DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Duale Hochschule Baden-Württemberg Mannheim

Seminararbeit

Analyse der Lösbarkeit von Instanzen des 15-Puzzzles mit Implementierung

Studiengang Informatik

Studienrichtung angewandte Informatik

Verfasser(in):	Kai Fischer, Max Stubenbord
Matrikelnummer:	3683691, 5379506
Kurs:	TINF18AI1
Studiengangsleiter:	Prof. Dr. Holger Hofmann
Wissenschaftliche(r) Betreuer(in):	Prof. Dr. Karl Stroetmann
Bearbeitungszeitraum:	24.03.2021 – 11.06.2021

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Titel "*Analyse der Lösbarkeit von Instanzen des 15-Puzzzles mit Implementierung*" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum Kai Fischer, Max Stubenbord

Danksagung

Hier können Sie eine Danksagung schreiben.

Kurzfassung (Abstract)

Hier können Sie die Kurzfassung (engl. Abstract) der Arbeit schreiben. Beachten Sie dabei die Hinweise zum Verfassen der Kurzfassung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

AD	Archiv für Diplomatik, Schriftgeschichte, Siegel- und Wappenkunde
BMBF	Bundesministerium für Bildung und Forschung
DHBW	Duale Hochschule Baden-Württemberg
ECU	European Currency Unit
EU	Europäische Union
RDBMS	Relational Database Management System

1 Einleitung

1.1 Geschichte

Die erste bekannte Erscheinung des heute so bekannten Puzzels ist in den 1870'er Jahren in Amerika aufgetreten. Bisher ist man davon ausgegangen, dass der Erfinder des Puzzels der Amerikaner Sam Loyd ist, jedoch ist nach einer Untersuchung von Jerry Slocum and Dieter Gebhardt Sam Loyd gar nicht der echte Erfinder des 15-Puzzels [**anchor-puzzle:book**, **the-15-puzzle:online**]. Demnach hat Sam Loyd die Idee nur gut vermarktet und sich selbst dadurch in die Öffentlichkeit gestellt [**wiki-15-puzzle:online**].
`zitat? -> woher die information` Nichtsdestotrotz hat Sam Loyd es geschafft die Aufmerksamkeit der Öffentlichkeit auf das Puzzel zu lenken und somit das Interesse vieler geweckt.
Das Ziel des Puzzels ist es 15 Puzzelsteine numeriert von 1-15 auf einer 4 x 4 Ebene durch Verschiebung in seine Ursprüngliche sortierte Form zurückzubringen (vgl. Abb.1.1).

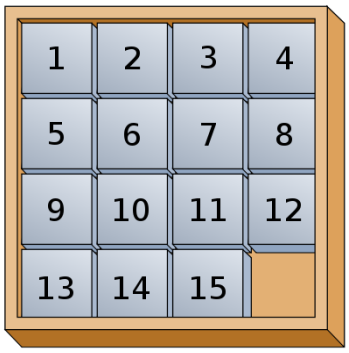

Abbildung 1.1: Zielzustand des 15 Puzzels

Der schnelle Ruhm des Puzzels ergab sich nun daraus, dass Sam Loyd ein Preisgeld von $\$1000$ für denjenigen ausschrieb, der sein Puzzel lösen kann. Darauf folgte ein öffentlicher Ansturm auf das Puzzel, welches als „14-15 Puzzel" bekannt wurde, da lediglich die 14 und 15 vertauscht wurden. (Vgl. Abb.1.2)

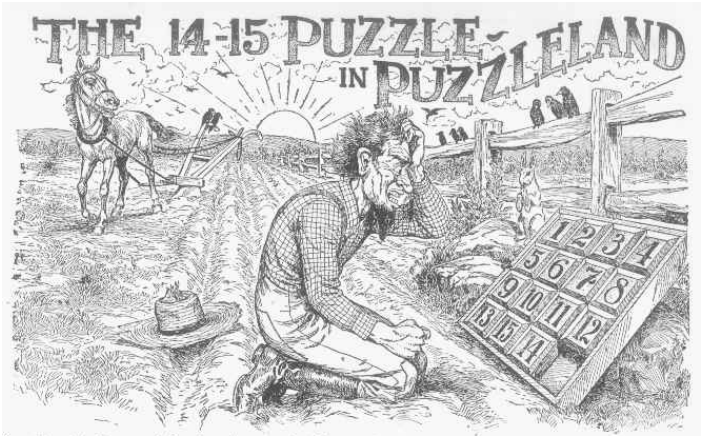

Abbildung 1.2: Illustration des 14-15 Puzzels von Sam Loyd (1914)
Quelle: [**loyd-cyclopedia:book**]

