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.

Kai Fischer, Max Stubenbord

Danksagung

Ein besonderer Dank geht an Prof. Dr. Stroetmann für die Ermöglichung einer alternativen Prüfungsleistung. Wie wir während der Corona-Zeit festgestellt haben, ist ein solches Engagement nicht selbstverständlich.
Abschließend wollen wir uns auch bei dem Kurs TINF18AI1 bedanken für die faire und strukturierte Verteilung der Themen.

Abstract

The 15-puzzle first gained public attention in the 1870s when Sam Loyd promised $\$1000$ in prize money to anyone who could solve his puzzle, which became known as the 14-15 puzzle.
It was not until nearly a decade later that authors of the American Journal of Mathematics published a proof that the puzzle is unsolvable. Today, the 15-puzzle is a classic use case when it comes to modelling algorithms with heuristics, such as *A** or *IDA**.
The main purpose of this work is to outline and implement the algorithm for checking the solvability of an instance of the 15-puzzle. This algorithm is based on Bradlow's approach where a puzzle is considered solvable, if the parity of the transposition count and the manhattan distance of the blank field match.
The In the process written source-code can be found within the appendix. To test this implementation, an evaluation of specific test-cases is conducted.
Altough the algorithm specifies the solvability for a given puzzle instance, it remains uncertain how many steps are required to obtain this solution. Hence the algorithm provides no indicator for computing time.

Inhaltsverzeichnis

Abbildungsverzeichnis **V**

1 Einleitung **1**
1.1 Geschichte 1
1.2 Aufgabenstellung und Abgrenzung 2

2 Grundlagen **4**
2.1 Puzzle zu Listen wandeln 4
2.2 Zustände als Permutationen 5
2.3 Nutzen des Algorithmus an Beispielen 7

3 Implementation **13**
3.1 Umsetzung 13
3.2 Testing 15

4 Zusammenfassung **16**

Literaturverzeichnis **17**

A Anhang **19**

Abbildungsverzeichnis

Abbildung 1.1 Zielzustand des 15 Puzzles 1
Abbildung 1.2 Illustration des 14-15 Puzzles von Sam Loyd 2

Abbildung 2.1 Beispiel Zustand eines 4x4-Puzzle 4
Abbildung 2.2 Häufig verwendeter Zielzustand eines 4x4-Puzzle 5
Abbildung 2.3 Verwendeter Zielzustand eines 4x4-Puzzles aus dem Skript von Herrn Stroetmann [4] 5
Abbildung 2.4 Beispiel1: Startzustand eines 4x4-Puzzles nach figure 2.20 aus [4] . 7
Abbildung 2.5 Beispiel1: Zu erreichender Zielzustand für das Puzzle nach figure 2.20 aus [4] 7
Abbildung 2.6 Beispiel 1: Anwendung des Algorithmus auf ein Vorlesungsbeispiel . 8
Abbildung 2.7 Beispiel2: Startzustand für das Loyd Puzzle 9
Abbildung 2.8 Beispiel2: Zu erreichender Zielzustand für das Loyd Puzzle 9
Abbildung 2.9 Beispiel2: Anwendung des Algorithmus auf das Loyd Puzzle 10
Abbildung 2.10 Beispiel 2: Anwendung des Algorithmus auf die Überführbarkeit der Endzustände von Loyd und Stroetmann. 11

Abbildung A.1 Testing Output der verschiedenen Startzustände 24
Abbildung A.2 Lösbarkeitsberechnungsdauer 25
Abbildung A.3 Exemplarisches Lösen eines 15-Puzzles mit hohem Zeitaufwand (1h) 25

1 Einleitung

Diese Arbeit beschäftigt sich mit dem Überprüfen der Lösbarkeit einer Instanz des 15-Puzzles. Zunächst wird der historische Hintergrund beleuchtet, anschließend wird der Algorithmus von Herrn Bradlow vorgestellt und das notwendige Vorgehen anhand von Beispielen dargestellt.
Abschließend wird die Implementierung und das Testen der Umsetzung des Algorithmus' in der Programmiersprache Python beschrieben.

