

ABSTRACT

Schiebepuzzles - besonders das bekannte 15-Puzzle - stellen bereits seit Jahrzehnten eine Faszination für Puzzleliebhaber dar. Über die Jahre hat sich der Trend so entwickelt, dass die meisten Schiebepuzzles nicht mehr simple Zahlen oder Buchstaben abbilden, sondern ganze Bilder zerlegen. In dieser Arbeit werden zwei bekannte Probleme der Informatik untersucht und so kombiniert, dass diese einen möglichen Lösungsansatz zu dem bildbasierten Schiebepuzzle-Problem geben.

INHALTSVERZEICHNIS

1	EINLEITUNG	1
1.1	Problemstellung und Zielsetzung	2
1.2	Aufbau der Arbeit	2
2	OPENCV GRUNDLAGEN	4
2.1	Mat - Der OpenCV Bild-Container	5
2.2	Saturate-Casting	6
2.3	Datentypen für Pixel	7
2.4	Beispiel: Verwendung von Matrizen und Region of Interests	9
2.5	Transformation von Bildern	10
3	METRIKEN ZUR BILDREKONSTRUKTION	14
3.1	$(L_p)^q$ Norm	17
3.2	Dissimilarity-Based Compatibility (DBC)	18
3.3	Prediction-Based Compatibility (PBC)	19
3.4	Dynamic Time Warping (DTW)	23
3.5	Mahalanobis Gradient Compatibility (MGC)	30
4	ALGORITHMUS ZUR PUZZLE-ASSEMBLIERUNG	33
4.1	Baum-basierte Assemblierung nach Gallagher	34
4.2	Trimming	43
4.3	Filling	44
4.4	Kanten-Vorverarbeitung	45
4.5	Ergebnis	46
5	LÖSBARKEIT EINES SCHIEBEPUZZLES	47
6	LÖSEN DES SCHIEBEPUZZLES	54
6.1	Breadth-First Search (BFS)	55
6.2	Iterative Deepening Depth-First Search (IDDFS)	56
6.3	Heuristiken	57
6.3.1	Hamming-Distanz	59
6.4	Summe der Manhatten-Distanzen	59
6.5	A*	60
6.6	Iterative Deepening A* (IDA*)	61
6.7	Ergebnis	62
7	FAZIT & AUSBLICK	64

ABBILDUNGSVERZEICHNIS

Abbildung 1.1	Zwei Arten des 15-Puzzle	1
Abbildung 2.1	Ausgabe vom Mat und ROI Beispiel	10
Abbildung 2.2	Rotation als affine Transformation	12
Abbildung 2.3	Abbildung zweier Dreiecke als affine Transformation	13
Abbildung 3.1	Beispiel eines Jigsaw-Puzzles	14
Abbildung 3.2	Anzahl der Kanten bei einem Type 3 Puzzle	16
Abbildung 3.3	Beispiel eines Type 1 Puzzles	17
Abbildung 3.4	Lineare Approximierung der Pixel am rechten und am oberen Bildrand	22
Abbildung 3.5	Direkte Zuordnung der Pixel	24
Abbildung 3.6	Euclidean Matching im Vergleich mit DTW	25
Abbildung 3.7	Zuordnung der Pixel beim DTW-Algorithmus	30
Abbildung 4.1	Koordinatentransformation im DSF bei Pfadkomprimierung	40
Abbildung 4.2	Die 4 Stages eines Puzzles im Gesamtalgorithmus	43
Abbildung 4.3	Fehler beim Filling Algorithmus	45

TABELLENVERZEICHNIS

Tabelle 3.1	Räumlicher Zusammenhang zweier Puzzlestücke	18
Tabelle 3.2	Backtracking im DTW-Algorithmus	29

LISTINGS

2.1	Verwendung von Mat und ROI	9
3.1	Lineare Approximierung (nach Taylor) von Pixel am rechten Bildrand	21

LIST OF ALGORITHMS

1	DTW-Algorithmus	28
2	Initialisierung eines Disjoint-Set-Forest	36
3	FindRoot Operation mit Pfadkomprimierung	36
4	Union Operation als Union by Size	37
5	Erweiterte InitForest Operation	38
6	Erweiterte FindRoot Operation	39
7	Erweiterte Union Operation	41
8	Breadth-First Search (BFS)	55
9	Iterative Deepening Depth-First Search (IDDFS)	57
10	A*	61
11	Iterative Deepening A* (IDA*)	62

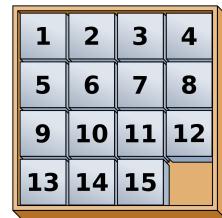
1

EINLEITUNG

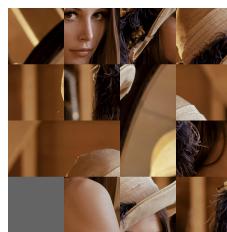
Schiebepuzzles wie das 15-Puzzle haben zum Ende des 19. Jahrhunderts das Interesse vieler amerikanischer Puzzle-Enthusiasten geweckt[25, 19] Heutzutage gibt es solche Schiebepuzzles in vielen Varianten. Dazu gehört die weitverbreitete Variante, bei der es nicht das Ziel ist, Zahlen aufsteigend zu sortieren, sondern bei dem das Puzzle aus einem Bild besteht, welches in quadratische Stücke aufgeteilt und durcheinander gemischt wurde. Das Originalbild lässt sich erst dann komplett erkennen, nachdem die Puzzlestücke in die richtige Reihenfolge gebracht wurden.



(a) 15-Puzzle mit Zahlen
(gemischt)



(b) 15-Puzzle mit Zahlen
(gelöst)



(c) 15-Puzzle mit Bild
(gemischt)



(d) 15-Puzzle mit Bild
(gelöst)

Abbildung 1.1: Zwei Arten des 15-Puzzle

In der Informatik ist das Lösen von Schiebepuzzles ein klassisches Problem der künstlichen Intelligenz und ein übliches Beispielproblem für die Modellierung und Illustration von Suchalgorithmen. Dabei beschränkt sich die meiste Literatur auf das Lösen des klassischen Schiebepuzzles mit Zahlen.

Um diese Lösungsverfahren auf ein Schiebepuzzle mit Bild zu übertragen, muss also zuvor separat das Originalbild rekonstruiert werden.

In dieser Arbeit wird genau solch ein zweischrittiger Ansatz erläutert und analysiert.

1.1 PROBLEMSTELLUNG UND ZIELSETZUNG

In dieser Arbeit wird das Problem anhand des allgemeinen N -Puzzles betrachtet mit $N = m \times n - 1$ (für das 15-Puzzle gilt somit $m = n = 4$). Die N Puzzlestücke sind dabei rechteckige Teile eines Bildes. Diese müssen nicht zwingend quadratisch sein, sollten aber alle das gleiche Seitenverhältnis aufweisen, damit diese als Teil eines Schiebepuzzles auch wirklich alle verschiebbar sind und sich somit in den Zustand eines Schiebepuzzles zusammensetzen lassen.

Die Position des leeren Feldes ist dabei nicht zwingend fest (so wie in 1.1 z. B. immer unten-rechts) und kann variieren. Diese Position ist außerdem unbekannt und muss damit anhand der Zusammensetzung der restlichen N Puzzleteilen und den bekannten Dimensionen des Puzzles ausgemacht werden.

Außerdem wird noch eine $m \times n$ Anordnung der N Puzzlestücke (und dem leeren Feld) als Anfangszustand vorgegeben.

Folgende Fragen sollen dann (in dieser Reihenfolge) beantwortet werden:

1. Wie sah das originale Bild aus und wo befindet sich das leere Feld im Ausgangsbild (wo fehlt also ein Stück des Bildes)?
2. Ist das Puzzle lösbar, lässt sich also der gegebene Anfangszustand unter einer endlichen Sequenz von legalen Zügen (also ausschließlich durch das hin- und herschieben von Puzzlestücken) in den zuvor ermittelten Endzustand transformieren?
3. Wie viele Schritte (Verschiebungen) sind mindestens nötig, um das Puzzle zu lösen und wie sieht ein optimaler Lösungsweg (ein Lösungsweg mit der kleinstmöglichen Anzahl an Verschiebungen) aus?

Dazu wird ein vollautomatischer Schiebepuzzleslöser programmiert, der genau diese Schritte abarbeitet und am Ende bei einer gefundenen (optimalen) Lösung das Ergebnis dem Benutzer präsentiert.

Programmiert wird der Puzzleslöser in der Programmiersprache C++. Zur Analyse und Verarbeitung des Bildes wird die Open Source Computer Vision Bibliothek OpenCV verwendet.

1.2 AUFBAU DER ARBEIT

Zunächst werden einige Grundlagen angesprochen, die den Umgang mit OpenCV erläutern und unter anderem aufklären, wie Bilder in OpenCV dargestellt und bearbeitet werden können.

Da zur Rekonstruktion Informationen wie die Kompatibilität (oder auch Ähnlichkeit) zweier Puzzlestücke notwendig ist, werden daraufhin einige Metriken vorgestellt, die diese Information repräsentieren.

*m: Anzahl der Zeilen
n: Anzahl der Spalten*

Daraufhin wird die Rekonstruktion des Bildes weiter erläutert und ein Greedy Verfahren vorgestellt, welches dieses Problem mithilfe der zuvor berechneten Metriken löst.

Im nächsten Schritt wird das Puzzle mit seinem Anfangs- und Endzustand in ein äquivalentes Schiebepuzzle mit Zahlen übersetzt. Damit werden die nächsten Schritte anhand dem klassischen Zahlenpuzzle (jedoch weiterhin mit beliebigen Dimensionen und beliebigem Anfangs- und Endzustand) gezeigt.

Es wird dann erläutert, wie sich die Lösbarkeit eines Schiebepuzzles bestimmen lässt. Zusätzlich wird geklärt, wie sich zufällige Schiebepuzzle so generieren lassen, dass diese wahlweise immer lösbar oder immer unlösbar sind.

Zum Schluss werden Suchalgorithmen aus der künstlichen Intelligenz vorgestellt, die das eigentliche Lösen des Puzzles übernehmen. Dazu werden sowohl uninformierte Suchalgorithmen, als auch informierte Suchalgorithmen (welche Heuristiken in ihre Suche mit einbeziehen) untersucht und verglichen. An dieser Stelle werden somit auch verschiedene Heuristiken für Schiebepuzzle vorgestellt.

2

OPENCV GRUNDLAGEN

OpenCV (Open Source Computer Vision) ist eine Open Source Programm-bibliothek, die mehr als 2500 optimierte Algorithmen aus den Bereichen Bildverarbeitung, maschinelles Sehen und maschinelles Lernen beinhaltet.[1] OpenCV steht als freie Software (unter den Bedingungen der BSD-Lizenz) für die Programmiersprachen C++, Java, Python und MATLAB und den Betriebssystemen Windows, Linux, Android und Mac OS zur Verfügung. OpenCV wurde nativ in der Programmiersprache C++ geschrieben und bietet somit Schnittstellen an, die reibungslos mit der C++ Standard Template Library (STL) arbeiten.

OpenCV ist modular aufgebaut und bietet unter anderem folgende wichtige Module an:

- **core** - Dieses Modul definiert einen Großteil der grundlegenden Funktionen und Datenstrukturen von OpenCV (wie z. B. die `cv::Mat`-Klasse beschrieben in 2.1).
- **imgproc** - Das Image-Processing Modul implementiert Filter (sowohl lineare Filter als auch nicht lineare Filter), geometrische Bildtransformationen (lineare, linear-affine und perspektivische Transformationen), Histogramme und Methoden zur Konvertierung zwischen verschiedenen Farträumen.
- **video** - Modul zur Videoanalyse, welches unter anderem Object-Tracking und auch Motion-Estimation Algorithmen implementiert.
- **calib3d** - Implementiert Algorithmen zur Kamerakalibrierung und Elemente zur 3D-Rekonstruktion.
- **features2d** - Bietet Algorithmen zum Erkennen und Matchen von Features in 2D an.
- **objdetect** - Erkennung von Objekten vordefinierter Klassen (z. B. Gesichtserkennung)
- **highgui** - Bietet dem Programmierer Möglichkeiten an, simple grafische Benutzungsschnittstellen zu erstellen.

2.1 MAT - DER OPENCV BILD-CONTAINER

Die OpenCV Programmzbibliothek war ursprünglich eine Programmzbibliothek für die Programmiersprache C. Damit wurden die Datenstrukturen der Bibliothek als C-Structs implementiert, wodurch der Programmierer selber sicherstellen musste, dass der Speicher für diese Datenstrukturen richtig allokiert, deallokiert und gegebenenfalls auch kopiert wird.

Mit der Version OpenCV 2.0 wurde erstmals eine neue C++ Schnittstelle implementiert. Auf Basis der objektorientierten Prinzipien von C++ wie z. B. Klassen, Konstruktoren (auch Copy- und Move-Konstruktoren), Destruktor, Resource Acquisition Is Initialization (RAII), Operatorüberladung und Template-Programmierung, somit entstand ein abstrakteres und damit auch einfacheres Interface für den Umgang mit Bildern in OpenCV. Die Klasse `cv::Mat` ist ein wichtiger Bestandteil dieser Schnittstelle.

Die `cv::Mat` Klasse stellt eine n -dimensionale *dense* Matrix da. Im Gegensatz zu einer *sparse* Matrix (im OpenCV durch die Klasse `cv::SparseMat` implementiert), welche nur Elemente $\neq 0$ speichert, werden bei `cv::Mat` alle Elemente gespeichert. Die Elemente können sowohl Single-Channel bzw. skalar (z. B. Intensität bei Graustufenbildern) oder Multi-Channel bzw. vektorwertig (z. B. Intensität einzelner Farbchannel) sein.

Bei n -Dimensionen und Dimensionsgrößen (m_1, \dots, m_n) ergeben sich somit

$$N = \prod_{k=1}^n m_k$$

Elemente, welche intern kontinuierlich in einem N -Element großen (eindimensionalen) Array gespeichert sind. Der (nullbasierte) Array-Index j eines Elementes mit den Dimensions-Indizes (i_1, \dots, i_n) lässt sich dann durch folgende rekursive Relation berechnen[2] (mit $j = j_n$):

$$j_k = \begin{cases} j_{k-1} \cdot m_k + i_k, & k \neq 0 \\ 0 & k = 0 \end{cases}$$

Im zweidimensionalen Fall ($n = 2$) reduziert sich dies zu $j = i_1 \cdot m_2 + i_2$. Mit i_1 als Zeilenindex, i_2 als Spaltenindex und der Anzahl an Spalten m_2 . Somit werden zweidimensionale Matrizen also Zeile-für-Zeile gespeichert, dreidimensionale Matrizen zunächst Ebene-für-Ebene und dann jede Ebene wieder Zeile-für-Zeile.

