def sudokus_from_textfile(path: str):
```

Öffnet die Textdatei im angegebenen Pfad, teilt sie in eine Zeilenliste für Sudoku 1 und eine für Sudoku 2 auf und gibt mithilfe von *sudoku_from_lines* zwei Referenzen zu den entsprechenden Sudoku-Objekten zurück.

Die Funktion „sudokus_from_input“:

```
def sudokus_from_input(path = ""):
```

Erwartet die Eingabe des Namens einer Textdatei, bis eine Datei mit dem angegebenen Namen unter *path* existiert. Die dort gespeicherten Sudokus werden mithilfe von *sudokus_from_textfile* zurückgegeben.

Kernfunktion 1: find_rearrangements:

```
def find_rearrangements(s1: sudoku, s2: sudoku, depth = 0,
assignments = list(set(range(1, 10)) for _ in range(9)), arr = None):
```

Diese rekursive Funktion ist dafür zuständig, Umordnungen und die zugehörigen möglichen Umbenennungen zu finden, mit denen das Sudoku *s1* zu *s2* umgeformt werden kann, und zwar nach dem Prinzip, das im ersten Teil des Lösungs-Algorithmus auf Seite 2 beschrieben wurde. Das Sudoku mithilfe der für die aktuelle Tiefe in *s1.FUNC_BY_DEPTH* referenzierten Funktion mit allen Permutationen von 0, 1 und 2 nacheinander umgeordnet, bis in der Überschneidung der Umbenennungen in den 3 permutierten Bereichen bzw. in den ermittelten positionellen Umbenennungen kein leeres Set-Objekt vorhanden ist.

Dann gibt es für diese Permutation gültige Umbenennungen und die Funktion ruft sich selbst mit der um 1 erhöhten Tiefe *depth* und mit der aktuellen Permutation ergänzten Umordnungen auf. Wenn dieser Aufruf eine Rückgabe außer None hat, wird er zurückgegeben. Falls in der letzten Tiefe eine Rückgabe der finalen Umordnungen und Umbenennungen erfolgt, wird diese also durch alle Funktionsebenen bis ganz nach oben weitergegeben.

Kernfunktion 2: related_sudoku_check:

```
def related_sudoku_check(s1: sudoku, s2: sudoku):
```

Rotiert das Sudoku *s1* um 0° und 90° und sucht jeweils mit *find_rearrangements* nach Lösungen. Wenn es für eine der Rotationen eine Lösung gibt, wird diese zu einer Liste *solutions* hinzugefügt und auch die (garantiert vorhandene) Lösung für diese Rotation plus 180° ermittelt und zur Liste hinzugefügt.

Als finale Lösung wird aus der Liste die ausgewählt, die am wenigsten Schritte hat (bzw. die erste der Lösungen mit den wenigsten Schritten), was mithilfe von *total_steps()* aus *sudoku_solution* ermittelt wird. Diese und ihre mit *inverse()* ermittelte Rücklösung werden durch ihre Methode *instructions()* jeweils als Anweisungen ausgegeben.

Wenn es keine Lösungen gibt, wird der Benutzer informiert, dass die Sudokus keine Kopien voneinander sind.

Zusätzliche Programme

Neben dem Hauptprogramm ***Sudokopie.py*** gibt es noch eine Version davon ohne Kommentare. Außerdem habe ich ein Programm ***random_sudoku.py*** geschrieben, mit dessen Hilfe zufällige Sudokus erstellt und zufällig umgeformt werden können. Es lässt sich ein bezugsloses Sudoku-Paar erstellen oder eines, das ineinander umformbar ist.

Die vom Benutzer auszuführenden Programme (die nicht nur Definitionen beinhalten), sind mit dem Wort „Run“ am Anfang gekennzeichnet. Mit ***Run_From_Input.py*** werden die Sudokus aus einer in der Konsole mit dem Namen angegebenen Textdatei überprüft (Angabe ohne „.txt“ am Ende).

In ***Run_Random.py*** wird ein Sudoku-Paar erstellt, das zu einer Wahrscheinlichkeit von 50% ineinander umformbar ist, mit *related_sudoku_check* überprüft und auf Wunsch des Benutzers in einer Textdatei mit dem Namen zufall0, zufall1 usw. gespeichert.

Genaueres ist in den ersten Zeilen der beiden Programme angegeben.

Beispiele

Hier stehen die Programmausgaben von *Run_From_Input.py* für die vorgegebenen Beispieldateien und für einige eigene Dateien. Nur für das erste Beispiel beinhalte ich auch die anfängliche Ausgabe der Sudokus.

Programmausgabe sudoku0.txt:

Sudoku 1:

```
6 0 0  0 8 0  0 0 1
0 9 0  4 0 0  6 0 0
0 0 0  0 9 0  0 0 2

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

0 0 0  0 7 0  0 0 0
0 0 6  8 0 3  9 7 0
0 0 0  0 0 0  0 8 0
```

Sudoku 2:

```
0 0 9  0 0 4  0 0 6
0 6 0  8 0 0  1 0 0
0 0 0  9 0 0  2 0 0

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

0 0 0  7 0 0  0 0 0
0 0 0  0 0 0  0 8 0
6 0 0  0 3 8  0 7 9
```

Diese Sudokus sind Kopien voneinander.

Umformungen von 1 zu 2:

-> Zeilen 1-3 umordnen zu: 2 1 3

-> Zeilen 7-9 umordnen zu: 7 9 8

-> Spalten 1-3 umordnen zu: 3 1 2

-> Spalten 4-6 umordnen zu: 5 6 4
-> Spalten 7-9 umordnen zu: 9 8 7

Umformungen von 2 zu 1:

-> Zeilen 1-3 umordnen zu: 2 1 3
-> Zeilen 7-9 umordnen zu: 7 9 8
-> Spalten 1-3 umordnen zu: 2 3 1
-> Spalten 4-6 umordnen zu: 6 4 5
-> Spalten 7-9 umordnen zu: 9 8 7

Programmausgabe sudoku1.txt:

Diese Sudokus sind Kopien voneinander.
Umformungen von 1 zu 2:

-> Das Sudoku 1 mal um 90° im Uhrzeigersinn drehen

-> Zeilenblöcke umordnen zu: 2 3 1
-> Spaltenblöcke umordnen zu: 2 3 1

Umformungen von 2 zu 1:

-> Zeilenblöcke umordnen zu: 3 1 2
-> Spaltenblöcke umordnen zu: 3 1 2

-> Das Sudoku 3 mal um 90° im Uhrzeigersinn drehen

Programmausgabe sudoku2.txt:

Diese Sudokus sind Kopien voneinander.
Umformungen von 1 zu 2:

-> Spaltenblöcke umordnen zu: 3 2 1
-> Zeilen 4-6 umordnen zu: 4 6 5

-> Ziffern 1 bis 9 umbenennen zu: 2 3 4 5 6 7 8 9 1

Umformungen von 2 zu 1:

-> Spaltenblöcke umordnen zu: 3 2 1

-> Zeilen 4-6 umordnen zu: 4 6 5

-> Ziffern 1 bis 9 umbenennen zu: 9 1 2 3 4 5 6 7 8

Programmausgabe sudoku3.txt:

Diese Sudokus sind keine Kopien voneinander.

Programmausgabe sudoku4.txt:

Diese Sudokus sind Kopien voneinander.
Umformungen von 1 zu 2:

-> Das Sudoku 3 mal um 90° im Uhrzeigersinn drehen

-> Zeilenblöcke umordnen zu: 3 1 2
-> Spaltenblöcke umordnen zu: 2 1 3
-> Spalten 4-6 umordnen zu: 4 6 5
-> Spalten 7-9 umordnen zu: 7 9 8

-> Ziffern 1 bis 9 umbenennen zu: 4 8 1 9 2 5 7 3 6

Umformungen von 2 zu 1:

-> Zeilenblöcke umordnen zu: 2 3 1
-> Spaltenblöcke umordnen zu: 2 1 3
-> Spalten 1-3 umordnen zu: 1 3 2
-> Spalten 7-9 umordnen zu: 7 9 8

-> Das Sudoku 1 mal um 90° im Uhrzeigersinn drehen

-> Ziffern 1 bis 9 umbenennen zu: 3 5 8 1 6 9 7 2 4

Programmausgabe zufall0.txt:

Diese Sudokus sind Kopien voneinander.
Umformungen von 1 zu 2:

-> Das Sudoku 3 mal um 90° im Uhrzeigersinn drehen

-> Spaltenblöcke umordnen zu: 3 2 1
-> Zeilen 7-9 umordnen zu: 8 9 7

Umformungen von 2 zu 1:

-> Spaltenblöcke umordnen zu: 3 2 1
-> Zeilen 7-9 umordnen zu: 9 7 8

-> Das Sudoku 1 mal um 90° im Uhrzeigersinn drehen

Programmausgabe diagonale_rotiert.txt:

Diese Sudokus sind Kopien voneinander.
Umformungen von 1 zu 2:

-> Das Sudoku 1 mal um 90° im Uhrzeigersinn drehen

Umformungen von 2 zu 1:

-> Das Sudoku 3 mal um 90° im Uhrzeigersinn drehen

Programmausgabe 2_gleiche_sudokus.txt:

Diese Sudokus sind gleich, keine Umformungen nötig.

Programmausgabe nur_umbenannt.txt:

Diese Sudokus sind Kopien voneinander.
Umformungen von 1 zu 2:

-> Ziffern 1 bis 9 umbenennen zu: 2 8 5 9 7 4 3 1 6

Umformungen von 2 zu 1:

-> Ziffern 1 bis 9 umbenennen zu: 8 1 7 6 3 9 5 2 4

Quellcode

Dies ist die unkommentierte Version des Quellcodes von *Sudokopie.py*:

```
import numpy as np
from os import listdir
from itertools import permutations, product

class sudoku_solution:
    def __init__(self, rotation: int, rearrangements, reassignments, rotation_first =
    True):
```

```
        self.rotation = rotation
        self.rearrangements = list(rearrangements)
        self.reassignments = list(reassignments)
        self.rotation_first = rotation_first

    def get_data(self):
        return self.rotation, self.rearrangements.copy(), \
            self.reassignments.copy(), self.rotation_first

    def inversed(self):
        rot, rearr, reass, rot_first = self.get_data()
        if rot in (1, 3):
            rot = (rot + 2) % 4

        rearr_row_blocks = rearr[0] = reverse_perm(rearr[0])
        rearr_col_blocks = rearr[1] = reverse_perm(rearr[1])
        rearr[2:5] = (rearr[i+2] for i in rearr_row_blocks)
        rearr[5:8] = (rearr[i+5] for i in rearr_col_blocks)
        for i in range(2, 8):
            rearr[i] = reverse_perm(rearr[i])

        reass = list(map(lambda i: reass.index(i) + 1, range(1, 10)))

        return sudoku_solution(rot, rearr, reass, not rot_first)

    def total_steps(self):
        nperms = 0
        if self.rotation:
            nperms += 1

        nperms += 8 - self.rearrangements.count((0,1,2))

        if self.reassignments != list(range(1, 10)):
            nperms += 1
        return nperms

    def instructions(self):
        rot, rearr, reass, rot_first = self.get_data()
        rearr_el1 = ["Zeilenblöcke", "Spaltenblöcke"] + 3*["Zeilen"] + 3*["Spalten"]
        rearr_el2 = 2*[""] + 2*[" 1-3", " 4-6", " 7-9"]
        instructions = []
        rot_instruction = ""
        rearr_instructions = []
        reass_instruction = ""

        if rot:
            rot_instruction = f"-> Das Sudoku {rot} mal um 90° im Uhrzeigersinn drehen"
            instructions.append(rot_instruction)

        for i, arr in enumerate(rearr):
            if arr != (0,1,2):
                inst = rearr_el1[i]
```