Allerdings zeigte sich im Nachhinein, dass das Puzzel unlösbar ist, da es eine Transformation von einer geraden zu einer ungeraden Permutation erfordert [**wiki-15-puzzle:online**]. Die Unlösbarkeit des Puzzels wurde erstmals von Wm. Woolsey Johnson und William E. Story im Jahre 1879 bei einer Veröffentlichung des American Journal of Mathematics bewiesen. [**ajom-notes-15-puzzle:article**] Somit gewann keiner das Preisgeld von $\$1000$. Das 15-Puzzel wurde daraufhin auch bei einer Illustration mit dem Titel „The Great Presidential Puzzle“ der US-Präsidentschaftswahl 1880 referenziert. Zu finden ist dies in der United States Library of Congress's Prints and Photographs division unter der Digitalen ID ppmsca.15782 [**presidental-puzzle:online**].

1.2 Aufgabenstellung und Abgrenzung

Hätte man damals bewiesen, dass das Puzzel aus der Einleitung von Sam Loyd nicht lösbar ist, wäre wohl vielen Menschen nächtelange Verzweiflung erspart geblieben.
Nun stellt sich Frage wann eine Instanz des Puzzels lösbar ist. Allgemein gesprochen ist eine Instanz eines 15 Puzzels dann lösbar, falls es eine Sequenz von zulässigen Zügen gibt, welche vom Startzustand zum Zielzustand führen.
Diese Annahme gilt als Zutreffend für genau die hälfte aller möglichen $16! \approx 2 \cdot 10^{13}$ Puzzel Kombinationen [**sliding-piece-puzzels:book**, **solving-15-puzzle-lvi:article**]. Das Puzzel Problem ist ein klassischer Anwendungsfall wenn es um Modellierung von Algorithmen mit Heuristiken geht. Üblicherweise werden Algorithmen wie *A** oder *IDA** zur Lösung solcher Heurisiken genutzt [**wiki-15-puzzle:online**, **solving-15-puzzle-lvi:article**, **depth-first-id:article**].

Im Rahmen dieser Arbeit geht es nun darum bei gegebener Puzzelinstanz vorherzusagen, ob diese als lösbar oder unlösbar gilt. Was bei dieser Arbeit nicht betrachtet wird, sind die verschiedenen Algorithmen wie *A** oder auch *IDA**. Diese werden ausfürhlich im zugehörigen Vorlesungsskript in den Sektionen ***2.7*** *A* Search* und ***2.10*** *A*-IDA* Search* erläutert. Lediglich werden diese genutzt um die als lösbar identifizierten Puzzels anschließend auch zu lösen.

1.3 Vorgehen

Das Vorgehen der Arbeit ist dabei wiefolgt:

2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen und Gedanken für die Umsetzung im nächsten Kapitel 3 vorgestellt. Die Lösung und das Vorgehen basieren maSSgeblich auf dem Beitrag „Why is this Puzzle Impossible? - Numberphile“ von Herrn Steven Bradlow zur Lösbarkeit des „14-15 puzzles“ aus der Einleitung auf dem Youtube Kanal „Numberphile“ [**Unsolvable-14-15-Numberphile-YT:online**].

2.1 Puzzle zu Listen wandeln

Der Lösungsansatz aus [**Unsolvable-14-15-Numberphile-YT:online**] basiert auf Permutationen. Um 4x4 Puzzle besser auf Permutationen untersuchen zu können, werden die Puzzle als Listen von Zahlen dargestellt. Dazu werden die Inhalte der Zellen des Puzzles Zeilenweise hintereinander in eine Liste geschrieben. Die Leerstelle, auch als „blank“ beschrieben, wird dabei als Zahl „0“ interpretiert. Der Zustand des Puzzles aus Abb.2.1