Dieses Speicherlayout ist üblich für *dense* Arrays und ist kompatibel mit anderen Bibliotheken, die ein solches Speicherlayout voraussetzen. Dazu gehört auch die C++ STL. Diese Kompatibilität ermöglicht es nicht nur, STL-Algorithmen auf die Daten eines `cv::Mat`-Objektes anzuwenden, sondern es bietet sich auch die Möglichkeit an, selbst allokierte Daten in ein `cv::Mat`-Objekt zu wrappen und diese Daten dann mit OpenCV spezifischen Methoden zu bearbeiten.

Die `cv::Mat`-Klasse ist stark an die Matrizen aus MATLAB angelehnt, womit OpenCV die Möglichkeit bietet Matrizen im MATLAB-Style zu initialisieren mit z. B. `cv::Mat::zeros()` um alle Elemente der Matrix mit 0 zu

initialisieren, `cv::Mat::ones()` um alle Elemente mit 1 zu initialisieren und `cv::Mat::eye()` zur Konstruktion einer Einheitsmatrix.

Die Datenstrukturen von OpenCV (und damit auch `cv::Mat`) implementieren bereits die nötigen Maßnahmen zur Speicherverwaltung, womit dies nicht vom Programmierer übernommen werden muss. Dabei sei jedoch zu beachten, dass `cv::Mat` jedoch nicht wie die meisten C++ Datenstrukturen mit dynamischem Speicher (z. B. `std::vector`) implementiert ist.

Ein `std::vector` wird beim Kopieren (entweder beim Initialisieren eines anderen Vectors mit dem Kopier-Konstruktor oder durch den Assignment-Operator) komplett kopiert und bekommt seinen eigenen Speicher, der nun die gleichen Daten enthält wie der Originalvector. Es wird kein Speicher zwischen den beiden Objekten geteilt und jedes Objekt ist für seinen eigenen Speicher zuständig. Um die lineare Laufzeit beim Kopieren zu umgehen, wo diese nicht nötig ist, wird Referenzübergabe verwendet. Auch das Kopieren von temporären Objekten, sogenannten `rvalues` (da solche Objekte bei nur links vom Assignment-Operator stehen dürfen), lässt sich seit C++11 mithilfe von Move-Semantiken in konstanter Zeit durchführen.

Im Gegensatz dazu teilen `cv::Mat`-Objekte ihre Ressourcen (den darunterliegenden Speicher) gegebenenfalls mit anderen `cv::Mat`-Objekten. Dazu implementiert `cv::Mat` Reference-Counting und arbeitet somit ähnlich wie der Smart-Pointer `std::shared_ptr` der C++11-Library oder auch wie Objekte in anderen Programmiersprachen wie Java oder C#. Dies hat zu Folge, dass die Destruktoren von `cv::Mat`-Objekten zunächst die Reference-Count dekrementieren und den Speicher erst dann wieder freigeben, wenn die Reference-Count mit dem destruktiven des letzten Objektes auf 0 gefallen ist. Der Vorteil hierbei ist, dass das Kopieren einer Matrix genau genommen die Daten nicht wirklich kopiert, sondern lediglich den Matrix-Header (mit Metadaten wie Dimensionsgrößen und der Adresse zum Speicher der Daten) übernimmt und den Reference-Counter inkrementiert. Das Kopieren einer Matrix ist also unabhängig von der Größe der Matrix und somit konstant. Um eine Matrix vollständig zu kopieren bietet OpenCV in der `cv::Mat`-Klasse die Methode `cv::Mat::clone()` an.

Die Kopiersemantik von `cv::Mat` ist dann von wichtiger Bedeutung, wenn es gilt, einen Teilbereich einer Matrix zu extrahieren oder zu ändern. Solche Bereiche werden als Region of Interest (ROI) bezeichnet und können eine oder mehrere Zeilen, eine oder mehrere Spalten, eine Diagonale oder ein rechteckiger Bereich aus der Matrix sein. Diese Operationen sind weiterhin alle $\Theta(1)$, da lediglich ein neuer Matrix-Header erstellt werden muss und die eigentlichen Elemente der Matrix nicht kopiert werden, sondern nur referenziert werden. Damit wirken sich Änderungen der Daten in einer Matrix also auch implizit auf alle anderen Matrizen aus, die diese Daten referenzieren.

2.2 SATURATE-CASTING

Die einzelnen Pixels eines Bildes sind in OpenCV Elemente einer zweidimensionalen `cv::Mat`. Ob diese Elemente skalar oder vektorwertig sind hängt vom gewählten Farbraum ab. Die Wertebereiche der einzelnen Channel eines

Elementes hängen wiederum vom gewählten Datentyp ab (vgl. 2.3). Dabei werden oftmals 8- oder 16-bit (signed oder unsigned) per Channel gewählt. Für unsigned 8-bit (Datentyp `cv::uchar` in OpenCV) stehen damit nur ganzzählige Werte aus dem (halboffenen) Intervall [0, 256) zu Verfügung. Viele Operationen auf Bildern (z. B. das Interpolieren von Bildern oder das Konvertieren zwischen verschiedenen Farbräumen) können Werte ausserhalb dieses Intervalles erzeugen. Anstatt jedoch nur die niedrigsten 8 Bits des Ergebnisses zu verwenden (was in einem Bild unmittelbar zu sichtbaren Artefakten führt), bietet OpenCV einen `cv::saturate_cast<>()` an, welcher den berechneten Wert auf den nächsten Wert im zugelassenen Wertebereich vom Datentyp abbildet. Dazu wird der Wert zunächst gerundet. Falls der gerundet Wert ausserhalb des gültigen Wertebereiches liegt, wird dieser auf den nächsten Wert gültigen Wert gesetzt (jeweils auf das Minimum oder Maximum der Intervalls, je nachdem ob der Wert auf dem Zahlenstrahl links oder rechts vom Intervall liegt). Der letzte Schritt wird auch als *Clamping* bezeichnet. Eine mögliche Implementation[3] für einen 8-bit Channel mit einem berechneten Wert v ist

$$v' = \min\{\max\{ \lfloor v + 0.5 \rfloor, 0 \}, 255\}$$

2.3 DATENTYPEN FÜR PIXEL

Anstatt die Wahl des Datentyps für Pixel generisch zu halten (mithilfe von C++ Templates), gibt OpenCV eine feste limitierte Menge an primitiven Datentypen für Matrizen vor.[3] Der Grund dafür ist, dass große Template Klassen die Compilezeit und die Größe des Codes stark erhöhen. Außerdem lassen sich Templates nur schlecht in Definition und Implementation aufteilen (wie es sonst mit Header- und Source-Dateien üblich ist). Des Weiteren besitzen die anderen Sprachen, welche von OpenCV unterstützt werden (Python, MATLAB und Java), keine oder nur begrenzte Sprachkonstrukte, welche die Implementation von Templates ermöglichen würde. Stattdessen basiert die aktuelle OpenCV Implementation hauptsächlich auf Polymorphismus. In C++ stehen aus Performancegründen (um die dynamische Laufzeitbindung von polymorphen Methoden zu vermeiden) jedoch einzelne Template Klassen, Methoden und Funktionen zur Verfügung (dazu gehört auch der `cv::saturate_cast<>()`).

In OpenCV stehen folgende primitive skalare Datentypen für Matrizen zur Verfügung:

- 8-bit unsigned Integer - `cv::uchar`
- 8-bit signed Integer - `cv::schar`
- 16-bit unsigned Integer - `cv::ushort`
- 16-bit signed Integer - `cv::short`
- 32-bit signed Integer - `cv::int`
- 32-bit floating-point Zahl - `cv::float`

- 64-bit floating-point Zahl - `cv::double`

Für Multi-Channel Matrizen wird vorausgesetzt, dass jeder Channel den gleichen Datentyp besitzt (außerdem muss dieser natürlich einer der oben aufgeführten Datentypen sein). Des Weiteren ist die maximale Anzahl an möglichen Channels per Matrixelement auf 512 begrenzt.

Die oben aufgeführten Typen werden zwar als Template-Argument für einzelne Funktionen von OpenCV verwendet (z. B. für den bereits erwähnten `cv::saturate_cast<>()`), spezifizieren aber nicht den Datentyp für Matrixelemente (da `cv::Mat` schließlich nicht generisch ist). Stattdessen besitzen diese Klassen dann einen weiteren Parameter im Konstruktur, mit dem der Datentyp angegeben werden kann. Für die skalaren Typen gilt dabei folgende Enumeration:

```
enum {
    CV_8U = 0,
    CV_8S = 1,
    CV_16U = 2,
    CV_16S = 3,
    CV_32S = 4,
    CV_32F = 5,
    CV_64F = 6
};
```

Die Namen für Multi-Channel Konstanten mit $k = 1 \dots 4$ Channels entsprechen dem Namen des Datentyps eines Channels (entsprechend der obigen Enumeration), gefolgt von einem Ck . Ein Drei-Channel Array vom Typen 16-bit unsigned Integer entspricht also der Konstanten `CV_16UC3`.

Um einen Matrix mit mehr als vier Channels zu initialisieren oder falls die Anzahl der Channel bei der Compilezeit unbekannt ist, lassen sich die Makros `CV_U8C(n)` bis `CV_64FC(n)` verwenden. Genau genommen generieren diese Makros lediglich eine numerische Konstante, die intern eine Kombination von Datentyp und Anzahl an Channels identifiziert. Dazu wird die Konstante des Datentyps (entsprechend den Werten in der Enumeration) in die niedrigsten 3 Bits gehschrieben und die Anzahl der Channel dekrementiert (da 0 Channel sowieso nicht zugelassen sind, lässt sich dadurch eine Möglichkeit mehr darstellen) und in die restlichen Bits geschrieben. Dementsprechend sind für die Anzahl der Channels standardmäßig $\log_2(512) = 9$ Bits vorgesehen. Die Berechnung der numerischen Konstante eines Multi-Channel Datentyps ist in OpenCV mit dem Makro `CV_MAKETYPE(depth, cn)` implementiert, dabei entspricht `depth` der numerischen Konstante des Datentyps entsprechend der Enumeration und `cn` die Anzahl der Channel (`cn > 0`). Eine mögliche Implementation nach [4] dieses Makros sieht folgendermaßen aus:

```
#define CV_MAKETYPE(depth, cn) (((depth) & 7) | (((cn)-1) << 3))
```

Damit ergeben sich viele äquivalente Möglichkeiten, den Datentyp einer Matrix festzulegen. So gilt z. B. `2 == CV_16U == CV_16UC1 == CV_16UC(1) == CV_MAKETYPE(CV_16U, 1) == (2 & 7) | ((1 - 1) « 3) == 2`, was mit der ursprünglichen Definition aus der Enumeration übereinstimmt.

Was genau die Werte eines einzelnen Elementes darstellen, hängt vom Farbraum des Bildes ab. Beim Einlesen eines Bildes in OpenCV mit der Funktion `cv::imread()` lässt sich als Parameter mit angeben, ob dieses Bild als

Graustufenbild gelesen werden soll (`IMREAD_GRAYSCALE`) oder als Farbbild (`IMREAD_COLOR`). Als Rückgabewert liefert die Funktion eine (zweidimensionale) Matrix in den Dimensionen des ursprünglichen Bildes. Jedes Element der Matrix stellt einen Bildpunkt des Bildes in dem entsprechenden Farbraum (wird dieser nicht explizit angegeben, so wird implizit der Parameter `IMREAD_COLOR` verwendet) dar. Bei Graustufenbildern bestehen diese Elemente nur aus einem Channel (der Intensität dieses Bildpunktes), während bei Farbbildern das BGR-Format verwendet wird. Hierbei steht der Wert jedes Channels (von insgesamt drei) für die Intensität der entsprechenden Farbe (**Blau**, **Grün** oder **Rot**) im Bildpunkt. Das BGR-Format ist abgesehen von der Reihenfolge der Farbchannel identisch zu dem bekannten RGB-Format.

Um Bilder zwischen verschiedenen Farträumen zu konvertieren lässt sich die OpenCV Funktion `cv::cvtColor()` verwenden. Es lässt sich als Parameter angeben, welche Art von Konvertieren stattfinden soll (z. B. `COLOR_BGR2GRAY`).[5]

2.4 BEISPIEL: VERWENDUNG VON MATRIZEN UND REGION OF INTERESTS

Es folgt ein Codeausschnitt, der die Eigenschaften von `Mat` nach Abschnitt 2.1 nochmals aufzeigt.

```

1 cv::Mat a(256, 256, CV_8UC3, cv::Scalar(100, 100, 100));
2 cv::Mat b = a;
3 cv::Mat c = a.clone();
4
5 a.colRange(50, 80).setTo(cv::Scalar(255, 0, 0));
6 b.diag().setTo(cv::Scalar(0, 255, 0));
7 c({ 0, 20 }, { 0, 20 }).setTo(cv::Scalar(0, 0, 255));
8
9 cv::namedWindow("a");
10 cv::imshow("a", a);
11
12 cv::namedWindow("b");
13 cv::imshow("b", b);
14
15 cv::namedWindow("c");
16 cv::imshow("c", c);
17
18 cv::waitKey();
19 cv::destroyAllWindows();

```

Listing 2.1: Verwendung von Mat und ROI

Im Beispiel wird zunächst eine zweidimensionale 256×256 Matrix `a` vom Typen `CV_8UC3` angelegt und jeder der drei 8-bit Farbchannel mit 100 initialisiert, was im BGR Farbraum einem Grauton entspricht. Die Matrix benötigt somit insgesamt exakt 192 Kibibyte, um die Farbwerte der Pixel zu speichern (mit Headerdaten ist diese natürlich noch größer). Die Initialisierung von `b` in der darauffolgenden Zeile führt wie in 2.1 beschrieben lediglich dazu, dass die Headerdaten von `a` übernommen werden. Damit befindet sich weiterhin nur ein (komplett graues) Bild im Speicher, welches jedoch von zwei verschiedenen `Mat`-Objekten (`a` und `b`) referenziert wird. Die Initialisierung von Matrix

`c` entspricht hingegen einer tiefen Kopie der Matrix `a`, wodurch nun zwei (gleiche) Bilder im Speicher liegen. Diese Operation muss den kompletten Speicher von `a` kopieren, was zu einem linearen Laufzeitverhalten führt.