1.1 Geschichte

Die erste bekannte Erscheinung des heute so bekannten Puzzles ist in den 1870'er Jahren in Amerika aufgetreten. Bisher ist man davon ausgegangen, dass der Erfinder des Puzzles der Amerikaner Sam Loyd ist, jedoch ist nach einer Untersuchung von Jerry Slocum und Dieter Gebhardt Sam Loyd gar nicht der echte Erfinder des 15-Puzzles [6, 13]. Demnach hat Sam Loyd die Idee nur gut vermarktet und sich selbst dadurch in die Öffentlichkeit gestellt [2].
Nichtsdestotrotz hat Sam Loyd es geschafft, die Aufmerksamkeit der Öffentlichkeit auf das Puzzle zu lenken und somit das Interesse vieler zu wecken.
Das Ziel des Puzzles ist es 15 Puzzlesteine numeriert von 1-15 auf einer 4 x 4 Ebene durch Verschiebung in seine Ursprüngliche sortierte Form zurückzubringen (vgl. Abb.1.1).

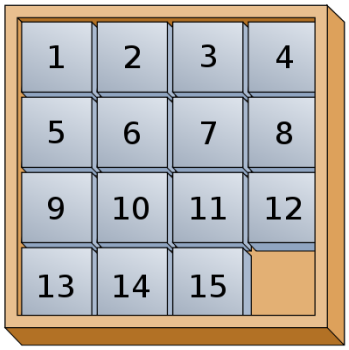

Abbildung 1.1: Zielzustand des 15 Puzzles

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

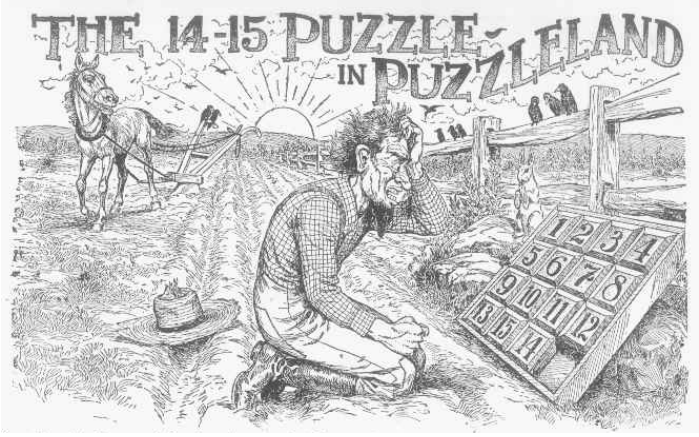

Abbildung 1.2: Illustration des 14-15 Puzzles von Sam Loyd (1914)
Quelle: [11, pp. 234-235]

Allerdings zeigte sich im Nachhinein, dass das Puzzle unlösbar ist, da es eine Transformation von einer geraden zu einer ungeraden Permutation erfordert [2]. Die Unlösbarkeit des Puzzles wurde erstmals von Wm. Woolsey Johnson und William E. Story im Jahre 1879 bei einer Veröffentlichung des American Journal of Mathematics bewiesen. [9] Somit gewann keiner das Preisgeld von $\$1000$. Das 15-Puzzle 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 [1].

1.2 Aufgabenstellung und Abgrenzung

Hätte man damals bewiesen, dass das Puzzle aus der Einleitung von Sam Loyd nicht lösbar ist, wäre wohl vielen Menschen nächtelange Verzweiflung erspart geblieben.
Nun stellt sich die Frage, wann eine Instanz des Puzzles lösbar ist. Allgemein gesprochen ist eine Instanz eines 15 Puzzles 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 $16! \approx 2 \cdot 10^{13}$ Puzzle

Kombinationen [7, 5]. Das Puzzle Problem ist ein klassischer Anwendungsfall, wenn es um Modellierung von Algorithmen mit Heuristiken geht. Üblicherweise werden Algorithmen wie *A** oder *IDA** zur Lösung solcher Heuristiken genutzt [2, 5, 10]. Im Rahmen dieser Arbeit geht es nun darum, bei gegebener Puzzleinstanz 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 ausführlich im zugehörigen Vorlesungsskript [4] in den Sektionen ***2.7*** *A* Search* und ***2.10*** *A*-IDA* Search* erläutert. Lediglich werden diese genutzt, um die als lösbar identifizierten Puzzles anschließend auch zu lösen.