	1	2	3
4	5	6	8
14	7	11	10
9	15	12	13

Abbildung 2.1: Beispiel Zustand eines 4x4-Puzzles

wird als Liste aus Zahlen wie folgt dargestellt:

$$State = \{0, 1, 2, 3, 4, 5, 6, 8, 14, 7, 11, 10, 9, 15, 12, 13\}$$

Wichtig ist bei der Betrachtung der lösbaren Puzzle und des Vorgehen der Lösung aus [**Unsolvable-14-15-Numberphile-YT:online**] aber auch anderer Verfügbarer Quellen wie [**bibid**] `TODO: die ganzen anderen Quellen mal eintragen`. Die Bezeichnung der Leerstelle oder die Art der Konvertierung eines Puzzles zu einer Liste variiert. Die meisten Lösungen sehen den Zielzustand aus Abb.2.2 vor, wobei die Leerstelle dann die Nummer „16“ trägt. Um mit den Darstellungen von Herrn Stroetmann aus dem Vorlesungsskript `TODO :strotmann zitat` übereinzustimmen, wird der Zielzustand aus Abb.2.3 angestrebt, bei dem die Leerstelle die Nummer „0“ trägt.

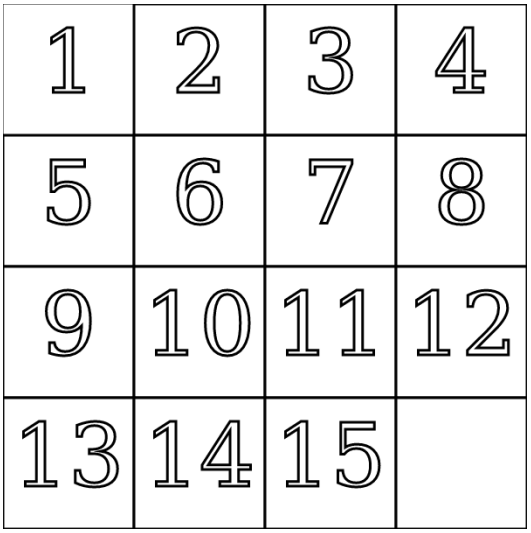

Abbildung 2.2: Häufig verwendeter Zielzustand eines 4x4-Puzzles

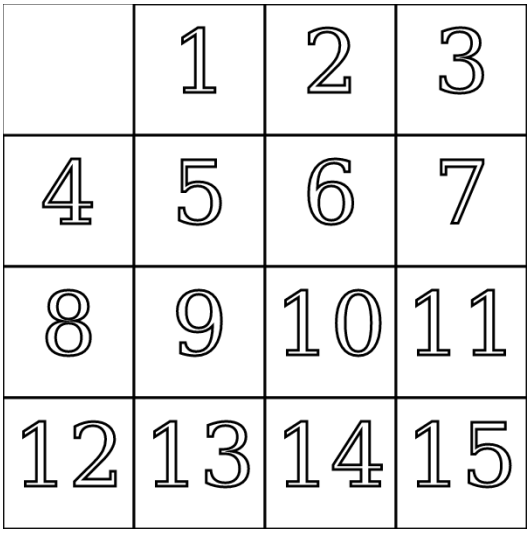

Abbildung 2.3: Verwendeter Zielzustand eines 4x4-Puzzles im Skript von Herrn Stroetmann `TODO:StroetmannSkritp`

2.2 Zustände als Permutationen

Die Grund-Idee der Lösbarkeitsüberprüfung von Bradlow basiert auf dem Vergleichen der Paritäten zwischen benötigten Transpositionen und benötigter Züge zum Verrücken der Leerstelle. Im Folgenden wird die Idee aus seinem Beitrag [**Unsolvable-14-15-Numberphile-YT:online**] zusammengefasst vorgestellt.

Die Vorstellung des Verfahrens beginnt Bradlow damit die Puzzle wie in der vorherigen Sektion in Zahlen Reihen zu wandeln. Durch das Vertauschen von 2 beliebigen Zahlen aus

dieser Zahlenreihe entsteht eine Permutation der vorherigen Reihe. Auf diese Weise sei jeder möglicher Zustand in dem sich das Puzzle befinden kann als eine Permutation einer zugrundeliegenden aufsteigend sortieren Zahlenreihe zu sehen. Für diese Permutationen beschreit er die folgenden zwei Eigenschaften:

E1 Jede dieser Permutationen lässt sich nur durch das Nutzen von Transpositionen, also dem Vertauschen von Zwei Elementen aus der Liste, während die restlichen gleich bleiben, in eine andere Permutation überführen [**Unsolvable-14-15-Numberphile-YT:online**].