Die nächsten Zeilen zeigen, wie ROIs in OpenCV verwendet werden können, um Teilbereiche einer Matrix zu extrahieren und zu bearbeiten. Zeile 5 erstellt eine temporäre Matrix, welche auf den gleichen Speicherbereich von `a` zeigt, die Headerdaten jedoch so definiert werden, dass diese Matrix nur einen Teilbereich von `a` schreiben und lesen kann. Als ROI werden hier die Spaltenvektoren von `a` mit Spaltenindizes (nullbasiert) aus dem Intervall $[50, 80)$ genommen. Damit ergibt sich eine temporäre 256×30 Matrix. Alle Elemente dieser Matrix werden daraufhin blau gefärbt, wodurch natürlich auch die entsprechenden Spalten in `a` betroffen sind und damit auch implizit das Bild der Matrix `b` verändert wird (was schließlich das Bild aus der Matrix `a` ist). Zeile 6 färbt die Hauptdiagonale von `b` $\{b_{ij} : (i, j) \in [0, 256]^2 \wedge i = j\}$ grün (was wiederum die Matrix `a` mit betrifft). Die Matrix `c` hat zu diesem Zeitpunkt immer noch ein rein graues Bild. Zeile 7 zeigt, wie man eine beliebige rechteckige ROI aus einer Matrix extrahiert. Im Beispiel wird eine 20×20 ROI erstellt mit den Matrixelementen $\{c_{ij} : 0 \leq i, j < 20\}$, welche daraufhin rot gefärbt werden. Diese Operation betrifft ausschließlich die Matrix `c`; `a` und `b` werden nicht verändert.

Die nächsten Zeilen stellen den Inhalt der Matrizen dar. Die Ausgabe ist in 2.1 dargestellt. Wie erwartet zeigen die Matrizen `a` und `b` das selbe Bild.

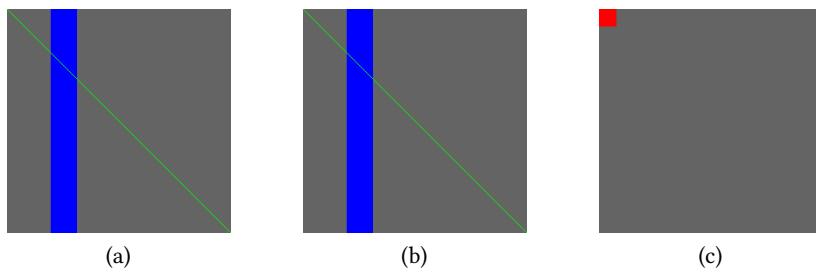


Abbildung 2.1: Ausgabe vom Mat und ROI Beispiel

2.5 TRANSFORMATION VON BILDERN

Um bekannte lineare Transformationen (Skalierung, Rotation, Scherung) und affine Transformationen (lineare Transformationen mit einer Translationskomponente) auf Bildern anzuwenden, bietet OpenCV die Funktion `cv::warpAffine()` an. Neben dem Ausgangsbild, auf welches die Transformation angewendet werden soll, und einem Zielbild, welches das Ergebnis der Transformation halten soll, übernimmt diese Funktion noch eine 2×3 Transformationsmatrix.

Eine affine Transformation $T : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ bildet einen Vektor $\vec{x} = \begin{bmatrix} x \\ y \end{bmatrix} \in \mathbb{R}^2$ auf einen Vektor $T(\vec{x}) \in \mathbb{R}^2$ ab. Eine solche Transformation hat die Form

$$T(\vec{x}) = \mathbf{A}\vec{x} + \vec{b}$$

mit $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \in \mathbb{R}^{2 \times 2}$ und dem Translationsvektor $\vec{b} \in \mathbb{R}^2$.

Für $\vec{b} = \vec{0}$ ist T eine lineare Transformation. Eine solche Transformation erfüllt folgende Bedingung (T ist homogen und additiv): $T(\alpha\vec{u} + \vec{v}) = \alpha T(\vec{u}) + T(\vec{v})$ für ein Skalar $\alpha \in \mathbb{R}$ und zwei Vektoren $\vec{u}, \vec{v} \in \mathbb{R}^2$. Eine Implikation dieser Bedingung ist, dass der Nullvektor stets auf sich selber abgebildet wird: $T(\vec{0}) = T(0 \cdot \vec{e}_1 + 0 \cdot \vec{e}_2) = 0 \cdot T(\vec{e}_1) + 0 \cdot T(\vec{e}_2) = \vec{0} + \vec{0} = \vec{0}$.

Wird \vec{x} in homogenen Koordinaten betrachtet ($\vec{x} = [x \ y \ 1]^\top$), so lässt sich eine Matrix \mathbf{M} zur Transformation T angeben.

$$\begin{aligned} T(\vec{x}) &= \mathbf{A}\vec{x} + \vec{b} \\ T(\vec{x}) &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ T(\vec{x}) &= x \cdot \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} + y \cdot \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix} + 1 \cdot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ T(\vec{x}) &= \underbrace{\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \end{bmatrix}}_{=: \mathbf{M} \in \mathbb{R}^{2 \times 3}} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \end{aligned}$$

Die Funktion `cv::warpAffine()` transformiert das Originalbild dann so, dass

$$dst(\vec{x}) = src(T^{-1}(\vec{x}))$$

[6]

Damit arbeitet die Funktion also mit der inversen Transformation, welche sich mit der Funktion `cv::invertAffineTransform()` berechnen lässt. Standardmäßig wird dies bereits von `cv::warpAffine()` übernommen. Liegt die Transformation jedoch schon invertiert vor, lässt sich diese auch an die Funktion übergeben. Das Setzen von dem `WARP_INVERSE_MAP`-Flag bewirkt dann, dass das Invertieren der Transformation übersprungen wird.

Um Beispielsweise das Bild um 90° (gegen den Uhrzeigersinn) um den Bildmittelpunkt zu rotieren, lässt sich eine Transformation als Komposition von Translation und Rotation angeben. Dabei ist zu beachten, dass der Ursprung des Koordinatensystems sich an der linkeren oberen Ecke des Bildes befindet. Die x-Achse zeigt nach rechts und die y-Achse nach unten. Bei einem Bild mit Breite w und Höhe h gilt $(x, y) \in [0, w) \times [0, h)$ womit sich der Bildmittelpunkt bei $(\frac{w-1}{2}, \frac{h-1}{2})$ befindet.

Die entsprechende Transformationsmatrix \mathbf{M} lautet dann

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & \frac{w-1}{2} \\ 0 & 1 & \frac{h-1}{2} \end{bmatrix} \cdot \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Rotation}} \cdot \begin{bmatrix} 1 & 0 & -\frac{w-1}{2} \\ 0 & 1 & -\frac{h-1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

Die homogene Komponente wird bis zur letzten Transformation erhalten, um die Komposition von affinen und linearen Transformationen zu ermöglichen. Da `cv::warpAffine()` jedoch eine 2×3 Matrix benötigt, wird die letzte Matrix auch als 2×3 Matrix angegeben.



Abbildung 2.2: Rotation als affine Transformation

Um die entsprechende Transformationsmatrix einer beliebigen Rotation direkt zu bekommen, bietet OpenCV die Funktion `cv::getRotationMatrix2D()` an, welche neben dem Rotationswinkel (gemessen in Grad und gegen dem Uhrzeigersinn) auch einen Punkt übernimmt, um den rotiert werden soll.

Da eine affine Transformation Dreiecke auf Dreiecke abbildet, lässt sich eine eindeutige affine Transformation finden, welche eine geordnete Menge von drei nicht-kollinearen Punkten auf eine andere Menge dieser Art abbildet.

OpenCV bietet dafür die Funktion `cv::getAffineTransform()` an. Diese übernimmt drei Punkte (x_i, y_i) , welche auf drei andere Punkte (x'_i, y'_i) abgebildet werden sollen ($1 \leq i \leq 3$). Als Ergebnis liefert diese Funktion eine 2×3 Transformationsmatrix \mathbf{M} . Dabei ist \mathbf{M} genau die Matrix der affinen Transformation, für die

$$\begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} = \mathbf{M} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}, 1 \leq i \leq 3 \quad (2.1)$$

[6]

Mit

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \end{bmatrix}$$

wird dann intern von der Funktion ein 6×6 Gleichungssystem gelöst, um die Koeffizienten von \mathbf{M} herauszufinden. Das Gleichungssystem lautet

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} m_{11} \\ m_{12} \\ m_{13} \\ m_{21} \\ m_{22} \\ m_{23} \end{bmatrix} = \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \\ y'_1 \\ y'_2 \\ y'_3 \end{bmatrix}$$

Alternativ dazu lassen sich auch zwei Matrizen \mathbf{A} , \mathbf{B} definieren mit

$$\mathbf{A} = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} x'_1 & x'_2 & x'_3 \\ y'_1 & y'_2 & y'_3 \end{bmatrix}$$

Die Matrix \mathbf{A} bildet die drei Standardbasisvektoren des \mathbb{R}^3 ($\vec{e}_1, \vec{e}_2, \vec{e}_3$) auf die drei Vertices des Dreiecks ab (in homogenen Koordinaten). Es gilt

also $[x_i \ y_i \ 1]^T = \mathbf{A}\vec{e}_i, 1 \leq i \leq 3$. Allgemein werden alle Vektoren $[\alpha \ \beta \ \gamma]^T$ mit $\alpha + \beta + \gamma = 1$ auf einen Vektor in der Ebene des Dreiecks (also einen Vektor mit 1 als homogene Komponente) abgebildet. Gilt außerdem noch $\alpha, \beta, \gamma \geq 0$, dann ist dies sogar ein Vektor im Dreieck.

Die Matrix \mathbf{B} hat ähnliche Eigenschaften wie \mathbf{A} . Hier wurden lediglich die Vertices des zweiten Dreiecks als Spaltenvektoren der Matrix verwendet. Außerdem fehlt hier jeweils die homogene Komponente in den Spaltenvektoren. Dadurch ist der Ergebnisvektor direkt ein Vektor im \mathbb{R}^2 (ohne homogene Komponente).

Damit lässt sich \mathbf{M} nun folgendermaßen als Komposition dieser beiden Matrizen darstellen:

$$\mathbf{M} = \mathbf{B} \cdot \mathbf{A}^{-1}$$

Mit

$$[x_i \ y_i \ 1] \xrightarrow{\mathbf{A}^{-1}} \vec{e}_i \xrightarrow{\mathbf{B}} [x'_i \ y'_i]$$

ist dies nach 2.1 genau die gesuchte Matrix für die Transformation.

Die folgende Abbildung zeigt das Ergebnis einer solchen Transformation. Im Bild mit der Breite w und Höhe h wird das Dreieck mit den Vertices $(0, 0), (w-1, 0), (0, h-1)$ auf das Dreieck mit den Vertices $(\frac{w-1}{2}, \frac{h-1}{2}), (0, 0), (w-1, 0)$ abgebildet.

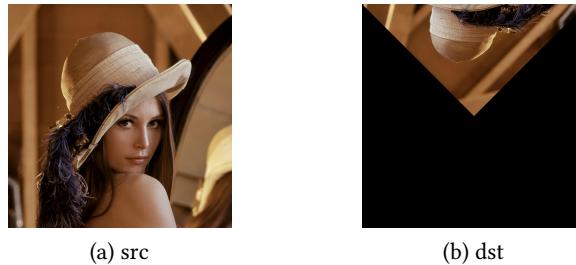


Abbildung 2.3: Abbildung zweier Dreiecke als affine Transformation

Um die Abbildung zwischen zwei allgemeinen (konvexen) Vierecken darzustellen, muss eine perspektivische Transformation gewählt werden. Schließlich bildet eine affine Transformation parallele Linien wieder auf parallele Linien ab, womit Parallelogramme stets auf Parallelogramme abgebildet werden.

Eine solche perspektivische Transformation lässt sich durch eine 3×3 Matrix \mathbf{P} darstellen, welche vier Punkte auf vier andere Punkte abbildet (genau genommen auf ein Vielfaches dieser Punkte), sodass:

$$\begin{bmatrix} \lambda_i x'_i \\ \lambda_i y'_i \\ \lambda_i \end{bmatrix} = \underbrace{\begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix}}_{\mathbf{P}} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}, 1 \leq i \leq 4$$

In OpenCV lässt sich diese Matrix mit der Funktion `cv::getPerspectiveTransform()` finden. Die Funktion `cv::warpPerspective()` übernimmt dann die eigentliche Transformation des Bildes mit der Matrix.

3

METRIKEN ZUR BILDREKONSTRUKTION

Um Schiebepuzzles mit Bildern lösen zu können, muss zunächst das Ausgangsbild (als gewünschten Endzustand) bekannt sein. Der erste Schritt ist somit, dieses Bild aus einer Menge von (rechteckigen) Puzzlestücken zu rekonstruieren. Diese Art von Problemstellung erinnert an eine andere Form von Puzzle, welche im englischen Sprachraum als *Jigsaw Puzzle* bezeichnet werden. Abbildung 3.1 zeigt ein Beispielduzzle dieser Art.



(a) Ausgangspuzzle



(b) Teillösung



(c) Lösung

Abbildung 3.1: Beispiel eines Jigsaw-Puzzles

Normalerweise sind die Puzzlestücke dabei speziell geformt (oftmals ähnlich wie in Abb. 3.1). Dies schränkt die Möglichkeiten der gültigen Zusammensetzungen des Puzzles stark ein, besonders da die Möglichkeit besteht zwischen Rahmenstücken und inneren Puzzlestücken zu unterscheiden (Stücke am Rand des Bildes haben mindestens eine glatte Kante). Dadurch entsteht ein Problem, welches sich teilweise nur durch die äußere Form der Puzz-

lestücke und deren Kompatibilität untereinander lösen lässt. Es gibt sogar Puzzles bei denen diese Information hinreichend ist um das Puzzle vollständig zu lösen (die Stücke selber weisen dann keine weiteren optischen Merkmale auf und sind meist einfarbig).

Mit der Motivation später Schiebepuzzle lösen zu können, muss das Problem auf rechteckige Puzzlestücke verallgemeinert werden. Damit kann weder gesagt werden, ob ein Puzzlestück ein Rahmenstück ist oder sich im inneren des Puzzles befinden muss, noch können Aussagen über die Kompatibilität zweier Puzzlestücke gemacht werden, ohne dabei die Bilder der Puzzlestücke zu analysieren.

Eine noch speziellere Variante von rechteckigen Jigsaw-Puzzles schränkt die Puzzlestücke auf eine quadratische Größe ein. Bei diesen Square Jigsaw Puzzles werden in der Literatur nach [16] zwischen drei Typen unterschieden. Alle Typen haben (sofern nicht anders definiert) Informationen über die Dimensionsgrößen (m Zeilen, n Spalten) des Puzzles. Alle Puzzleteile sind quadratisch (mit der gleichen Seitenlänge) und besitzen Farbinformationen (in Form von einem Bild für jedes Stück). Es gibt exakt $N = m \times n$ Puzzlestücke, womit das Puzzle also keine Lücken oder überschüssige Teile haben wird.

Bei dem Type 1 Square Jigsaw Puzzle ist die Position der Puzzlestücke unbekannt, deren Orientierung jedoch fest und bekannt. Alle möglichen Puzzlestellungen sind dann eine Permutation der vorhandenen Puzzlestücke. Davon gibt es $N!$ verschiedene. Außerdem lassen sich somit zwei Puzzleteile (x_i, x_j) auf vier verschiedene Arten zusammensetzen (x_i kann links, rechts, oben oder unten von x_j platziert werden). Jede Seite von x_i hat nur eine Seite von x_j als möglichen Partner. Das Type 1 Puzzle ist der am meisten untersuchteste Typ von den dreien und hat den Namen *jig swap*[20] bekommen.