2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen und Gedanken für die Umsetzung in 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“ auf dem Youtube Kanal „Numberphile“ [12].

2.1 Puzzle zu Listen wandeln

Der Lösungsansatz aus [12] 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 Puzzle zeilenweise hintereinander in eine Liste geschrieben. Die Leerstelle, auch als „blank“ beschrieben, wird dabei als Zahl „0“ interpretiert. Der Zustand des Puzzle 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-Puzzle

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 Vorgehens der Lösung aus [12]

aber auch anderer verfügbarer Quellen [5, 8, 3], dass die Bezeichnung der Leerstelle oder die Art der Konvertierung eines Puzzle 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 [4] übereinzustimmen, wird der Zielzustand aus Abb.2.3 angestrebt, bei dem die Leerstelle die Nummer „0“ trägt.

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

Abbildung 2.2: Häufig verwendeter Zielzustand eines 4x4-Puzzle

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

Abbildung 2.3: Verwendeter Zielzustand eines 4x4-Puzzles aus dem Skript von Herrn Stroetmann [4]

2.2 Zustände als Permutationen

Die Grundidee 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 [12] zusammengefasst vorgestellt.

Die Vorstellung des Verfahrens beginnt Bradlow damit, die Puzzle der vorherigen Sektion in Zahlenreihen zu wandeln. Durch das Vertauschen von 2 beliebigen Zahlen aus dieser Zahlenreihe entsteht eine Permutation der vorherigen Reihe. Auf diese Weise sei jeder mögliche Zustand, in dem sich das Puzzle befinden kann, als eine Permutation einer zugrundeliegenden aufsteigend sortierten Zahlenreihe zu sehen. Für diese Permutationen beschreibt 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 [Vgl. 12, 7min, 07sec].

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 [Vgl. 12, 10min, 13sec].

Basierend auf diesen Eigenschaften fährt er fort, dass die Parität der Anzahl notwendiger Transpositionen durch das zielgerichtete Tauschen der Zahlen, mit dem Ziel einer aufsteigend sortierten 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 sortierte 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 des Puzzles nicht betrachtet und somit möglicherweise nicht eingehalten werden können.

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 Leerstellenbewegung ist legal, während die Parität der zur Sortierung notwendigen Züge nur hypothetisch ist. Stimmen die Paritäten nicht überein, existiert nach E2 keine Möglichkeit beide Ziele, also die Sortierung ebenso wie das Einhalten von Zugregeln zu berücksichten.

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 Nutzen des Algorithmus an Beispielen

Aus der Vorlesung von Herrn Stroetmann und dem begleitenden Skript [4] ist bekannt, dass sich das Puzzle aus Abb. 2.4 lösen, also in den Zielzustand aus Abb. 2.5 überführen lässt.

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

Abbildung 2.4: Beispiel1: Startzustand eines 4x4-Puzzles nach figure 2.20 aus [4]

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

Abbildung 2.5: Beispiel1: Zu erreichender Zielzustand für das Puzzle nach figure 2.20 aus [4]

Der Algorithmus aus dem vorherigen Abschnitt schätzt diese Puzzles ebenfalls als lösbar ein.