E2 Die Anzahl der Schritte, die für das Überführen einer Permutation in eine andere Permutation mithilfe von Transpositionen benötigt wird, ist nicht Festgelegt, aber die Parität dieser Zahl ist fest [**Unsolvable-14-15-Numberphile-YT:online**].

Basierend auf diesen Eigenschaften fährt er fort, dass die Parität der Anzahl notwendiger Transpositionen durch das zielgerichtete Tauschen der Zahlen, mit dem ziel einer aufsteigen sortieren Zahlenreihe, ermittelt werden kann. Wie in der vorherigen Sektion beschrieben, sieht auch Bradlow die sortierte Zahlenreihe

$$Goal_{Bradlow} = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16$$

als Abbild des Zielzustandes vor. Daher ist in seinem Vortrag das Ziel der Transpositionen die aufsteigend sortierte Liste. Durch E1 ist sicher gestellt, dass die sortiere Liste erreicht werden kann. E2 begründet, dass die Parität der für die Sortierung notwendigen Transpositionen mit der Parität der notwendigen Züge, wenn das Puzzle lösbar ist, übereinstimmt, obwohl die Regeln nicht eingehalten wurden.

Im zweiten Schritt der Lösbarkeitsüberprüfung ermittelt Bradlow die Parität der notwendigen legalen Züge zum Überführen der Leerstelle aus dem Startzustand zum Zielzustand. Dabei ist ebenfalls jeder einzelne Zug als eine Transposition zu sehen.

Lösbar ist ein Puzzle nach Bradlow dann, wenn die ermittelten Paritäten übereinstimmen. Denn nach E2 sind die Paritäten festgelegt und können nicht verändert werden. Die Parität der Leerstellen Bewegung ist Legal, während Parität der notwendigen zur Sortierung notwendigen Züge nur hypothetisch sind. stimmen die Paritäten nicht über ein existiert nach E2 keine Möglichkeit beide Ziele, also die Sortierung ebenso wie das einhalten Zugregeln einzuhalten.

In der nächsten Sektion wird dieses Verfahren an einem Beispiel schrittweise durchgegangen.

Für die Sektion 3.1, soll noch erwähnt werden, dass sich die Leerstelle nur horizontal und vertikal „bewegen“ darf und sich die Anzahl der Züge so mithilfe der Manhattan-Distanz ermitteln lässt.

2.3 Kontext Vorlesung + Abgrenzung

2.4 Sortieralgorithmen

Mit Ordnung?!

3 Implementation

3.1 Umsetzung

`Der vollständige Quellcode ist im Anhang unter A.1 zu finden.`

Zur Umsetzung wird zu erst eine Datenstruktur definiert, in welcher die Instanzen des 15-Puzzels ausgewertet werden. Hierfür werden Tupel von Tupeln verwendet. Anschließend ist die gernerelle Idee, wie auch schon in `kais abschnitt?` vorgestellt zu schauen, ob die Anzahl der Permutationen bei der Datenstruktur als 1-Dimensionale Liste und der Abstand des Blank-Feldes vom Start- zum Zielzustand die gleiche Parität besitzen. Ist diese gleich, so ist das Puzzel lösbar, vice versa unlösbar.
Spannend ist hier allerdings, dass dies kein Indikator für die Komplexität der Lösbarkeit gibt. So kann ein lösbares Puzzel mit den zur Verfügung stehenden Algorithmen wie *A** oder *IDA** nicht in angemessener Zeit gelöst werden.
Das weitere Vorgehen der Implementierung ist weitesgehend die Bearbeitung von kleinen Teilproblemen, welche sich aus dem eben genannten Vorgehen ergeben.
So betrachten wir zunächst die Anzahl der Permutationen, welche sich in einer Puzzelinstanz befinden. Um diese zu berechnen wird als Datenstruktur eine Liste mit der Dimension 1 benötigt. Anschließend muss die Liste durch tauschen der Elemente sortiert werden. Praktischerweise ist der Lösungszustand des Puzzels so definiert, dass der Index innerhalb der Liste und der Wert des zugehörigen Elementes identisch sind. Somit kann für jeden Index der entsprechende Wert innerhalb der Liste gesucht werden, sodass eine minimale Anzahl von *swaps* durchgeführt werden.
Die Anzahl der getätigten *swaps* liefert uns dann die Anzahl der vorhandenen Permutationen innerhalb der Puzzelinstanz.
Die Implementierung ist dabei wie folgt:

```
1 def get_inversion_count(Puzzle: tuple) -> int:
2     working_1d_puzzle = to_1d(Puzzle)
3     count = 0
4     for i in range(len(working_1d_puzzle)):
5         if working_1d_puzzle[i] != i:
```

```
6               count += 1
7               swap(i, find_tile_1d(i, working_1d_puzzle),
                ↪  working_1d_puzzle)
8       return count
```