Das Type 2 Puzzle lässt eine unbekannte Orientierung der Puzzleteile zu. Somit lassen die Puzzleteile sich nicht nur anordnen, sondern auch rotieren. Die Anzahl der möglichen Anordnungen des Puzzles vervielfacht sich somit um einen Faktor von 4^N auf insgesamt $4^N \cdot N!$. Zwei Puzzlestücke (x_i, x_j) lassen sich nun auf 16 verschiedene Arten anordnen, da jede Seite von x_i mit jeder Seite von x_j kombiniert werden kann.

Beim Type 3 Puzzle ist die Orientierung zwar unbekannt, die Positionen der Puzzleteile sind jedoch fest und bekannt. Da die Anordnung fest ist und jedes Puzzlestück sich rotieren lässt, hat dieses Puzzle mit 4^N möglichen Anordnungen die geringste Komplexität. Zwei Puzzlestücke (x_i, x_j) lassen sich auch hier auf 16 verschiedene Arten kombinieren. Der Unterschied dabei zu Type 1 und Type 2 Puzzles ist jedoch, dass nur die Kompatibilität zwischen Seiten von bereits benachbarten Teilen analysiert werden muss (da die Anordnung fest ist). Bei Type 1 und Type 2 Puzzles kommen für ein Teil x_i alle anderen Teile $x_j, j \neq i$ als mögliche Nachbarn in Frage.

Die Anzahl der möglichen Kombinationen die es gibt, wenn man zwei Seiten von zwei Puzzlestücken vergleichen möchte, ist bereits eine gute untere Schranke für die Laufzeit- und Speicherkomplexität von Algorithmen, welche die Kompatibilität aller Puzzlestücke (in allen möglichen Anordnungen) braucht, um Entscheidungen treffen zu können (um beispielsweise zunächst die Puzzleteile zu kombinieren, welche von allen die höchste Kompatibilität aufweisen).

Type 1 Puzzle müssen alle Puzzlestücke mit allen anderen Puzzlestücken vergleichen. Da die Kompatibilität zwischen zwei Stücken symmetrisch ist und es 4 Möglichkeiten gibt die zwei Stücke in einem Type 1 Puzzle anzugeben ergeben sich somit

$$4 \binom{N}{2} = 2N(N - 1) \in \Theta(N^2)$$

Kompatibilitäten die berechnet werden müssen.

Bei Type 2 Puzzle ändert sich lediglich der konstante Faktor vor dem Binomialkoeffizienten von 4 auf 16, womit sich auch die Anzahl der berechneten Metriken vervierfacht und somit auf $8N(N - 1)$ wächst, was weiterhin $\Theta(N^2)$ ist.

Im Gegensatz dazu werden bei Type 3 Puzzles lediglich die Kompatibilitäten zu den benachbarten Puzzlestücken berechnet. Durch die mögliche Rotation der beiden Stücke ergeben sich (wie bei Type 2) 16 Möglichkeiten pro Paar. Jedes Paar entspricht einer (inneren) Kante im Puzzle (vgl. Abb. 3.2). Die Anzahl der vertikalen Kanten (in der Abbildung rot gekennzeichnet) entspricht $m(n - 1)$, die Anzahl der horizontalen Kanten (in der Abbildung grün gefärbt) entspricht dann der Symmetrie nach $n(m - 1)$. Damit ergeben sich insgesamt

$$\begin{aligned} & 16[m(n - 1) + n(m - 1)] \\ & = 16[2mn - m - n] \\ & = 16(2N - m - n) \in \Theta(N) \end{aligned}$$

mögliche Kompatibilität, die berechnet werden müssen.

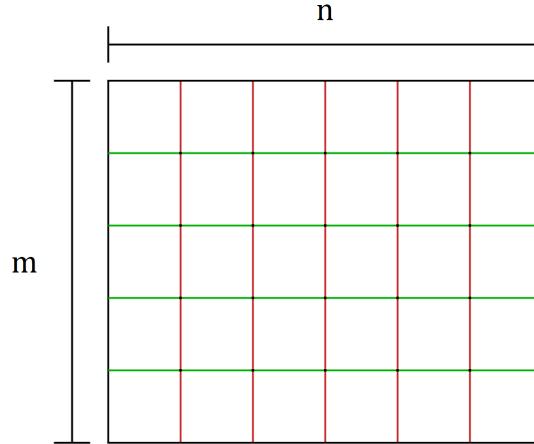


Abbildung 3.2: Anzahl der Kanten bei einem Type 3 Puzzle

Es ist zu beachten, dass N bereits quadratisch wächst (bei einem Puzzle mit quadratischen Dimensionen $m = n$). Bei einem $n \times n$ Puzzle ist der Aufwand eines Type 3 Puzzles somit mindestens quadratisch von n abhängig. Bei Type 1 und Type 2 Puzzles wächst dies noch schneller: $\Theta(N^2) = \Theta(n^4)$.

Außerdem sei noch erwähnt, dass mögliche Lösungen beim Type 2 Puzzle aufgrund der erlaubten Rotation (und der Tatsache, dass die Position der Stücke nicht fixiert ist) nicht unbedingt eindeutig sind. Selbst wenn ein

Algorithmus es schafft das Bild so zu rekonstruieren, dass jede Seite eines Puzzlestückes den richtigen Partner hat, kann es sein, dass das Gesamtbild jedoch rotiert ist.

Im weiteren Verlauf dieser Arbeit werden hauptsächlich Type 1 Puzzles betrachtet, da Schiebepuzzle sowieso keine Rotation zulassen. Abbildung 3.3 zeigt ein 8×8 Type 1 Puzzle.

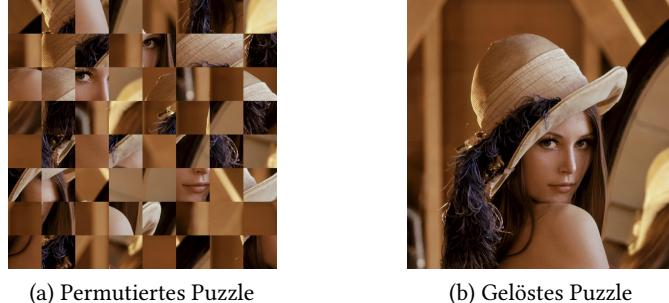


Abbildung 3.3: Beispiel eines Type 1 Puzzles

Zur Rekonstruktion des Bildes werden Kompatibilitätsmetriken betrachtet, welche später als Information vom eigentlichen Algorithmus verwendet werden, um Entscheidungen zu treffen, welche Puzzlestücke miteinander kombiniert werden sollen und welche nicht.

Genau genommen werden im Folgenden eher Metriken (Distanzfunktionen) betrachtet, die genau das Gegenteil von der Kompatibilität zweier Puzzlestücke darstellen. Den Begriff der Kompatibilität wurde jedoch in [13] eingeführt und wird seitdem in weiterer Literatur verwendet.

3.1 $(L_p)^q$ NORM

Um Aussagen über die Kompatibilität bzw. die Unähnlichkeit (mit Hilfe einer Distanzfunktion) zweier Puzzlestücke zu machen, muss zunächst geklärt werden, wie die Distanz zweier Pixel beschrieben werden kann. Welche Pixel wirklich miteinander verglichen werden beim Vergleich zweier Puzzlestücke ist schließlich erst dann relevant, wenn Möglichkeiten bestehen, die Unähnlichkeit von zwei Pixeln anzugeben.

Da weiterhin Farbbilder aus dem (OpenCV üblichen) BGR-Raum betrachtet werden und somit jeder Pixel drei Channel (mit je 8 Bit) hat, lässt sich ein Pixel als einen Vektor aus dem Raum $[0, 256]^3$ auffassen. Damit lässt sich eine Norm $\|\cdot\| : [0, 256]^3 \rightarrow [0, \infty)$ definieren. Mit einer Norm lässt sich aber mit $d(\vec{u}, \vec{v}) := \|\vec{u} - \vec{v}\|$ eine Metrik einführen.

Die aus der Mathematik bekannte L_p Norm mit

$$L_p(\vec{v}) = \|\vec{v}\|_p := \sqrt[p]{\sum_{i=1}^n |v_i|^p}, p \in \mathbb{R}^{\geq 1}$$

liegt dabei am nächsten. Besonders die L_2 Norm (euklidische Norm) oder die L_1 Norm (Taxicab Norm oder Manhatten Norm) würden in Frage kommen.

In [13] wurde jedoch für Jigsaw Puzzles mit der $(L_p)^q$ Norm eine weitere (allgemeinere) Norm eingeführt. Die $(L_p)^q$ Norm berechnet sich zu

$$(L_p)^q[\vec{v}] = \|\vec{v}\|_p^q := \sqrt[q]{\sum_{i=1}^n |v_i|^p}$$

Die $(L_p)^q$ Norm ist damit die mit q potenzierte L_p Norm: $(L_p)^q[\vec{v}] := (L_p[\vec{v}])^q$. Auch in [13] (empirisch) gezeigt wurde, dass besonders $p = \frac{3}{10}$ und $q = \frac{1}{16}$ gute Ergebnisse liefern.

3.2 DISSIMILARITY-BASED COMPATIBILITY (DBC)

Die erste Metrik beschrieben in [13] ist die Dissimilarity-Based Compatibility (DBC). Diese summiert die Distanzen zweier Puzzlestücke entlang einer Kante auf. Da es für Type 1 Puzzle vier verschiedene Möglichkeiten gibt, zwei Stücke x_i und x_j anzugeordnen, wird zunächst eine numerische Konstante $r \in \{0, 1, 2, 3\}$ definiert, die diesen Zusammenhang darstellt.

$r = 0$	$r = 1$	$r = 2$	$r = 3$
x_i x_j	x_i x_j	x_j x_i	x_j x_i

Tabelle 3.1: Räumlicher Zusammenhang zweier Puzzlestücke

Die DBC entlang einer Kante, die beim Zusammensetzen zweier Puzzlestücke entsteht, wird dann mit $D(x_i, x_j, r)$ bezeichnet. Da diese Metrik symmetrisch ist gilt außerdem $D(x_i, x_j, r) = D(x_j, x_i, r')$. Wobei r' die inverse Relation der beiden Puzzlestücke darstellt und sich zu

$$r' = (r + 2) \bmod 4$$

berechnet.

Als Referenzstellung der beiden Puzzleteile wird im weiteren Verlauf $r = 0$ verwendet (x_i befindet sich also links von x_j). Andere Stellungen lassen sich entweder ähnlich berechnen (lediglich durch das Vertauschen der Dimensionen oder durch das Vertauschen von i und j) oder sogar gleich berechnen, sofern die beiden Puzzlestücke zuvor entsprechend rotiert wurden.

Diese DBC zweier Puzzlestücke $x_i(x, y)$ und $x_j(x, y)$ mit Höhe h und Breite w und basierend auf einer Norm $\|\cdot\|$ berechnet sich dann zu:

$$D(x_i, x_j, 0) = \sum_{k=0}^{h-1} \|x_i(w-1, k) - x_j(0, k)\| \quad (3.1)$$

Da diese Distanz bei rechteckigen Puzzlestückchen die kürzeren Seiten generell bevorzugt (die Summanden sind stets positiv, womit selbst gute Kanten bei vielen Summanden große Werte liefern), kann diese Distanz noch durch

die Anzahl der Summanden bzw. die Länge der Seite K (bei $r = 0$ beispielsweise die Höhe der Puzzlestücke) dividiert werden:

$$\bar{D}(x_i, x_j, r) = \frac{1}{K} \cdot D(x_i, x_j, r)$$

Um diese Distanz gegebenenfalls noch auf das Intervall $[0, 1]$ zu normalisieren, lässt sich die durchschnittliche Distanz (pro Summand) durch die maximale Distanz d_{max} zwischen zwei Pixel dividieren. Diese maximale Distanz hängt von der gewählten Norm ab. Bei der euklidischen Norm ist dies beispielsweise $d_{max} = \sqrt{255^2 + 255^2 + 255^2} = \sqrt{3} \cdot 255 = \sqrt{3} \cdot 255$.

Damit lässt sich nun die Kompatibilität zweier Puzzlestücke als die Gegenwahrscheinlichkeit dieser relativen Distanz definieren:

$$C(x_i, x_j, r) := 1 - \frac{\bar{D}(x_i, x_j, r)}{d_{max}}$$

Die DBC bietet eine simple Möglichkeit die Kompatibilität (oder Distanz) zwischen zwei Puzzlestücken zu berechnen. Die Berechnung einer solchen Distanz bezüglich einer Kante ist linear in der Länge dieser Kante $\Theta(K)$ (Höhe oder Breite der Puzzlestücke). Ein Nachteil ist, dass lediglich die äußersten Pixel eines Puzzlestückes betrachtet werden. Dadurch werden Farbübergänge, die zwar im Gesamtbild Sinn ergeben, an den Kanten als harsch angesehen, da diese dort für große Distanzen sorgen.

3.3 PREDICTION-BASED COMPATIBILITY (PBC)

Um neben den Randpixeln auch noch die Informationen von deren (inneren) benachbarten Pixeln nutzen zu können, wird bei der Prediction-Based Compatibility (PBC) das Bild eines Puzzlestückes um einen Pixel in jede Richtung erweitert. Diese approximierten Pixel einer Seite werden dann anstatt den eigentlichen Pixeln dieser Seite (wie es bei der DBC der Fall ist) in der Distanzberechnung verwendet. Es wird also die Distanz zwischen einem approximierten Pixel in einem Puzzlestück und einem echten Pixel in einem anderen Puzzlestück ermittelt.