S1.1 Konvertierung der Puzzle-Zustände in Zahlenreihen

$$start_{ex1} = \{0, 1, 2, 3, 4, 5, 6, 8, 14, 7, 11, 10, 9, 15, 12, 13\}$$
$$end_{ex1} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$$

S1.2 Notwendige Transpositionen T_{ex1} um $start_{ex1}$ zu sortieren:

$0, 1, 2, 3, 4, 5, 6, 8, 14, 7, 11, 10, 9, 15, 12, 13$	$(8, 7)$
$0, 1, 2, 3, 4, 5, 6, 7, 14, 8, 11, 10, 9, 15, 12, 13$	$(14, 8)$
$0, 1, 2, 3, 4, 5, 6, 7, 8, 14, 11, 10, 9, 15, 12, 13$	$(14, 9)$
$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10, 14, 15, 12, 13$	$(11, 10)$
$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 14, 15, 12, 13$	$(14, 12)$
$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 14, 13$	$(15, 13)$
$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$	

$$T_{ex1} = \{(8, 7), (14, 8), (14, 9), (11, 10), (14, 12), (15, 13)\}$$
$$|T_{ex1}| = 6 \rightarrow \texttt{Parität} = 0$$

S1.3 Anzahl der notwendigen Züge z_{ex1} um das Leerfeld von der Postion aus dem Startzustand (Koordinaten $(0, 0)$) an die Postion des Zielzustandes (Koordinaten $(0, 0)$) zu verschieben.

$$z_{ex1} = |0 - 0| + |0 - 0|$$
$$z_{ex1} = 0 \rightarrow \texttt{Parität} = 0$$

S1.4 Die Paritäten stimmen überein, das Puzzle ist lösbar.

Abbildung 2.6: Beispiel 1: Anwendung des Algorithmus auf ein Vorlesungsbeispiel

Dazu wird wie in Abb. 2.6 schrittweise vorgegangen. Zunächst werden, wie im Abschnitt 2.1 und in Schritt 1 (S1.1) dargestellt, die Zustände in Zahlenreihen konvertiert. Anschließend werden die Paritäten ermittelt (S1.2 und S1.3). Zunächst die Parität der Anzahl notwendiger Transpositionen und anschließend die Parität benötigter Züge um die Leerstelle korrekt zu platzieren. Im abschließenden Schritt (S1.4) werden die ermittelten Paritäten verglichen. In diesem Beispiel stimmen diese Paritäten überein. Das Beispiel ist also lösbar.

Als nächstes wird das Puzzle von Loyd aus der Einleitung betrachtet. Der bereits bekannte Startzustand ist in Abb. 2.7 zur Vereinfachung erneut abgebildet. Der Zielzustand ist

daneben in Abb. 2.8 dargestellt.

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

Abbildung 2.7: Beispiel2: Startzustand für das Loyd Puzzle

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

Abbildung 2.8: Beispiel2: Zu erreichender Zielzustand für das Loyd Puzzle

Auch hier wird der beschriebene Algorithmus angewandt, das Vorgehen ist in Abb. 2.9 skizziert.

S2.1 Konvertierung der Puzzle-Zustände in Zahlenreihen

$$start_{ex2} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 14, 0\}$$
$$end_{ex2} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0\}$$

S2.2 Notwendige Transpositionen T_{ex2} um $start_{ex2}$ zu end_{ex2} zu überführen:

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 14, 0 \qquad (15, 14)$$
$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0$$

$$T_{ex2} = \{(14, 15)\}$$
$$|T_{ex2}| = 1 \rightarrow \texttt{Parität} = 1$$

S2.3 Anzahl der notwendigen Züge z_{ex2}, um das Leerfeld von der Postion aus dem Startzustand (Koordinaten $(3, 3)$) an die Postion des Zielzustandes (Koordinaten $(3, 3)$) zu verschieben.

$$z_{ex1} = |3 - 3| + |3 - 3|$$
$$z_{ex1} = 0 \rightarrow \texttt{Parität} = 0$$

S2.4 Die Paritäten stimmen nicht überein, das Puzzle ist somit nicht lösbar.