```
                inst += rearr_el2[i]
                inst += " umordnen zu: "
                if i in (0, 1, 2, 5):
                    inc = 1
                elif i in (3, 6):
                    inc = 4
                else:
                    inc = 7
                inst += " ".join(map(lambda i: str(i + inc), arr))
                rearr_instructions.append("-> " + inst)
        if rearr_instructions:
            instructions.append("\n".join(rearr_instructions))

        if reass != list(range(1, 10)):
            reass_instruction = "-> Ziffern 1 bis 9 umbenennen zu: " + " ".join(str(i) for
            i in reass)
            instructions.append(reass_instruction)

        if rot and rearr_instructions and not rot_first:
            instructions[:2] = instructions[1::-1]

        for i in range(len(instructions) - 1):
            instructions[i] += "\n"
        return "\n".join(instructions)

class sudoku:
    INDICES = [list(tuple([i]) for i in range(3)), list((rb, cb) for rb in range(3) for cb
    in range(3))]
    for i in range(3):
        INDICES.append(list(zip(3*[i], 3*[slice(None)], range(3))))
    for i in range(3):
        INDICES.append(list(zip(3*[slice(None)], 3*[i], 3*[slice(None)], range(3))))

    def __init__(self, data: np.ndarray):
        self.data = np.array(data)
        self.FUNC_BY_DEPTH = (self.arrange_row_blocks, self.arrange_col_blocks) + \
        tuple(3 *[ self.arrange_rows]) + tuple(3 * [self.arrange_cols])

    def __eq__(self, other):
        return np.array_equal(self.data, other.data)

    def output(self, dig_spaces = 1, dig_breaks = 0, sec_spaces = 4, sec_breaks = 1):
        string = []
        for row_block in range(3):
            for row in range(3):
                substring = []
                for sector in range(3):
                    string_array = np.char.mod("%i", self.data[row_block, sector, row, :])
                    substring.append((dig_spaces * " ").join(string_array))

                string.append((sec_spaces * " ").join(substring))
                if (row < 2) and dig_breaks + 1:
```

```
                string.append((dig_breaks + 1) * "\n")
            if (row_block < 2) and sec_breaks + 1:
                string.append((sec_breaks + 1) * "\n")
        return "".join(string)

    def arrange_row_blocks(self, indices):
        self.data[:] = np.array(list(map(lambda i: self.data[i], indices)))

    def arrange_col_blocks(self, indices):
        self.data[:, 0], self.data[:, 1], self.data[:, 2] = \
        np.array(list(map(lambda i: self.data[:, i], indices)))

    def arrange_rows(self, row_block, indices):
        self.data[row_block, :, range(3)] = \
        np.array(list(map(lambda i: self.data[row_block, :, i], indices)))

    def arrange_cols(self, col_block, indices):
        self.data[:, col_block, :, range(3)] = \
        np.array(list(map(lambda i: self.data[:, col_block, :, i], indices)))

    def rotate_90_clockwise(self, n=1):
        self.data = np.rot90(self.data, n, (1,0))
        self.data = np.rot90(self.data, n, (3,2))

    def rename_digits(self, new_digits):
        t_list = []
        for n in range(1, 10):
            t_list.append(self.data == n)
        for t, n in zip(t_list, new_digits):
            self.data[t] = n

    def do_solution(self, solution: sudoku_solution):
        rot, rearr, reass, rot_first = solution.get_data()

        if rot and rot_first:
            self.rotate_90_clockwise(rot)
        for i, p in enumerate(rearr):
            if p != (0,1,2):
                if i > 1:
                    self.FUNC_BY_DEPTH[i] ((i - 2) % 3, p)
                else:
                    self.FUNC_BY_DEPTH[i] (p)
        if rot and not rot_first:
            self.rotate_90_clockwise(rot)

        if reass != tuple(range(1, 10)):
            self.rename_digits(reass)

    def digit_distribution(self, area = None):
        if type(area) not in [np.ndarray, np.int32]:
            area = self.data
        abs_n_dig = dict()
```