Zur Approximation der Farbwerte lässt sich ein Bild I mit der Breite w und der Höhe h im BGR-Farbraum wieder als vektorwertige Funktion $\vec{I}(\vec{x}) : [0, w) \times [0, h) \rightarrow [0, 256]^3$ betrachten. Es wird eine lineare Approximation nach Taylor gewählt. Um eine Entwicklungsstelle \vec{x}_0 gilt dann

$$\vec{I}(\vec{x}) \approx \vec{I}(\vec{x}_0) + J_I(\vec{x}_0) \cdot (\vec{x} - \vec{x}_0)$$

Wobei mit $J_I(\vec{x}_0)$ die Jacobi-Matrix (ausgewertet an der Entwicklungsstelle \vec{x}_0) gemeint ist. Die Jacobi-Matrix fasst alle partiellen Ableitungen der Funktion zusammen (die Komponenten von I werden entsprechend der Farbchannel als b , g und r bezeichnet):

$$J_I = \begin{bmatrix} \frac{\partial b}{\partial x} & \frac{\partial b}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \\ \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} \end{bmatrix}$$

Um die einzelnen Ableitungen numerisch zu bestimmen, werden finite Differenzen verwendet. Dabei wird unterschieden zwischen dem Forward-Differenzenquotienten $\Delta_F[f](x)$, Backward-Differenzenquotienten $\Delta_B[f](x)$ und Central-Differenzenquotienten $\Delta_C[f](x)$.

$$\begin{aligned}\Delta_F[f](x) &= \frac{f(x+h) - f(x)}{h} \\ \Delta_B[f](x) &= \frac{f(x) - f(x-h)}{h} \\ \Delta_C[f](x) &= \frac{\Delta_F[f](x) + \Delta_B[f](x)}{2} \\ &= \frac{f(x+h) - f(x-h)}{2h}\end{aligned}$$

Der Central-Differenzenquotient ist damit also das arithmetische Mittel vom Forward-Differenzenquotienten und vom Backward-Differenzenquotienten. Da die Ableitung einer Funktion f an einer Stelle x gerade der Grenzwert dieser Quotienten darstellt, für die $h \rightarrow 0$, werden zur numerischen Berechnung kleine $h \neq 0$ gewählt. Die Koordinaten (x, y) eines Bildes können jedoch nur diskrete Werte annehmen, womit das kleinstmögliche h somit $h = 1$ ist. Das Aufstellen der Differenzenquotienten für vektorwertige Funktionen lässt sich analog zum skalaren Beispiel konstruieren (wobei die Funktionsdifferenz im Zähler dann eine Vektordifferenz ist, anders gesagt wird der Differenzenquotient für jede Komponente gebildet). Für Funktionen, die von mehreren Variablen abhängig sind, entsprechen die Differenzenquotienten dann einer Approximation der partiellen Ableitungen, wobei lediglich der Parameter variiert wird, nach dem abgeleitet werden soll. Die restlichen Parameter werden konstant gehalten.

Als Beispiel lässt sich somit ein Bild an den Rändern erweitern, indem der nächstgelegene Pixel als Entwicklungsstelle für die lineare Approximation genommen wird. In Abbildung 3.4 wird gezeigt, wie ein Bild sowohl nach oben als auch nach rechts um jeweils einen Pixel erweitert wird.

Ein Pixel am (erweiterten) rechten Rand (in der Zeile y) des Bildes hat die Koordinaten (w, y) . Als Entwicklungsstelle wird der links benachbarte Pixel mit den Koordinaten $(w-1, y)$ gewählt. Der Farbwert des neuen (erweiterten) Pixels lässt sich dann mit

$$\begin{aligned}\vec{I}(w, y) &= \vec{I}(w-1, y) + \left[\frac{\partial \vec{I}}{\partial x} \quad \frac{\partial \vec{I}}{\partial y} \right] \Big|_{(x,y)=(w-1,y)} \cdot \left(\begin{bmatrix} w \\ y \end{bmatrix} - \begin{bmatrix} w-1 \\ y \end{bmatrix} \right), \quad \forall y : 0 \leq y < h \\ &= \vec{I}(w-1, y) + [\vec{I}_x(w-1, y) \quad \vec{I}_y(w-1, y)] \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= \vec{I}(w-1, y) + \vec{I}_x(w-1, y)\end{aligned}\tag{3.2}$$

approximieren.

Bei der numerischen Approximation der partiellen Ableitung von \vec{I} entlang der x -Achse kommt ausschließlich der Backward-Differenzenquotient in Frage. Dies liegt daran, dass die Pixel mit einer x -Koordinate von $w-1$

sich in der letzten Spalte des Bildes befinden und somit entlang der x -Achse keinen rechts benachbarten Pixel haben. Damit ergibt sich:

$$\begin{aligned}\vec{I}_x(w-1, y) &= \Delta_B \left[\vec{I} \right] (w-1, y) \\ &= \frac{\vec{I}(w-1, y) - \vec{I}(w-1-h, y)}{h}\end{aligned}$$

Mit $h = 1$ folgt

$$\vec{I}_x(w-1, y) = \vec{I}(w-1, y) - \vec{I}(w-2, y) \quad (3.3)$$

Insgesamt ergibt sich mit 3.2 und 3.3 die vollständige Formel zu Berechnung des erweiterten Randes vom Bild.

$$\vec{I}(w, y) = 2\vec{I}(w-1, y) - \vec{I}(w-2, y) \quad (3.4)$$

Die Formulierung einer solchen Approximierung für Pixel an anderen Seiten des Bildes lässt sich ähnlich angeben. Lediglich bei der Approximierung der Pixel in einer (neuen) Ecke des Bildes müssen beide partiellen Ableitungen approximiert werden (als Entwicklungsstelle wird die alte Ecke des Bildes genommen, womit sowohl ein Schritt entlang der x -Achse, also auch einen Schritt entlang der y -Achse gegangen wird).

Außerdem ist noch zu beachten, dass mit der reinen Berechnung nach 3.4 noch nicht sichergestellt ist, dass die Channel des neuen Pixels sich auch wieder in dem Wertebereich $[0, 256]$ befinden. Es gilt also zu beachten, dass die Berechnung keine modulare Arithmetik verwenden sollte (dazu sollte die Berechnung in einem größeren Vorzeichen behafteten Datentyp stattfinden). Um die Werte danach wieder auf das Intervall $[0, 256]$ zu bringen, sollte dann unbedingt der in 2.2 angesprochene `cv::saturate_cast<>()` verwendet werden.

Der folgende Ausschnitt an Quelltext zeigt, wie die Berechnung nach 3.4 mit einem Quellbild `src` und einem Zielbild `dst` in OpenCV umgesetzt werden kann.

```

1 for (size_t row = 0; row < src.rows; row++)
2 {
3     for (size_t ch = 0; ch < 3; ch++)
4     {
5         dst.at<cv::Vec3b>(row, src.cols)[ch] =
6             cv::saturate_cast<cv::uchar>(
7                 2 * dst.at<cv::Vec3b>(row, src.cols - 1)[ch]
8                 - dst.at<cv::Vec3b>(row, src.cols - 2)[ch]
9                 )
10    ;
11 }
12 }
```

Listing 3.1: Lineare Approximierung (nach Taylor) von Pixel am rechten Bildrand

Um den `cv::saturate_cast<>()` geeignet anwenden zu können, werden die Pixel im Quelltext Channel für Channel durchgegangen (Schleife in Zeile 3). Außerdem wird mit der Multiplikation der numerischen Konstante 2 in Zeile 7 die komplette Berechnung implizit zu einer Berechnung mit Integer

Werten. Wäre dies hier nicht der Fall, würde auch die Saturierung des berechneten Wertes nichts daran ändern, dass bereits bei der Berechnung selber gegebenenfalls Over- oder Underflows aufgetreten sind.

Abbildung 3.4 zeigt neben dem Ausgangsbild und der oberen rechten Ecke dieses Bildes, zwei Varianten zur Berechnung der Approximierung.

In 3.4c wurde der Saturate-Cast weggelassen, wodurch der (zunächst richtig) berechnete Wert bei der Zuweisung an einen kleineren Datentypen die Informationen in den höheren Bits verliert. Als Ergebnis entstehen einzelne Pixel, die stark vom eigentlichen Bildverlauf abweichen. Suggestieren zwei benachbarte Pixel etwa einen immer dunkler werdenden Farbverlauf, so kann es vorkommen, dass der approximierte Pixel als so dunkel angenommen wird, dass die Werte der Channel bei der Berechnung negativ werden. Diese Annahme ist nicht falsch. Ein impliziter Cast zu einem uchar bewirkt dann jedoch, dass die Channel sehr große Werte annehmen. Das bereits beschriebene Clamping vom Saturate-Cast verhindert dies, sodass ein überaus dunkler Pixel in jedem Channel gegebenenfalls auf den dunkelsten Wert (welcher 0 entspricht) gesetzt wird. Die gleiche Berechnung mit einem Saturate-Cast liefert, wie in 3.4d zu sehen ist, eine zum Farbverlauf passende und somit realistisch aussehende Approximierung der neuen Randpixel.



Abbildung 3.4: Lineare Approximierung der Pixel am rechten und am oberen Bildrand

Um diese Art der Approximierung nun zu nutzen um die Kompatibilität zweier Puzzlestücke herauszufinden, wird ähnlich wie in 3.2 vorgegangen. Als Referenzstellung wird wieder $r = 0$ gewählt, womit sich ein Puzzlestück x_i links von einem anderen Puzzlestück x_j befindet. Dabei wird die Berechnung diesmal jedoch in zwei Teile eingeteilt. Es wird zunächst die Distanz betrachtet, die entsteht wenn man die rechte Seite von x_i approximiert und mit den (echten) Pixeln der linken Seite von x_j vergleicht. Diese Distanz D_L berechnet sich dann analog zu der DBC in 3.1. Der einzige Unterschied besteht darin, dass die approximierte Spalte an Pixeln von x_i genommen wird. In der Formel aus 3.1 wird also das $x_i(w - 1, k)$ zu einem $x_i(w, k)$ geändert, was für die Pixel in der (nicht existierenden, aber approximierten) Spalte von x_i steht. Dann ergibt sich:

$$D_L(x_i, x_j, 0) = \sum_{k=0}^{h-1} \|x_i(w, k) - x_j(0, k)\|$$

Entsprechend der Approximierung in 3.4 lässt sich $x_i(w, k)$ aber als die Linearkombination zweier echten Pixel von x_i berechnen

$$D_L(x_i, x_j, 0) = \sum_{k=0}^{h-1} \|2x_i(w - 1, k) - x_i(w - 2, k) - x_j(0, k)\| \quad (3.5)$$

Um weiterhin garantieren zu können, dass die Distanz zwischen zwei Puzzlecken symmetrisch ist, wird die Gesamtdistanz zweier Puzzlestücke als die Summe beider Distanzen der Approximierungen gebildet. Dazu wird nun $D_R(x_i, x_j, 0)$ ähnlich wie in 3.5 berechnet.

$$D_R(x_i, x_j, 0) = \sum_{k=0}^{h-1} \|x_i(w - 1, k) - 2x_j(0, k) + x_j(1, k)\| \quad (3.6)$$

Wurden beide Seiten berechnet, so lässt sich die gesamte Distanz zu $D(x_i, x_j, r) = D_L(x_i, x_j, r) + D_R(x_i, x_j, r)$ berechnen. Damit ist sichergestellt, dass weiterhin der Symmetrie nach $D(x_i, x_j, r) = D(x_i, x_j, (r + 2) \bmod 4)$ gilt.

Da ähnlich wie bei der DBC in 3.2 die Distanzen entlang einer Kante aufsummiert werden, ist die Berechnung einer Distanz linear in der Länge dieser Kante. Bei der PBC kommen entsprechend noch zusätzliche Berechnung zur Approximierung der Pixel dazu. Um die Anzahl der konstanten Schritte möglichst minimal zu halten (im Austausch von zusätzlichem Speicherplatz), lassen sich die approximierten Seiten eines Puzzlestückes nur einmal initial berechnen und dann zwischenspeichern. Damit müssen für jedes der N Puzzlestücke zwei horizontale Kanten der Breite w (oben und unten) und zwei vertikale Kanten der Höhe h (links und rechts) abgespeichert werden. Insgesamt müssen damit $2N(w + h)$ Pixel berechnet und abgespeichert werden (alle jeweils mit drei Channel).

3.4 DYNAMIC TIME WARPING (DTW)

Eine weitere Methode zur Berechnung der Gesamtdistanz zwischen zwei Puzzlecken wurde in [8] beschrieben. Intuitiv betrachtet lässt sich vermuten, dass der Vergleich zweier Pixel sich nicht nur auf unmittelbar benach-

barte Pixel entlang einer Achse (beispielsweise linke und rechte Nachbarn) beschränken sollte.

Abbildung 3.5 zeigt als Beispiel zwei 8×1 Bildränder: R (etwa der rechte Bildrand eines Puzzlestückes x_i) und S (der linke Rand eines anderen Puzzlestückes x_j). Diese sollen beispielsweise nach 3.2 mit der DBC verglichen werden. Wird als Norm die L_1 Norm gewählt (Summe der absoluten Differenzen oder auch Manhatten oder Taxicab-Norm), ergibt sich nach dem aufsummieren dieser 8 Distanzen eine Gesamtdistanz von 2246.

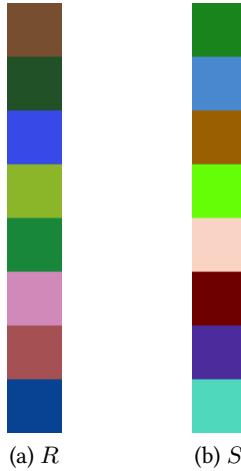


Abbildung 3.5: Direkte Zuordnung der Pixel

Die Farben in Beispiel 3.5 suggerieren bereits, dass eine andere Zuordnung, welche nicht nur ausschließlich direkt benachbarte Pixel vergleicht, eine geringere Distanz ergeben könnte. So würde beispielsweise das Verschieben von S um einen Pixel nach unten dafür sorgen, dass viele ähnliche Pixel (blau mit blau und grün mit grün) verglichen werden könnte.

Ein Algorithmus der dieses Problem behandelt ist der Dynamic Time Warping (DTW) Algorithmus. Ursprünglich wurde der Algorithmus entwickelt, um die best mögliche Zuordnung (die Zuordnung mit der geringsten Distanz bzw. mit der größten Ähnlichkeit) von zwei zeitabhängigen Sequenzen zu bestimmen (die nicht zwingend gleicher Länge sein müssen). Möchte man beispielsweise Aussagen über die Ähnlichkeit zweier Audiosignale machen, in denen zwei verschiedene Personen die gleiche Sequenz von Wörtern vorlesen (ein Anwendungsbeispiel aus dem Bereich der automatische Spracherkennung), so würde ein direkter Vergleich der beiden Signale auf zwei grundlegende Probleme stoßen: Werden ausschließlich die Frequenzen verglichen, die in den Signalen am gleichen Zeitpunkt t auftreten, so müsste vorausgesetzt werden, dass die beiden Signale die gleiche Länge T haben, was aufgrund der unterschiedlichen Sprachgeschwindigkeit von Menschen sehr unwahrscheinlich ist. Und selbst wenn beide Signale eine gleiche Länge aufweisen, so sorgt genau diese Sprachgeschwindigkeit dafür, dass gegebenenfalls ein Signal zeitlich immer leicht verschoben ist. Somit würden zwei ähnliche Signale als komplett verschieden anerkannt werden, weil bei der Zuordnung der Frequenzen nicht beachtet wird, dass gleiche Frequenzen zeitverschoben auftreten können. Abbildung 3.6 zeigt zwei Beispiel Matchings

einer zeitabhängigen Sequenz. Eine direkte Zuordnung (euklidisches Matching) sorgt dafür, dass zeitlich verschobene aber ähnliche Frequenzen nicht einander zugeordnet werden können. Der DTW-Algorithmus findet das beste Matching, indem die Sequenzen so *verzerrt* werden, dass diese Zuordnung der ähnlichen Frequenzen möglich wird.