Abbildung 2.9: Beispiel2: Anwendung des Algorithmus auf das Loyd Puzzle

Da die Zielzustände der beiden Puzzle verschieden sind (Abb. 2.5 und Abb. 2.8), unterscheiden sich auch die Zahlenreihen als welche diese dargestellt werden. Doch zur Übersichtlichkeit wurde sowohl in S1.1 als auch in S2.1 die Leerstelle mit der Zahl „0“ kodiert. Es ist leicht zu erkennen, dass nur eine Transposition notwendig ist, um die Zahlenreihe $start_{ex2}$ in die Reihe end_{ex2} zu überführen, die Parität des Vorgangs ist also 1. Da sich die Leerstelle bereits an der korrekten Stelle befindet, sind keine Züge notwendig, die Parität ist also 0. Da die Paritäten unterschiedlich sind, kategorisiert der Algorithmus dieses Puzzle korrekterweise als nicht lösbar ein.

Die Wahl des Endzustandes ist relevant bei der Lösbarkeitsüberprüfung eines Puzzles. In der Sektion 2.2 wurde explizit hervorgehoben, dass sich der in der Literatur übliche Endzustand (Abb. 2.8) von dem in Herrn Stroetmanns Skript [4] (Abb. 2.5) unterscheidet. In Abb. 2.10 zeigt der Algorithmus, dass diese Endzustände nicht ineinander Überführbar sind, In Schritt S3.1 werden Leerstellen erneut mit der Zahl „0“ kodiert.

S3.1 Konvertierung der Puzzle-Zustände in Zahlenreihen

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

S3.2 Notwendige Transpositionen T_{ex3} um end_{Loyd} zu $end_{Stroetmann}$ überführen:

$$\begin{aligned}
&1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0 && (1, 0)\\
&0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 1 && (2, 1)\\
&0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 2 && (3, 2)\\
&0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 3 && (4, 3)\\
&0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 4 && (5, 4)\\
&0, 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 5 && (6, 5)\\
&0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 6 && (7, 6)\\
&0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 7 && (8, 7)\\
&0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 8 && (9, 8)\\
&0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 9 && (10, 9)\\
&0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 10 && (11, 10)\\
&0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 11 && (12, 11)\\
&0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 12 && (13, 12)\\
&0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 13 && (14, 13)\\
&0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 14 && (15, 14)\\
&0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
\end{aligned}$$

$$T_{ex3} = \{(1, 0), (2, 1), (3, 2), (4, 3), (5, 4), (6, 5), (7, 6), (8, 7), (9, 8), (10, 9), (11, 10), (12, 11), (13, 12), (14, 13), (15, 14)\}$$

$$|T_{ex3}| = 15 \rightarrow \texttt{Parität} = 1$$

S3.3 Anzahl der notwendigen Züge z_{ex2} um das Leerfeld von der Postion aus Loyds Endzustand (Koordinaten $(3, 3)$) an die Postion des Zielzustandes von Stroetmann (Koordinaten $(0, 0)$) zu verschieben.

$$z_{ex3} = |3 - 0| + |3 - 0|$$
$$z_{ex3} = 6 \rightarrow \texttt{Parität} = 0$$

S3.4 Die Paritäten stimmen nicht überein, das Puzzle ist somit nicht lösbar.

Abbildung 2.10: Beispiel 2: Anwendung des Algorithmus auf die Überführbarkeit der Endzustände von Loyd und Stroetmann.

Ist ein Puzzle also für den einen Endzustand lösbar, ist es für den anderen Endzustand automatisch nicht lösbar, da es keine Möglichkeit gibt die Endzustände ineinander zu überführen.