```
        for n in range(1, 10):
            occ = np.count_nonzero(area == n)
            if occ not in abs_n_dig.keys():
                abs_n_dig[occ] = set()
            abs_n_dig[occ].add(n)

        return abs_n_dig

    def digit_assignments(self, target, indices = tuple()):

        assignments = list(set(range(1, 10)) for _ in range(9))
        indices = tuple(indices)
        dist_self = self.digit_distribution(self.data[indices])
        dist_target = self.digit_distribution(target.data[indices])

        if dist_self.keys() != dist_target.keys():
            return None

        for k, v in dist_self.items():
            for i in v:
                assignments[i-1] = dist_target[k].copy()

        return assignments

    def positional_digit_assignments(self, target, assignments, indices):
        for i in indices:
            data_self = np.reshape(self.data[i], 9)
            data_target = np.reshape(target.data[i], 9)
            for d1, d2 in zip(data_self, data_target):
                if d1 != 0:
                    if d2 not in assignments[d1 - 1]:
                        return None
                    else:
                        for i in range(9):
                            if i != d1 - 1:
                                assignments[i].discard(d2)
        return assignments

    def digit_assignment_product(assignments):
        return filter(lambda i: len(set(i)) == len(i), product(*assignments))

def sudoku_from_lines(lines):
    for i in range(len(lines)):
        int_list = [int(n) for n in lines[i].split(" ")]
        lines[i] = [int_list[:3], int_list[3:6], int_list[6:9]]
    row_blocks = [lines[:3], lines[3:6], lines[6:9]]
    for i in range(len(row_blocks)):
        row_blocks[i] = list(zip(*row_blocks[i]))
    return sudoku(row_blocks)

def sudokus_to_textfile(s1: sudoku, s2: sudoku, name, path = ""):
    if path and path[-2:] != "\\":
```

```
        path += "\\"
    with open(path + name + ".txt", 'w', encoding = "utf-8-sig") as file:
        for s in (s1, s2):
            sudoku_string = []
            for i_block in range(3):
                for i_row in range(3):
                    row = np.reshape(s.data[i_block, :, i_row], 9)
                    row_string = " ".join(str(i) for i in row)
                    sudoku_string.append(row_string)
            file.write("\n".join(sudoku_string))
            if s is s1:
                file.write("\n\n")

def sudokus_from_textfile(path):
    with open(path, 'r', encoding = "utf-8-sig") as file:
        lines = file.readlines()
        lines = list(l.strip() for l in lines)
        lines = list(filter(lambda l: l != "", lines))

        l1 = lines[:9]
        l2 = lines[9:]
        return sudoku_from_lines(l1), \
               sudoku_from_lines(l2)

def sudokus_from_input(path = ""):
    if path:
        if path[-2:] != "\\":
            path += "\\"
        files = listdir(path)
    else:
        files = listdir()

    filename = input("Welche Textdatei soll verwendet werden? ").strip()
    while (filename + ".txt") not in files:
        print(f"Die Datei \"{filename + '.txt'}\" existiert nicht unter dem angegebenen
        Pfad.\n")
        filename = input("Welche Textdatei soll verwendet werden? ").strip()

    return sudokus_from_textfile(path + filename + ".txt")

def reverse_perm(p: tuple):
    return tuple(map(lambda i: p.index(i), range(3)))

def find_rearrangements(s1: sudoku, s2: sudoku, depth = 0,
assignments = list(set(range(1, 10)) for _ in range(9)), arr = None):
    if arr is None:
        arr = []
    IND = sudoku.INDICES
    PERMS = s1.FUNC_BY_DEPTH
    if depth < 2:
        add_arg = tuple()
    else:
```