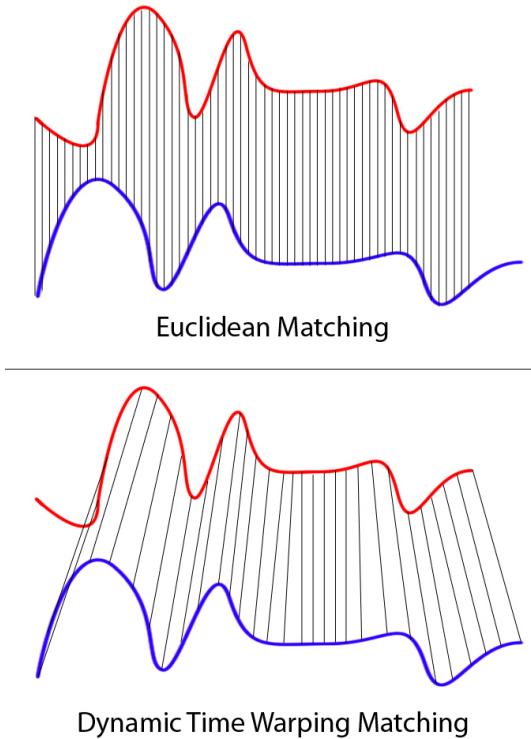


Abbildung 3.6: Euclidean Matching im Vergleich mit DTW

Quelle: Wiki Commons: File:Euclidean vs DTW.jpg (https://commons.wikimedia.org/wiki/File:Euclidean_vs_DTW.jpg)

Im Gegensatz zu zeitabhängigen Sequenzen (welche ein Beispiel für stetige Sequenzen sind) ist die Zuordnung zwischen zwei Pixelsequenzen diskret, da es nur eine endliche Anzahl an Indizes gibt. Bevor der Algorithmus selber beschrieben werden kann, sollte festgelegt werden, welche Eigenschaften eine Zuordnung aufweisen muss um gültig zu sein (bzw. welche Matchings zulässig sind).

Es werden zwei Sequenzen $R = (r_1, \dots, r_m)$ und $S = (s_1, \dots, s_n)$ betrachtet, mit entsprechenden Längen $|R| = m$ und $|S| = n$ (im konkreten Fall, wo R und S Seitenränder von Puzzlestücken sind gilt natürlich $m = n$). Auf den Elementen der Sequenz sei eine Metrik bzw. Distanzfunktion $d(r_i, r_j)$ definiert. Bei den Bildpunkten lässt sich die Matrik mit einer Norm $\|\cdot\|$ nach 3.1 als $d(r_i, r_j) = \|r_i - r_j\|$ definieren. Gesucht ist die minimale Gesamtdistanz D als Summe einzelner Distanzen durch das Matching jeweils zweier Elemente aus den Sequenzen $D(R, S) = \sum_{i,j} d(r_i, r_j)$. Dieses Matching muss folgende Eigenschaften beachten:

1. Jedes Element muss mindestens einen Summanden zur Gesamtdistanz beitragen. Anders gesagt muss jedes Element aus einer Sequenz mit

mindestens einem Element aus der anderen Sequenz verbunden werden. Es existieren also keine Elemente, die bei der Berechnung nicht in die Gesamtdistanz mit einfließen.

2. Ein Matching (r_i, s_j) besagt, dass das Element r_i mit dem Element s_j verbunden wurde (es wird also $d(r_i, s_j)$ aufsummiert). Existiert solch ein Matching (r_i, s_j) , dann darf kein Matching (r_k, s_l) existieren mit $(k < i \wedge l > j) \vee (k > i \wedge l < j)$. Anschaulich bedeutet dies, dass Zuordnungen sich nicht überkreuzen dürfen (wie auch in Abbildung 3.6 zu sehen ist).
3. Zusammen implizieren Punkt 1 und 2 damit außerdem, dass beide Anfangs- und Endelemente der beiden Sequenzen stets in Verbindung stehen. Es existieren also mit Sicherheit die Verbindungen (r_1, s_1) und (r_m, s_n) . Um dies zu begründen, wird die Verbindung (r_1, s_1) betrachtet (die Begründung für beide Endelemente der Sequenzen geht analog). Nach Punkt 1 muss mindestens ein Matching (r_1, s_k) existieren. Ist $k = 1$, so gilt Punkt 3 trivialerweise. Andernfalls kann k nur größer 1 sein. Jetzt gilt nach Punkt 1 wieder, dass aber auch eine Verbindung (r_l, s_1) existieren muss. Wählt man nun auch ein $l > 1$, so existieren gleichzeitig die Verbindungen $(r_1, s_{k>1})$ und $(r_{l>1}, s_1)$. Nach Punkt 2 kreuzen die beiden Verbindungen sich dann aber. Es würde also nur ein $l \leq 1$ in Frage kommen. Das einzige l was damit in Frage kommt, wäre $l = 1$. Auch damit existiert also wieder die Verbindung (r_1, s_1) .

Der DTW-Algorithmus arbeitet rekursiv. Dabei werden Teillösungen der Gesamtdistanz $D(R, S)$ betrachtet, die sich auf die Gesamtdistanz zwischen einem Präfix von R und einem Präfix von S beziehen. Diese Zwischenlösungen werden im Folgenden als $D_{i,j}(R, S)$ geschrieben und stellen die Gesamtdistanz zwischen den ersten i Elementen von R und den ersten j Elementen von S dar. Es gilt also $D_{i,j}(R, S) := D((r_1, \dots, r_i), (s_1, \dots, s_j))$. Die am Ende gesuchte Gesamtdistanz von allen Elementen von R und allen Elementen von S ist dann $D(R, S) := D_{m,n}(R, S)$. Mit dieser Schreibweise lässt sich der Algorithmus dann folgendermaßen rekursiv definieren:

$$D_{i,j}(R, S) = \begin{cases} 0 & i = 0 \wedge j = 0 \\ d(r_i, s_j) + \min \begin{cases} D_{i-1,j}(R, S) \\ D_{i,j-1}(R, S) \\ D_{i-1,j-1}(R, S) \end{cases} & 1 \leq i \leq m \wedge 1 \leq j \leq n \\ \infty & \text{sonst} \end{cases} \quad (3.7)$$

Der erste Fall ist der Basisfall, wo die Gesamtdistanz von zwei leeren Sequenzen berechnet werden soll. Da somit keine Verbindungen zwischen Elementen entstehen können und somit auch keine Distanzen berechnet werden können, wird definiert dass $D(\emptyset, \emptyset) = 0$. Beim dritten Fall der Fallunterscheidung handelt es sich auch um einen Basisfall. Hier ist $i = 0 \vee j = 0$ nicht aber $i = 0 \wedge j = 0$. Damit soll die Gesamtdistanz von einer nicht leeren Sequenz

und einer leeren Sequenz berechnet werden. Da jedoch jedes Element mindestens Teil einer Verbindung sein muss und es keine möglichen Partner für Elemente aus der nicht leeren Sequenz gibt (da die andere Sequenz leer ist), wird die Distanz hier als maximal bzw. ∞ definiert. Da der Algorithmus probiert zu minimieren, wird also niemals eine Zuordnung gewählt, die im Laufe des Algorithmus auf solch einen Fall trifft (da offensichtlich nicht alle Elemente einen Partner zugewiesen bekommen haben und das Matching somit nicht gültig ist). Im Rekursionsschritt werden dann die Elemente r_i und s_j verbunden und deren Distanz $d(r_i, s_j)$ berechnet und mit der minimalen Distanz aller Rekursionsfälle addiert. Dabei wird zwischen drei Fällen unterschieden. Da sowohl r_i als auch s_j nun mindestens einen Partner besitzen, lassen sich diese von den jeweiligen Sequenzen ausschließen und es kann mit dem entsprechenden Präfix der Sequenz weitergerechnet werden. Dabei können beide ausgeschlossen werden, womit im weiteren Verlauf keine Verbindung mehr mit r_i oder s_j hergestellt werden kann oder es wird genau einer von beiden ausgeschlossen, wobei der jeweils andere in der nächsten Rekursionstiefe wieder eine (weitere) Verbindung mit dem Vorgänger von dem ausgeschlossenen Element aufbaut. Es sei zu beachten, dass zu keiner Zeit im Algorithmus eine Verbindung mit einem Element aufgebaut werden kann, welches sich außerhalb der beiden Präfixe befindet. Damit ist implizit sichergestellt, dass ein Matching nie zwei Verbindungen enthält, welche sich kreuzen.

Da eine rekursive Implementation nach 3.7 gleiche Teilprobleme öfters berechnet, ergibt sich aufgrund der kombinatorischen Explosion des Algorithmus ein exponentielles Laufzeitverhalten. Der DTW-Algorithmus ist ein klassisches Problem der dynamischen Programmierung (DP) und erinnert in seiner Formulierung stark an andere ähnliche DP-Probleme, wie etwa die Berechnung der Editierdistanz (Levenshtein-Distanz) von zwei Sequenzen oder der Berechnung der größten gemeinsamen Teilsequenz von zwei Sequenzen (Longest Increasing Subsequence - LCS).

Stattdessen wird der DTW-Algorithmus in der Regel als iterativer Algorithmus formuliert, welcher die Teilprobleme in einem $m \times n$ Array speichert, um diese Zwischenergebnisse nicht erneut berechnen zu müssen (dies wird auch als Memoisation bezeichnet). Dabei wählt man einen Bottom-Up Ansatz: Es werden zunächst die Basisfälle in das Array eingetragen. Daraufhin lässt sich so über das Array iterieren, dass jeder Wert direkt berechnet werden kann (da die nötigen Teilprobleme bereits gelöst wurden und deren Ergebnisse abgespeichert wurden). Am Ende ist das komplette Array gefüllt und man kann die Gesamtdistanz zum Ausgangsproblem einfach im Array auslesen und zurückgeben.

Der Folgende Ausschnitt an Pseudocode (1) beschreibt diese Art des DTW-Algorithmus.

Algorithm 1 DTW-Algorithmus

```

1: function DTW( $R : [r_1 \dots r_m], S : [s_1 \dots s_n]$ )
2:    $D \leftarrow \text{array}[0 \dots m, 0 \dots n]$ 
3:    $D[0, 0] \leftarrow 0$ 
4:   for  $i = 1$  to  $m$  do
5:      $D[i, 0] \leftarrow \infty$ 
6:   end for
7:   for  $j = 1$  to  $n$  do
8:      $D[0, j] \leftarrow \infty$ 
9:   end for
10:  for  $i = 1$  to  $m$  do
11:    for  $j = 1$  to  $n$  do
12:       $D[i, j] \leftarrow d(r_i, s_j) + \min \begin{cases} D[i - 1, j] \\ D[i, j - 1] \\ D[i - 1, j - 1] \end{cases}$ 
13:    end for
14:  end for
15:  return  $D[m, n]$ 
16: end function

```

Die Laufzeitkomplexität von Algorithmus 1 ist nun nicht mehr exponentiell sondern polynomial. Die dafür ausschlaggebende Operation ist die Berechnung in Zeile 12. Diese Operation ist zwar konstant, wird aufgrund der Schleifen in Zeile 10 und Zeile 11 jedoch exakt $m \cdot n$ mal ausgeführt. Dementsprechend ist das Laufzeitverhalten vom DTW-Algorithmus $\Theta(m \cdot n)$. Angewandt auf den Anwendungsfall, die minimale Distanz zweier Kanten von zwei Puzzlestücken zu berechnen, ist dieser Algorithmus damit quadratisch in der Länge der Kante (hier mit K bezeichnet): $\Theta(K^2)$. Im Vergleich dazu war die Berechnung nach einer euklidischen Zuordnung (beschrieben anhand der DBC in 3.2) linear. Außerdem besitzt Algorithmus 1 auch eine Speicherkomplexität von $\Theta(m \cdot n)$, um die Zwischenergebnisse per Memoisation im Array zwischenspeichern zu können. Es existieren für solche Art von DP-Algorithmen einfache Erweiterungen, welche die Speicherkomplexität auf $\Theta(\min(m, n))$ verbessern können. Die Begründung dafür ist, dass bei genauerer Betrachtung des Algorithmus auffällt, dass zur Berechnung in Zeile 12 lediglich Zwischenergebnisse aus der aktuell betrachteten Zeile und der vorherigen Zeile notwendig sind. Zwei Zeilen als zusätzlicher Speicher sind also hinreichend zur Berechnung des Endergebnisses (und es lässt sich auch zeigen, dass dies sich noch weiter auf nur eine Zeile und eine zusätzliche Variable verbessern lässt, was jedoch an der Speicherkomplexität nichts ändert). Da der Speicheraufwand jedoch selbst in der Version, welche in Algorithmus 1 dargestellt wird, bereits vergleichsweise gering ist und die zusätzlichen Zwischenergebnisse in der Matrix durchaus hilfreiche Informationen über das eigentliche Matching liefern, gilt Algorithmus 1 an dieser Stelle als hinreichend gut. Als Begründung für die erste Aussage (dass der Speicheraufwand vergleichsweise gering ist), lässt sich sagen, dass zu jederzeit nur eine solche Matrix im Speicher liegen sollte, welche außerdem bei jeder Berechnung wiederverwendet werden kann. Im Vergleich dazu wurde bei der PBC in 3.3

für alle vier Seiten von jedem Puzzlestück eine zusätzliche Zeile bzw. Spalte an approximierten Pixeln zwischengespeichert.

Um zuletzt noch zu zeigen, dass die Zwischenergebnisse in der Matrix verwendet werden können, um das Matching zweier Sequenzen bestimmen zu können (und nicht nur die Distanz von diesem minimalen Matching), wird wieder das Beispiel 3.5 betrachtet. Ein euklidisches Matching mit der L_1 Norm ergab eine Distanz von 2246. Tabelle 3.2 zeigt die Matrix nach einem Durchlauf des Algorithmus mit den Pixel-Sequenzen R und S aus Abbildung 3.5. Wie an der untersten rechten Zelle zu erkennen ist, ist die minimale Distanz 1434. Da $1434 < 2246$ existiert also ein Matching was nicht dem euklidischen Matching entspricht, aber geringere Distanzen liefert.

Dazu lässt sich mit den Werten in der Matrix nun der Algorithmus zurückverfolgen (Backtracking). Dazu wird rechts unten beim Endergebnis gestartet. An dieser Stelle wurden die letzten beiden Pixel von R und S verbunden (deshalb ist die Zelle farblich hinterlegt). Außerdem wurde an dieser Stelle die Entscheidung getroffen, als Teillösung die Lösung zu nehmen, die den letzten Pixel von S ausschließt (da dies die Teillösung war mit der geringsten Gesamtdistanz als Zwischenergebnis). Es wird also in der Zelle links vom Endergebnis weiter gemacht. Dort lässt sich wieder ablesen, welche Pixel verbunden wurden und anhand der Werte in den benachbarten Zellen (direkt links, direkt oben und diagonal links-oben der aktuell betrachteten Zelle) lässt sich sagen, welche Entscheidung der Algorithmus an dieser Stelle getroffen hat. Dies lässt sich bis zum Basisfall zurückverfolgen, wo beide Präfixe der Sequenzen leer sind. Es sei gesagt, dass auch der erste Pixel von R mit dem ersten Pixel von S verbunden wird (es wurde bereits gezeigt, dass sowohl die Anfangs-, als auch die Endelemente beider Sequenzen stets verbunden werden).