3 Implementation

3.1 Umsetzung

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

Zur Umsetzung wird zuerst eine Datenstruktur definiert, in welcher die Instanzen des 15-Puzzles ausgewertet werden. Hierfür werden Tupel von Tupeln verwendet. Anschließend ist die generelle Idee, wie auch schon in Abschnitt 2.3 und 2.2 vorgestellt, zu schauen, ob die Anzahl der Permutationen bei der Datenstruktur als eindimensionale Liste und der Abstand des Blank-Feldes vom Start- zum Zielzustand die gleiche Parität besitzen. Ist diese gleich, so ist das Puzzle lösbar, vice versa unlösbar.
Spannend ist hier allerdings, dass dies kein Indikator für die Komplexität der Lösbarkeit ist. So kann ein lösbares Puzzle mit den zur Verfügung stehenden Algorithmen wie *A** oder *IDA** nicht in angemessener Zeit gelöst werden.
Das weitere Vorgehen der Implementierung ist weitestgehend 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 Puzzleinstanz 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. Vereinfacht wird dieser Schritt dadurch, dass der Lösungszustand des Puzzles so definiert ist, 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 mindestens durchzuführenden Permutationen innerhalb der Puzzleinstanz.
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 eindimensionalen Liste zurück, an dem das entsprechende Element zu finden ist. Diese ist auch im Anhang unter A.1 zu finden.
Nun muss als Nächstes die Distanz des Blank-Feldes vom Startzustand zum Zielszustand berechnet werden. Wie schon in Abschnitt 2.2 angesprochen, kann die Anzahl der notwendigen Züge, um das Blank-Feld von der Position im Startzustand zu der Position im Zielzustand zu bewegen, durch die Manhattan-Distanz ermittelt werden.
Da die x, y Position des Blank-Feldes im Endzustand immer $0, 0$ ist, muss diese nicht berechnet werden, wird in der Funktion trotzdem gesucht, sodass bei einer Erweiterung noch der Endzustand variieren kann. Sei x_1, y_2 nun die Position des Blank-Feldes im Startzustand und x_2, y_2 die Position des Blank-Feldes im Endzustand, so ist der Abstand beider Blank-Felder durch

$$|x_1 - x_2| + |y_1 - y_2|$$

definiert.
Die dazugehörige Funktion **manhattan** ist auch im Anhang unter A.1 zu finden.
Abschließend wird verglichen, ob die Distanz und die Anzahl der Permutationen die gleiche Parität besitzen. Stimmen die Paritäten überein, so ist die Instanz lösbar, sind diese unterschiedlich, so ist diese unlösbar.

3.2 Testing

Um nun verschiedene Puzzleinstanzen und deren Lösbarkeit anhand des Codes zu testen, werden verschiedene Startzustände und zu Testzwecken verschiedene Endzustände definiert. Anschließend werden alle Startzustände auf Lösbarkeit des „normalen“ Endzustandes getestet.
Um eine verbose Ausgabe zu ermöglichen, bei der noch die verschiedenen zuvor berechneten Werte ausgegeben werden können, kann der Funktion, welche die Puzzleinstanzen auf ihre Lösbarkeit prüft, noch eine entsprechende verbose Flag mitgegeben werden. Somit kann anhand von bekannten lösbaren Instanzen überprüft werden, ob die Implementierung vollständig ist.
Der Output dieses Testes befindet sich im dazugehörigen Jupyter-Notebook.
Diese Puzzle werden dann bspw. in das schon aus der Vorlesung bekannte Jupyter-Notebook zum Lösen der Instanzen gegeben, so dass geschaut werden kann, ob diese in angemessener Zeit lösbar sind, und wieviele Züge für die jeweilige Lösung gebraucht werden.
Ein Screenshot des Outputs findet sich im Anhang unter A.1. Das Problem hierbei ist, dass nicht jedes als lösbar einkategorisierte Puzzle auch in angemessener Zeit mit Hilfe des *IDA** oder *A** Algorithmus gelöst werden kann. So ist es durchaus vorgekommen, dass die Schritte zur Lösbarkeit eines Puzzles mehrere Stunden dauern (vgl. A.3), da vorher nicht bekannt ist, wieviele Schritte zur Lösung führen.
Somit weiß man initial nicht, ob das Puzzle richtig einklassifiziert wurde.

4 Zusammenfassung

Diese Arbeit zeigt, dass Instanzen des 15-Puzzle programmatisch auf ihre Lösbarkeit überprüft werden können, indem sie als Permutationen in einer sortierten Liste betrachtet werden. Der daraus resultierende von Bradlow vorgestellte Algorithmus ist leicht zu verstehen und lässt sich ohne Umwege in einem Python-Modul abbilden. Dieses kann als Filter genutzt werden, um sicherzustellen, dass es eine Lösung gibt, die die in der Vorlesung behandeltelten Suchalgorithmen [vgl. 4, Kap. 2] finden können.
Die Überprüfung durch einen Suchalgorithmus, der alle möglichen Zustände durcharbeitet, ist mit einem höherem Zeitaufwand verbunden, als es der Filter benötigt. So kann frühzeitig erkannt werden, dass ein Puzzle nicht lösbar ist und weder Suchalgorithmen noch Menschen, wie in der Einleitung 1.1 beschrieben, müssen nächtelang versuchen, eine unlösbare Aufgabe zu lösen.