```
        add_arg = tuple([(depth - 2) % 3])
    for p in permutations(range(3)):
        if p != (0,1,2):
            PERMS[depth] (*add_arg, p)

        if depth < 5:
            next_assignments = []
            for indices in IND[depth]:
                next_assignments.append(s1.digit_assignments(s2, indices))
            assignment_overlap = None
            if not None in next_assignments:
                assignment_overlap = list(set.intersection(*sets) for sets in
                zip(assignments, *next_assignments))
        else:
            assignment_overlap = s1.positional_digit_assignments(s2, assignments,
            IND[depth])

        if assignment_overlap:
            if not set() in assignment_overlap:
                if depth < 7:
                    next_depth_data = find_rearrangements(s1, s2, depth + 1,
                    assignment_overlap.copy(), arr.copy() + [p])
                    if next_depth_data:
                        PERMS[depth] (*add_arg, reverse_perm(p))
                        return next_depth_data
                else:
                    arr.append(p)
                    PERMS[depth] (*add_arg, reverse_perm(p))
                    return arr, \
                        next(sudoku.digit_assignment_product(assignment_overlap))

        PERMS[depth] (*add_arg, reverse_perm(p))

def related_sudoku_check(s1: sudoku, s2: sudoku):
    print("\nLösungen werden ermittelt...\n")
    info = ["Sudoku 1:\n", s1.output(), "\nSudoku 2:\n", s2.output()]
    if s1 == s2:
        info.append("\nDiese Sudokus sind gleich, keine Umformungen nötig.")
        return "\n".join(info), None, None

    solutions: list[sudoku_solution] = []
    data_rot0 = find_rearrangements(s1, s2)
    if data_rot0:
        solutions.append(sudoku_solution(0, *data_rot0))
        s1.rotate_90_clockwise(2)
        solutions.append(sudoku_solution(2, *find_rearrangements(s1, s2)))
        s1.rotate_90_clockwise(2)

    s1.rotate_90_clockwise()
    data_rot1 = find_rearrangements(s1, s2)
    s1.rotate_90_clockwise(3)
    if data_rot1:
```

```
        solutions.append(sudoku_solution(1, *data_rot1))
        s1.rotate_90_clockwise(3)
        solutions.append(sudoku_solution(3, *find_rearrangements(s1, s2)))
        s1.rotate_90_clockwise()

    if solutions:
        nsteps_each = tuple(map(lambda s: s.total_steps(), solutions))

        solution: sudoku_solution = solutions[nsteps_each.index(min(nsteps_each))]
        solution_back: sudoku_solution = solution.inversed()

        info.append("\nDiese Sudokus sind Kopien voneinander.\nUmformungen von 1 zu 2:\n")
        info.append(solution.instructions())
        info.append("\nUmformungen von 2 zu 1:\n")
        info.append(solution_back.instructions())
    else:
        info.append("\nDiese Sudokus sind keine Kopien voneinander.")
        solution, solution_back = None, None

    return "\n".join(info), solution, solution_back

def check_solutions(s1: sudoku, s2: sudoku, solution, solution_back):
    print("\nLösungen werden geprüft:")
    if not (solution and solution_back):
        print("Keine Lösungen vorhanden")
        return

    s1.do_solution(solution)
    if s1 == s2:
        print("Lösung von 1 zu 2 funktioniert")
    else:
        print("Lösung von 1 zu 2 funktioniert nicht")
    s1.do_solution(solution_back)

    s2.do_solution(solution_back)
    if s1 == s2:
        print("Lösung von 2 zu 1 funktioniert.")
    else:
        print("Lösung von 2 zu 1 funktioniert nicht")
```