$R \backslash S$	\emptyset	green	blue	brown	yellow	orange	dark red	purple	teal
\emptyset	0	∞							
green	∞	167	430	529	764	1174	1308	1494	1811
blue	∞	235	431	602	803	1267	1371	1507	1824
yellow	∞	525	339	689	1054	1170	1531	1496	1706
orange	∞	704	618	480	625	917	1169	1487	1728
dark red	∞	739	817	707	729	1063	1197	1411	1676
purple	∞	1086	898	990	1110	853	1273	1451	1622
teal	∞	1331	1169	1007	1303	1179	1071	1271	1596
	∞	1531	1365	1327	1429	1614	1386	1172	1434

Tabelle 3.2: Backtracking im DTW-Algorithmus

Markiert man sich nun den Weg (blau hinterlegt in 3.2), der beim Backtracking der Matrix entsteht, so lassen sich alle Verbindungen zwischen den einzelnen Pixeln ablesen und in Abbildung 3.5 eintragen. Das dabei entstandene Matching ist in Abbildung 3.7 dargestellt.

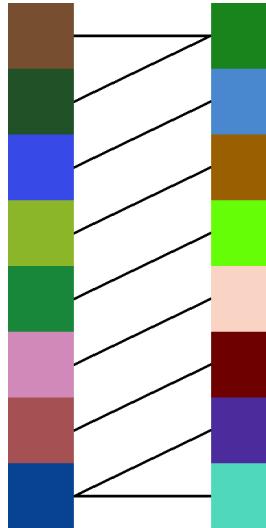


Abbildung 3.7: Zuordnung der Pixel beim DTW-Algorithmus

Wie zu sehen ist, ist dieses Matching auch gültig, da jedes Element mindestens einen Partner hat und sich außerdem keine Verbindungen überkreuzen.

3.5 MAHALANOBIS GRADIENT COMPATIBILITY (MGC)

Die Mahalanobis Gradient Compatibility (MGC) wurde zuerst in [16] beschrieben und später in weiteren Werken [28, 17, 20, 12, 10] verwendet oder je nach Problemstellung leicht angepasst.

Bei dieser Art der Kompatibilität wird zunächst von allen vier Seiten jedes Puzzlestücks der durchschnittliche Wert der Ableitungen (vgl. 3.3) berechnet (für allen drei Farb-Channels). Mit diesen Ableitungen und dem Durchschnitt lässt sich dann die Kovarianzmatrix bezüglich diesen Ableitungen berechnen. Beim Vergleich zweier Puzzlestücke werden dann wieder beide Seiten betrachtet. Es wird zunächst die Mahalanobis-Distanz von allen Ableitungen zwischen den Puzzlestücken bezüglich der Verteilung von einem Puzzlestück berechnet und aufsummiert. Um die Symmetrie beizubehalten wird dies auch noch von der anderen Seite gemacht. Die Mahalanobis-Distanz berechnet dabei die (euklidische) Distanz eines Ableitungsvektors (mit drei Komponenten für die entsprechenden Channels) bezüglich eines Koordinatensystems welches durch den Durchschnittsvektor und der Kovarianzmatrix (des jeweils anderen Puzzlestücks) beschrieben wird. Der Durchschnittsvektor bildet dabei den Ursprung des Koordinatensystems und die (normierten) Eigenvektoren der Kovarianzmatrix bilden die Achsen. Die Mahalanobis-Distanz führt zunächst diese Koordinatentransformation durch, um danach die Distanz eines Vektors in diesem Koordinatensystem berechnen zu können.

Es werden im Folgenden wieder zwei Puzzlestücke x_i und x_j betrachtet, wobei $r = 0$ und x_i damit links von x_j platziert wird. Die Distanz $D(x_i, x_j, r)$ wird wieder in zwei Teile aufgeteilt (ähnlich wie bei der PBC in 3.3): $D(x_i, x_j, r) = D_L(x_i, x_j, r) + D_R(x_i, x_j, r)$. Damit ist die Symmetrie der Distanz garantiert und es gilt $D(x_i, x_j, r) = D(x_i, x_j, r')$. Es wird im Folgenden dann nur die Berechnung für $D_L(x_i, x_j, 0)$ aufgeführt. Für diese

Berechnung ist es zunächst nötig, die Ableitungen von x_i an der rechten Seite zu berechnen. Wie in 3.3 wird dafür der Backward-Differenzenquotient entlang der x-Achse genommen. In der Literatur werden diese Ableitungen als Gradient G bezeichnet. Für die rechte Seite von x_i ergeben sich die Ableitungsvektoren dann folgendermaßen (es ist zu beachten, dass ein G drei Komponenten hat):

$$G_k = x_i(w - 1, k) - x_i(w - 2, k), \forall 0 \leq k < h \quad (3.8)$$

Damit lässt sich dann der durchschnittliche Ableitungsvektor bestimmen:

$$\mu = \frac{1}{h} \sum_{k=0}^{h-1} G_k \quad (3.9)$$

Mit allen Ableitungsvektoren und dem durchschnittlichen Ableitungsvektor lässt sich dann die Kovarianzmatrix S bestimmen. Dies ist hier eine 3×3 Matrix und lässt sich folgendermaßen bestimmen:

$$S = \frac{1}{h-1} \sum_{k=0}^{h-1} (G_k - \mu)(G_k - \mu)^T \quad (3.10)$$

Um numerische Probleme bei der späteren Invertierung von S zu vermeiden, lassen sich einige *dummy* Gradienten mit in die Berechnung der Matrix S einbeziehen. Dabei wurde sich an [16, 10] orientiert und es wurden neun solcher Gradienten gewählt. Diese Gradienten lauten

$$\begin{aligned} &(0, 0, 0), (1, 1, 1), (-1, -1, -1), \\ &(0, 0, 1), (0, 1, 0), (1, 0, 0), \\ &(-1, 0, 0), (0, -1, 0), (0, 0, -1) \end{aligned}$$

Die Mahalanobis-Distanz von einem Vektor x berechnet sich dann zu:

$$D_M(x) = \sqrt{(x - \mu)^T S^{-1} (x - \mu)} \quad (3.11)$$

Es ist nun möglich mit 3.9, 3.10 und 3.11 die einseitige Gesamtdistanz $D_L(x_i, x_j, 0)$ zu definieren. Dabei ist lediglich zu beachten, dass nach [16] die Wurzeloperation aus 3.11 weggelassen wird.

$$D_L(x_i, x_j, 0) = \sum_{k=0}^{h-1} (G_k^{i \rightarrow j} - \mu)^T S^{-1} (G_k^{i \rightarrow j} - \mu) \quad (3.12)$$

Dabei ist $G_k^{i \rightarrow j}$ der Ableitungsvektor der beim Übergang von einem Puzzlestück zum anderen entsteht, wenn man die beiden Puzzlestücke x_i und x_j nun (mit x_i links von x_j) zusammenfügt. Dafür gilt:

$$G_k^{i \rightarrow j} = x_j(0, k) - x_i(w - 1, k), \forall 0 \leq k < h \quad (3.13)$$

Bezüglich der Implementation lässt sich sagen, dass es durchaus möglich ist (ähnlich wie in 3.3), die Ableitungsvektoren, den durchschnittlichen Ableitungsvektor und auch die Inverse der Kovarianzmatrix (es wird zu jederzeit

nur die Inverse von S benötigt, daher lohnt es sich nicht auch S selber abzuspeichern) jeder Seite (und für jedes Puzzlestück) nur einmal zu berechnen und diese dann abzuspeichern. Bei dem eigentlichen Vergleich zweier Puzzlestücke müssen dann nur 3.12 und 3.13 berechnet werden. Trotz der großen Anzahl und höheren Komplexität der Berechnungen, ist das Laufzeitverhalten der kompletten Prozedur linear in der Länge der Kante (beispielsweise h wie in den obigen Gleichungen). Berechnungen wie 3.10 oder 3.12 haben aufgrund der Matrizenmultiplikationen zwar größeren konstanten Aufwand (in der Summe), da jede Matrix aber die Dimensionen 1×3 , 3×1 oder 3×3 besitzt, sind die Dimensionen der Matrizen und somit auch alle Operationen mit diesen Matrizen konstant.

Diese Linearität dieses Verfahrens bietet einen großen Vorteil bei der Berechnung von sehr hochauflösenden Bildern (bzw. hochauflösenden Puzzlestücken). Verfahren wie etwa DTW (3.4) sind dort aufgrund der quadratischen Laufzeit eher weniger geeignet. Diese Tatsache und ein komplexeres Verfahren zur Bewertung der Übergänge zwischen zwei Puzzlestücken (im Vergleich zu der trivialeren PBC in 3.3), hat dazu geführt, dass selbst komplexere Jigsaw-Puzzle-Solver (wie etwa [17]) noch diese Metrik verwenden. Aus diesen Gründen wurde diese Metrik auch für den folgenden Algorithmus verwendet.

4

ALGORITHMUS ZUR PUZZLE-ASSEMBLIERUNG

Ausgestattet mit einer Vielzahl von Metriken aus 3 gilt es nun einen Algorithmus zu entwickeln, der die eigentliche Assemblierung des Puzzles übernimmt. Dabei wird aktuell noch von einem Jigsaw-Puzzle ausgegangen bei dem keine Puzzlestücke fehlen (bei Schiebepuzzles fehlt entsprechend ein Stück, auf diesen Sonderfall wird dort eingegangen, wo dieser Fall Probleme beim Algorithmus bereitet). Da das Jigsaw-Puzzle nachweisbar als NP-Hard gilt [15], beruhen alle bekannten algorithmischen Puzzle-Löser auf einen heuristischen Ansatz, um große Puzzle lösen zu können.

Aktuelle Methoden zum Lösen von Jigsaw-Puzzles lassen sich grob in zwei Kategorien aufteilen:

1. Greedy Methoden [20, 13, 16], welche probieren aus den paarweisen Distanzen der einzelnen Puzzlestücke iterativ ein Gesamtbild zu assemblieren, indem in jedem Schritt die günstigsten bzw. besten Puzzlestücke miteinander verbunden werden.
2. Globale Methoden, welche probieren eine globale Kompatibilitätsfunktion zu maximieren (bzw. eine globale Distanzfunktion zu minimieren) um dabei ein globales Maximum (oder Minimum im Falle einer Distanzfunktion) zu finden. Darunter fallen unter anderem Ansätze aus der genetischen Programmierung [14], linearen Programmierung [28] und auch probabilistische Verfahren [29].

In dieser Arbeit wird sich auf das Spannbaum Verfahren von Gallagher [16] fokussiert. Dieses Verfahren lässt sich in die Kategorie der Greedy Methoden einordnen und basiert hauptsächlich auf der Observierung, dass gängige Algorithmen zur Berechnung eines minimalen Spannbaums (Minimum Spanning Tree - MST) mit wenigen Änderungen bereits in der Lage dazu sind, simple Jigsaw-Puzzle zu assemblieren. Diese Idee wurde dann so perfektioniert, dass Gallagher selber in der Lage dazu war, Puzzle mit mehreren tausend Puzzlestücken perfekt zu rekonstruieren. Neben dieser vielversprechenden Ergebnisse und der Tatsache, dass dieses Verfahren auch das erste Verfahren war, welches die MGC (vgl. 3.5) verwendet hat (auch diese wurde von Gallagher selber in [16] beschrieben), basiert dieses Verfahren trotz State-of-the-Art Ergebnisse hauptsächlich auf Methoden der Graphentheorie.

Andere Verfahren mit vergleichbaren Ergebnissen (etwa das probabilistische Verfahren von Cho et al. [29]) basieren auf tiefliegendere und komplexere stochastische Zusammenhänge.

Neben dem Algorithmus selber und einige Änderungen um diesen auch für Schiebepuzzle möglich zu machen, wird außerdem eine effiziente Datenstruktur beschrieben, auf die der Algorithmus arbeiten wird. Dabei handelt es sich um eine modifizierte Version der Disjoint-Set-Datenstruktur oder auch Disjoint-Set-Forest.[9] Bei den Anforderungen dieser Datenstruktur wird sich sowohl an Gallagher selber gerichtet [16], als auch an C. Zanoci und J. Andress Beschreibung von Gallaghers Algorithmus.[10]

4.1 BAUM-BASIERTE ASSEMBLIERUNG NACH GALLAGHER

Es wird das Jigsaw-Puzzle Problem als Graph $G = (V, E)$ betrachtet mit der Menge der Puzzlestücke als Vertices V und die Menge der möglichen Verbindungen von zwei Puzzlestücken als Kanten E . Diese Kanten sind gewichtet und erhalten ihre Kantengewichte nach einer der in Kapitel 3 vorgestellten Metriken (vorzugsweise die MGC). Außerdem sei zu beachten, dass zwei Puzzlestücke nicht nur durch eine Kante sondern durch exakt vier Kanten in Verbindung stehen - jeweils eine für jede mögliche räumliche Anordnung der beiden Puzzleteile (vgl. Tabelle 3.1).

Da das fertige Puzzle eine Zusammenhangskomponente des Graphen darstellt (und zwar eine mit minimaler oder nahezu minimaler Kantengewichtssumme), bietet sich ein Algorithmus zur Berechnung des minimalen Spannbaums an. Ein solcher Algorithmus wäre in der Lage eine solche minimale Zusammenhangskomponente zu finden. Der minimale Spannbaum der durch einen solchen Algorithmus entstehen würde stellt in aller Regel jedoch keine legale Puzzlestellung dar. Dazu muss zusätzlich noch sichergestellt werden, dass jederzeit keine zwei Puzzlestücke in der Zusammenhangskomponente übereinander liegen. Diese zusätzliche Einschränkung des Algorithmus ist eine weitere Motivation für die Implementation einer modifizierten Disjoint-Set-Datenstruktur, die den räumlichen Zusammenhang der Zusammenhangskomponenten kennt und vermeidet, dass Zusammenhangskomponenten zusammengeführt werden, welche zu übereinander liegenden Puzzlestücken führen würden.