Zukünftig könnte das Modul so erweitert werden, dass beliebige Endzustände möglich sind, wie bspw. in A.1 (Z. 52 - Z. 79) definiert sind. Aktuell wird nur auf die Lösbarkeit mit dem von Herrn Stroetmann definierten Endzustand geprüft.

Literaturverzeichnis

[1] *15–14–13. The great presidential puzzle*. http://loc.gov/pictures/resource/ppmsca.15782/. (Accessed on 05/30/2021).

[2] *15 puzzle - Wikipedia*. https://en.wikipedia.org/wiki15_puzzle. (Accessed on 05/30/2021).

[3] Aaron F. Archer. „A Modern Treatment of the 15 Puzzle“. In: *The American Mathematical Monthly* 106.9 (1999), S. 793–799. DOI: 10.1080/00029890.1999.12005124. eprint: https://doi.org/10.1080/00029890.1999.12005124. URL: https://doi.org/10.1080/00029890.1999.12005124.

[4] *Artificial-Intelligence/artificial-intelligence.pdf at master · karlstroetmann/Artificial-Intelligence*. https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Lecture-Notes/artificial-intelligence.pdf. (Accessed on 06/06/2021).

[5] Bastian Bischoff et al. „Solving the 15-Puzzle Game Using Local Value-Iteration“. In: Bd. 257. Jan. 2013, S. 45–54. DOI: 10.3233/978-1-61499-330-8-45.

[6] Jerry Slocum & Dieter Gebhardt. *THE ANCHOR PUZZLE BOOK*. Art of Play, 2012.

[7] Edward Hordern. *Sliding Piece Puzzles*. New York: Oxford University Press, 1986. ISBN: 978-0-198-53204-0.

[8] *How to check if an instance of 15 puzzle is solvable? - GeeksforGeeks*. https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/. (Accessed on 06/08/2021).

[9] Wm. Woolsey Johnson und William E. Story. „Notes on the "15"Puzzle“. In: *American Journal of Mathematics* 2.4 (1879), S. 397–404. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2369492.

[10] Richard E. Korf. „Depth-first iterative-deepening: An optimal admissible tree search“. In: *Artificial Intelligence* 27.1 (1985), S. 97–109. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(85)90084-0. URL: https://www.sciencedirect.com/science/article/pii/0004370285900840.

[11] S. Loyd und S. Sloan. *Sam Loyd's Cyclopedia of 5000 Puzzles Tricks and Conundrums with Answers*. Ishi Press International, 2007. ISBN: 9780923891787. URL: https://books.google.de/books?id=noWnPQAACAAJ.

[12] Numberphile. *Why is this Puzzle Impossible? - Numberphile*. https://www.youtube.com/watch?v=YI1WqYKHi78. (Accessed on 06/06/2021). 2020.

[13] *Review of The 15 Puzzle*. https://www.cut-the-knot.org/books/Reviews/The15Puzzle.shtml. (Accessed on 05/30/2021).