wobei die Funktion **to_1d** definiert ist durch

```
to_1d = lambda Puzzle: [elem for tupl in Puzzle for elem in tupl]
```

und die Funktion **swap** durch

```
def swap(idxA, idxB, Puzzle_1d):
    Puzzle_1d[idxA], Puzzle_1d[idxB] = Puzzle_1d[idxB],
    ↪  Puzzle_1d[idxA]
```

definiert ist. Die Funktion **find_tile_1d** gibt den Index einer 1-Dimensionalen Liste zurück, an der das entsprechende Element zu finden ist. Diese ist auch im Anhanhg unter A.1 zu finden.
Nun muss als nächstes die Distanz des Blank-Feldes vom Startzustand zum Zielszustand berechnet werden. Hierbei wird die x, y Position des Blank-Feldes im Startzustand gesucht und anschließend ähnlich wie bei der Manhattan Distanz die Absolute Differenz beider Zustände addiert. Da die x, y Position des Blank-Feldes im Endzustand immer $0, 0$ ist, muss diese nicht berechnet werden, wird in der Funktion allerdings auch gesucht, sodass bei einer Erweiterung noch der Endzustand variieren kann. Sei α, β nun die Position des Blank-Feldes der Matrix A und γ, ϵ die Position des Blank-Feldes der Matrix B, so ist der Abstand beider Blank-Felder durch

$$|\alpha - \gamma| + |\beta - \epsilon|$$

definiert. Die dazugehörige Funktion **manhattan** ist auch im Anhang unter A.1 zu finden. Zuallerletzt muss nur noch geschaut werden, ob das Produkt der Distanz und die Anzahl der Permutationen im Startzustand die gleiche Parität besitzen. Damit wissen wir nun, ob diese Instanz lösbar ist.

3.2 Testing

Um nun verschiedene Puzzelinstanzen und deren lösbarkeit anhand des Codes zu testen, werden verschiedene Startzustände und zu Testzwecken verschiedene Endzustände definiert. Anschließend werden alle Startzustände zunächst auf lösbarkeit des „normalen“ Endzustandes getestet, danach auf lösbarkeit von allen anderen definierten Endzuständen. Um eine verbose Ausgabe zu ermöglichen, bei der noch die verschiedenen zuvor berechneten Werte ausgegeben werden können, kann der Funktion, welche die Puzzelinstanzen auf ihre Lösbarkeit prüft noch ein entsprechender verbose Flag mitgegeben werden. Somit kann anhand schon bekannten lösbaren Instanzen überprüft werden, ob die Implementierung vollständig ist.
Der Output dieses Testes befindet sich im dazugehörigen Jupyter-Notebook.
Diese Puzzel werden dann bspw. in das schon aus der Vorlesung bekannte Jupyter-Notebook zum lösen der Instanzen gegeben, sodass geschaut werden kann, ob diese in angemessener Zeit lösbar sind und wieviele Züge für die jeweiliege Lösung gebraucht werden.

4 Zusammenfassung

A Beispiel-Anhang: Testanhang

Python Code zur Validierung der Lösbarkeit von Instanzen des 15-Puzzels

Quelltext A.1: Python Code zur Validierung der Lösbarkeit von Instanzen des 15-Puzzels

```
#!/usr/bin/env python
# coding: utf-8
start1 = {
    'data': ((13, 2, 10, 3),
             (1, 12, 8, 4),
             (5, 0, 9, 6),
             (15, 14, 11, 7)),
    'name': 'start1'}

start2 = {
    'data': ((0,  1,  2,  3),
             (4,  5,  6,  8),
             (14,  7, 11, 10),
             (9, 15, 12, 13)
             ),
    'name': 'start2'
}

start3 = {
    'data': ((6, 13, 7, 10),
             (8, 9, 11, 0),
             (15, 2, 12, 5),
             (14, 3, 1, 4)),
    'name': 'start3'
}