A Anhang

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

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

```
#!/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 to_1d_lambda(Puzzle): return [elem for tupl in Puzzle for elem in
    ↪  tupl]
90
91
92  def swap(idxA, idxB, Puzzle_1d):
93      Puzzle_1d[idxA], Puzzle_1d[idxB] = Puzzle_1d[idxB],
        ↪  Puzzle_1d[idxA]
94
95
```

```
96  def find_tile_1d(tile, State_1d):
97      n = len(State_1d)
98      for it in range(n):
99          if State_1d[it] == tile:
100             return it
101
102
103 def get_permutation_count(Puzzle: tuple) -> int:
104     working_1d_puzzle = to_1d(Puzzle)
105     count = 0
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 get_permutation_count_recursive(Puzzle: tuple) -> int:
114     Puzzle = to_1d(Puzzle)
115
116     def count(Puzzle, size):
117         if len(Puzzle) == 1:
118             return 0
119         curr_index = size - (len(Puzzle) - 1)
120         if Puzzle[0] != curr_index:
121             swap(0, find_tile_1d(curr_index, Puzzle), Puzzle)
122             return 1 + count(Puzzle[1:], size)
123         return count(Puzzle[1:], size)
124     return count(Puzzle, len(Puzzle) - 1)
125
126
127 assert get_permutation_count_recursive(
128     start1['data']) == get_permutation_count(start1['data'])
129
```

```
130
131 def find_tile(tile, State):
132     n = len(State)
133     for row in range(n):
134         for col in range(n):
135             if State[row][col] == tile:
136                 return row, col
137
138
139 def manhattan(stateA, stateB):
140     tile = 0
141     rowA, colA = find_tile(tile, stateA)
142     rowB, colB = find_tile(tile, stateB)
143     return abs(rowA - rowB) + abs(colA - colB)
144
145
146 def is_solvable(Start: tuple, verbose: bool = False) -> int:
147     Destination: tuple = upperLeft
148     if verbose:
149         print(f"Name: {Start['name']}")
150         print(f"Start is: {Start['data']}")
151         print(f"Inversion Count:
          ↪ {get_permutation_count(Start['data'])}")
152         print(
153             f"Manhattan Distance: {manhattan(Start['data'],
              ↪ Destination['data'])}")
154     return (get_permutation_count(Start['data']) +
      ↪ manhattan(Start['data'], Destination['data'])) % 2 == 0
155
156
157 for s in Starts:
158     print(f"is solvable: {is_solvable(s, verbose=True)}")
159     print('\n')
```

Testing Output der verschiedenen Startzustände

```
Name: start1
Start is: ((13, 2, 10, 3), (1, 12, 8, 4), (5, 0, 9, 6), (15, 14, 11, 7))
Inversion Count: 14
Manhattan Distance: 3
is solvable: False


Name: start2
Start is: ((0, 1, 2, 3), (4, 5, 6, 8), (14, 7, 11, 10), (9, 15, 12, 13))
Inversion Count: 6
Manhattan Distance: 0
is solvable: True


Name: start3
Start is: ((6, 13, 7, 10), (8, 9, 11, 0), (15, 2, 12, 5), (14, 3, 1, 4))
Inversion Count: 13
Manhattan Distance: 4
is solvable: False


Name: start4
Start is: ((3, 9, 1, 15), (14, 11, 4, 6), (13, 0, 10, 12), (2, 7, 8, 5))
Inversion Count: 13
Manhattan Distance: 3
is solvable: True


Name: start5
Start is: ((1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12), (13, 15, 14, 0))
Inversion Count: 14
Manhattan Distance: 6
is solvable: True
```

Abbildung A.1: Testing Output der verschiedenen Startzustände

Puzzleinstanz mit hoher Lösbarkeitsberechnungsdauer

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

Abbildung A.2: Lösbarkeitsberechnungsdauer

Exemplarisches Lösen der in A.2 angegebenen Puzzleinstanz (1h)

```
%%time
memory_before = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
Path          = search(start2, goal2, next_states, manhattan, 10000)
memory_after  = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
print(f'Total memory used: {round((memory_after - memory_before) / 2**10)} kiloytes.')
print(len(Path)-1)
```

```
limit = 41
limit = 43
limit = 45
limit = 47
limit = 49
limit = 51
limit = 53
limit = 55
limit = 57
limit = 59
Total memory used: 1 kiloytes.
61
CPU times: user 1h 9min 55s, sys: 27.8 ms, total: 1h 9min 55s
Wall time: 1h 9min 57s
```

```
animation(Path)
```

Abbildung A.3: Exemplarisches Lösen eines 15-Puzzles mit hohem Zeitaufwand (1h)