```

```
28 start4 = {
29     'data': ((3, 9, 1, 15),
30              (14, 11, 4, 6),
31              (13, 0, 10, 12),
32              (2, 7, 8, 5)),
33     'name': 'start4'
34 }
35
36 start5 = {
37     'data': ((1,  2,  3, 4),
38              (5,  6,  7, 8),
39              (9, 10, 11, 12),
40              (13, 15, 14, 0)),
41     'name': 'start5'
42 }
43
44 upperLeft = {
45     'data': ((0, 1, 2, 3),
46              (4, 5, 6, 7),
47              (8, 9, 10, 11),
48              (12, 13, 14, 15)),
49     'name': 'solved - Blank upper Left'
50 }
51
52 downRight = {
53     'data': ((1, 2, 3, 4),
54              (5, 6, 7, 8),
55              (9, 10, 11, 12),
56              (13, 14, 15, 0)),
57     'name': 'solved - Blank down Right'
58 }
59 upperRight = {
60     'data': ((1, 2, 3, 0),
61              (4, 5, 6, 7),
62              (8, 9, 10, 11),
```

```
63              (12, 13, 14, 15)),
64      'name': 'solved - Blank upper Right'
65  }
66  downLeft = {
67      'data': ((1, 2, 3, 4),
68               (5, 6, 7, 8),
69               (9, 10, 11, 12),
70               (0, 13, 14, 15)),
71      'name': 'solved - Blank down Left'
72  }
73  spirale = {
74      'data': ((1, 2, 3, 4),
75               (12, 13, 14, 5),
76               (11, 0, 15, 6),
77               (10, 9, 8, 7)),
78      'name': 'spirale Goal'
79  }
80
81  Starts = [start1, start2, start3, start4, start5]
82  Goals = [upperLeft, upperRight, downLeft, downRight, spirale]
83
84
85  def to_1d(Puzzle: tuple) -> list:
86      return [elem for tupl in Puzzle for elem in tupl]
87
88
89  def swap(idxA, idxB, Puzzle_1d):
90      Puzzle_1d[idxA], Puzzle_1d[idxB] = Puzzle_1d[idxB],
        ↪  Puzzle_1d[idxA]
91
92
93  def find_tile_1d(tile, State_1d):
94      n = len(State_1d)
95      for it in range(n):
96          if State_1d[it] == tile:
```

```
97              return it
98
99
100 def get_inversion_count(Puzzle: tuple) -> int:
101     working_1d_puzzle = to_1d(Puzzle)
102     count = 0
103     old_count = -1
104     while old_count != count:
105         old_count = count
106         for i in range(len(working_1d_puzzle)):
107             if working_1d_puzzle[i] != i:
108                 count += 1
109                 swap(i, find_tile_1d(i, working_1d_puzzle),
                    ↪  working_1d_puzzle)
110     return count
111
112
113 def find_tile(tile, State):
114     n = len(State)
115     for row in range(n):
116         for col in range(n):
117             if State[row][col] == tile:
118                 return row, col
119
120
121 def manhattan(stateA, stateB):
122     tile = 0
123     result = 0
124     rowA, colA = find_tile(tile, stateA)
125     rowB, colB = find_tile(tile, stateB)
126     result += abs(rowA - rowB) + abs(colA - colB)
127     return result
128
129
130 def is_solvable(Start: tuple, verbose: bool = False) -> int:
```

```
131     Destination: tuple = upperLeft
132     if verbose:
133         print(f"Name: {Start['name']}")
134         print(f"Start is: {Start['data']}")
135         print(f"Inversion Count:
            ↪ {get_inversion_count(Start['data'])}")
136         print(
137             f"Manhattan Distance: {manhattan(Start['data'],
                ↪ Destination['data'])}")
138     return (get_inversion_count(Start['data']) +
        ↪ manhattan(Start['data'], Destination['data'])) % 2 == 0
139
140
141 for s in Starts:
142     print(f"is solvable: {is_solvable(s, verbose=True)}")
143     print('\n')
144
145 get_inversion_count(start4['data'])
```

B Beispiel-Anhang: Noch ein Testanhang

nochmal: lipsum ...