Entsprechend solch einer Datenstruktur wird damit im Zusammenhang der Algorithmus von Kruskal [22] betrachtet, um den minimalen Spannbaum in einem Graphen zu finden. Dabei werden am Anfang des Algorithmus alle Vertices (die Puzzlestücke) zunächst als separate Zusammenhangskomponenten aufgefasst. Damit bildet jeder Vertex einen Baum, die Menge aller Bäume wird als Wald (Forest) bezeichnet. Daraufhin werden die Kanten des Graphen in absteigender Reihenfolge (bezüglich des Kantengewichts) betrachtet. Verbindet eine Kante zwei Vertices, welche sich bereits in der selben Zusammenhangskomponente befinden, so wird diese Kante nicht zum Baum hinzugefügt, da diese dort sonst eine Schleife erzeugen würde. Im Falle wo diese Kante zwei Vertices verbinden würde, welche sich in verschiedenen Zusammenhangskomponenten befinden, so wird diese Kante zum Baum hinzugefügt. Dieser Schritt verbindet die beiden Zusammenhangskomponen-

ten zu einer größeren Zusammenhangskomponente. Dieser Vorgang wird solange wiederholt, wie noch Kanten vorhanden sind und es sich noch kein Spannbaum gebildet hat (es existieren noch mindestens zwei separate Bäume im Wald).

Die normale Disjoint-Set-Datenstruktur (Disjoint-Set-Forest) [9] übernimmt dabei die Aufgabe, die Menge der Vertices V in disjunkte zusammenhängende Mengen (Zusammenhangskomponenten) zu partitionieren. Die üblichen Operationen auf eine solche Datenstruktur sind unter anderem das Initialisieren eines solchen disjunktten Waldes mit n Elementen (was in $\Theta(n)$ Zeit möglich ist), das Testen, ob zwei Elemente sich in der selben Menge befinden und das Vereinigen zweier Mengen.

Die Laufzeit der letzten beiden Operationen hängt stark von der Implementation der Datenstruktur ab. Bei einer guten Implementation sind beide Operationen durchschnittlich $\Theta(\alpha(n))$.[18, 23, 30, 31, 26] Wobei $\alpha(n)$ für die inverse Funktion der eindimensionalen Ackermann-Funktion $f(n) = A(n, n)$ steht. Die Funktion $f(n)$ nimmt bereits für kleine n sehr große Werte an. So ist bereits $f(4) = A(4, 4) = 2 \uparrow\uparrow 7 - 3$ in Knuth's *uparrow* Notation.

Als Potenz ergibt sich der *Power-Tower* $f(4) = 2^{2^{2^{2^2}}} - 3$. Die einfachste Potenz die dies noch darstellen kann ist $f(4) = 2^{2^{2^{65536}}} - 3$. Selbst die innerste Potenz 2^{65536} ist bereits unvorstellbar groß. Dementsprechend lässt sich sagen, dass die inverse Ackermann-Funktion selbst für sehr große n stets Werte annehmen wird, die 4 nicht überschreiten. Damit lässt sich sagen, dass Operationen mit einem Laufzeitverhalten von $\Theta(\alpha(n))$ *quasi konstant* sind.

Ein Disjoint-Set-Forest (DSF) wird in der Regel als kompaktes Array implementiert. Jedes Element im Array ist dann ein Element im Forest (also ein Element eines Baums im Wald). Die Array-Indizes können dann dazu dienen die Elemente zu identifizieren. Jedes Element enthält dann zusätzlich noch einen Parent-Pointer (bei Arrays bietet sich auch ein Parent-Index an). Um zwei Mengen später effizient vereinigen zu können, lässt sich auch noch ein zusätzliches Feld definieren, welches die Anzahl der Elemente mitspeichert, die sich in demselben Baum wie das Element befinden. Dieses Feld ist nur für Elemente von Bedeutung, die die Wurzel ihres Baumes darstellen (da nur dort die Anzahl erhöht wird). Die Beziehungen der Elemente über den Parent-Index bilden einen *rückverweisenden* Baum: Es lässt sich bei jedem beliebigen Element starten und solange der Parent-Index verfolgen, bis dieser entweder ungültig (`null`) ist oder (bei Arrays üblich) ein Element der Parent von sich selber ist. Dieses Element ist die Wurzel des Baums, der alle Elemente enthält, die dieses Element auch als Wurzel haben. Dieses Wurzelement *repräsentiert* seine Menge.

Die `InitForest` Operation initialisiert einen DSF mit einer vorgegeben Kapazität n . Jedes Element befindet sich anfangs in seiner eigenen Menge.

Algorithm 2 Initialisierung eines Disjoint-Set-Forest

```

procedure INITFOREST( $n$ )
    nodes  $\leftarrow$  array[ $n$ ]
    for  $i = 1$  to  $n$  do
        nodes[ $i$ ].parent  $\leftarrow i$             $\triangleright$  Element ist Wurzel seiner Menge
        nodes[ $i$ ].size  $\leftarrow 1$             $\triangleright$  Menge enthält nur dieses Element
    end for
end procedure

```

Die **FindRoot** Operation übernimmt den Index i eines Elementes x_i und gibt den Index des Wurzelementes der Menge zurück in der sich x_i befindet. Diese Operation arbeitet sich solange den Baum hoch bis ein Element gefunden wurde, welches als Parent-Index sich selbst eingetragen hat. Der Index dieses Elementes wird dann zurückgegeben. An dieser Stelle lässt sich noch eine wichtige Verbesserung vornehmen, welche notwendig (jedoch noch nicht hinreichend) dafür ist, dass durch die inverse Ackermann-Funktion beschränkte Laufzeitverhalten zu garantieren. Diese Erweiterung wird als Pfadkomprimierung bezeichnet und soll dafür sorgen, dass alle Elemente die auf dem Weg der **FindRoot** Operation betrachtet werden, als direkte Kinder des Wurzelementes eingetragen werden. Damit wird ein gegebenenfalls sehr langer Pfad im Baum durch viele kleine Pfade (direkte Verbindungen zwischen den Elementen des Pfades und des Wurzelementes) ersetzt. Dies ist eine valide Operation, da die Zugehörigkeit zu einer Menge lediglich von dem Wurzelement des Baums bestimmt wird und in keiner Weise von den Beziehungen abhängt, die die Elemente untereinander haben. Dadurch das Bäume so stets *flach* gehalten werden, wird das Laufzeitverhalten dieser und anderer Operationen die den Baum nach oben traversieren deutlich verbessert. Die **FindRoot** Operation mit Pfadkomprimierung wird dann in der Regel folgendermaßen rekursiv definiert:

Algorithm 3 FindRoot Operation mit Pfadkomprimierung

```

function FINDROOT( $i$ )
    if nodes[ $i$ ].parent  $\neq i$  then
        nodes[ $i$ ].parent  $\leftarrow$  FINDROOT(nodes[ $i$ ].parent)
    end if
    return nodes[ $i$ ].parent
end function

```

Die **Union** Operation übernimmt zwei Indizes i, j , welche für die Elemente x_i, x_j stehen und vereinigt die Menge in der sich x_i befindet mit der Menge in der sich x_j befindet. Falls x_i und x_j sich bereits in der selben Menge befinden, muss die Operationen keine weiteren Änderungen an der Datenstruktur vornehmen. Ein Wahrheitswert als Rückgabewert der Funktion informiert darüber, ob die Mengen vereinigt wurden (**true**) oder nicht (**false**). Der Algorithmus bestimmt dafür zunächst die beiden Indizes k, l der Wurzellemente r_i, r_j indem die Operation **FindRoot** entsprechend mit i und j aufgerufen wird. Sind diese Indizes gleich so wird **false** zurückgegeben. Andernfalls wird eines der beiden Wurzelemente (hier beispielsweise r_j)

als das neue Wurzelement für die vereinigte Menge festgelegt. Dafür wird der Parent von r_i auf den Index von r_j gesetzt. Ein kompletter Baum wird also direkt an der Wurzel des anderen Baumes angehängt. Es entsteht ein neuer großer Baum. Selbst mit Pfadkomprimierung in der `FindRoot` Operation kann das Vereinigen zweier Mengen dazu führen, dass die Bäume sehr schnell in die Tiefe wachsen (wodurch die `FindRoot` Operation und damit auch die `Union` Operation lange Pfade traversieren muss). Als weitere notwendige Erweiterung für die Effizienz der Datenstruktur (entsprechend der inversen Ackermann-Funktion) muss versichert werden, dass stets der kleinere der beiden Bäume an den größeren angehängt wird. Dabei bezieht sich die Größe der Bäume auf die Anzahl der Elemente in der Menge. Diese Erweiterung wird als *Union by Size* bezeichnet. Zusammen mit der Pfadkomprimierung in `FindRoot` ist damit das durchschnittliche Laufzeitverhalten von $\Theta(\alpha(n))$ garantiert.

Algorithm 4 Union Operation als Union by Size

```

function UNION( $i, j$ )
     $k \leftarrow \text{FindRoot}(i)$ 
     $l \leftarrow \text{FindRoot}(j)$ 
    if  $k = l$  then
        return false       $\triangleright x_i$  und  $x_j$  befinden sich in der selben Menge
    end if
    if nodes[ $k$ ].size > nodes[ $l$ ].size then
        Swap( $k, l$ )           $\triangleright k$  ist der Wurzelindex der kleineren Menge
    end if
    nodes[ $k$ ].parent  $\leftarrow l$        $\triangleright$  Kleinere Menge an die größere anhängen
    nodes[ $l$ ].size  $\leftarrow$  nodes[ $l$ ].size + nodes[ $k$ ].size   $\triangleright$  Anzahl aktualisieren
    return true
end function

```

Um die normale DSF-Datenstruktur für das rekonstruieren von Puzzles anzupassen, müssen zunächst weitere Attribute zu der Datenstruktur hinzugefügt werden, welche die räumliche Beziehung der Mengenelemente (welche jetzt konkret Puzzlestücke sind) beschreiben.

Neben einem Parent-Index, sollte jedes Element zusätzlich noch seine relative Position zu seinem Parent in Form von zweidimensionalen Koordinaten abspeichern. Dabei zeigt die positive x-Achse wieder nach rechts und die positive y-Achse nach unten. Die Koordinaten eines Elementes v bezüglich seines Parent p wird dann durch den Vektor von p nach v ausgedrückt: \vec{pv} . Alle Elemente einer Menge befinden sich in einem gemeinsamen Koordinatensystem, wobei das Wurzelement den Ursprung dieses Koordinatensystems bildet.

Auch das Attribut welches die Anzahl der Elemente in der Menge des Elementes angibt (welches lediglich dann von Nutzen ist, wenn das Element ein Wurzelement ist) lässt sich ersetzen (dabei muss aufgepasst werden, dass diese Information nicht komplett verloren geht, da die Anzahl für Algorithmus 4 benötigt wird). Stattdessen bekommt jedes Element (auch wenn das Attribut wieder nur für Wurzelemente gedacht ist) eine Menge von geordneten Paaren. Diese Paare bestehen aus einem Elementindex und aus einer Koordinate. Ein solches Paar repräsentiert ein Element v im Baum des

Wurzelementes r und die Koordinate ist die *absolute* Koordinate dieses Elementes im Koordinatensystem der Menge (also \vec{rv}). Jedes Wurzelement kennt also alle Elemente seiner Menge und weiß auch wo diese Elemente sich relativ zu ihm befinden. Die Kardinalität der Menge ist dann die Anzahl der Elemente in diesem Baum, womit klar wird wieso das Attribut `size` entfernt wurde: Diese Information ist nun implizit durch die Menge aller Elemente gegeben.

Als letztes Attribut soll sich die Datenstruktur auch merken, wie viele aktive Bäume sich im Wald befinden. Anfangs sind dies n Bäume für n Elemente (da jedes Element in seiner eigenen Menge ist), im Hinblick auf den Algorithmus von Kruskal ist es aber besonders interessant, wann diese Anzahl 1 ist (und somit eine Zusammenhangskomponente entstanden ist).

Damit lassen sich nun Algorithmen 2, 3, 4 so anpassen, dass die Datenstruktur für die Rekonstruktion von Puzzles geeignet ist (und somit die Aufgabe übernimmt, nur gültige Zusammensetzungen von Puzzlestücken als Zusammenhangskomponenten zuzulassen).

Algorithm 5 Erweiterte InitForest Operation

```

1: procedure INITFOREST( $n$ )
2:   nodes  $\leftarrow$  array[ $n$ ]
3:   tree_count  $\leftarrow n$             $\triangleright$  Anzahl der aktiven Bäume ist anfangs  $n$ 
4:   for  $i = 1$  to  $n$  do
5:     nodes[ $i$ ].parent  $\leftarrow i$        $\triangleright$  Element ist Wurzel seiner Menge
6:     nodes[ $i$ ].parent_coords  $\leftarrow (0, 0)$ 
7:     nodes[ $i$ ].tree  $\leftarrow$  set{( $index : i, root\_coords : (0, 0)$ )}
8:   end for
9: end procedure

```

Zeile 6 in Algorithmus 5 definiert für ein Element v die relativen Koordinaten zu seinem Parent als die Koordinaten des Vektors \vec{pv} . Da alle Elemente aber anfangs sich selber als Parent haben gilt $\vec{pv} = \vec{vv} = \vec{0}$. Zeile 7 erstellt die Menge, die alle Elemente mit Index und Koordinate (bezüglich des Wurzelement der Menge) enthält. Anfangs ist jedes Element in seiner eigenen Menge und das einzige Paar was bei der Initialisierung eingetragen werden muss ist das mit Paar mit dem Index des eigenen Elementes und den Koordinaten $(0, 0)$. Da jedes Element anfangs die Wurzel seiner Menge darstellt, befindet sich dieses Element auch im Ursprung des Koordinatensystems (welches ja seinen Ursprung beim Wurzelement der Menge hat). Anders gesagt ist anfangs $r = v$ und damit $\vec{rv} = \vec{vv} = \vec{0}$.

Algorithm 6 Erweiterte FindRoot Operation

```

1: function FINDROOT( $i$ )
2:   if  $\text{nodes}[i].\text{parent} \neq i$  then
3:      $k \leftarrow \text{FindRoot}(\text{nodes}[i].\text{parent})$ 
4:      $\vec{rp} \leftarrow \text{nodes}[\text{nodes}[i].\text{parent}].\text{parent\_coords}$ 
5:      $\text{nodes}[i].\text{parent\_coords} \leftarrow \vec{rp} + \text{nodes}[i].\text{parent\_coords}$ 
6:      $\text{nodes}[i].\text{parent} = k$ 
7:   end if
8:   return  $\text{nodes}[i].\text{parent}$ 
9: end function

```

Algorithmus 6 übernimmt neben der Rückgabe des Wurzelindex weiterhin eine Pfadkomprimierung. Da die Koordinaten eines Elementes v bezüglich seines Parent p bei der Pfadkomprimierung ihre Gültigkeit verlieren, weil das Element durch die Komprimierung einen neuen Parent (das Wurzelement) bekommt, muss die Koordinate von \vec{pv} auf \vec{rv} geändert werden. Dazu muss jedoch zunächst \vec{rp} bekannt sein. Diesen Vektor bekommt man durch einen rekursiven Aufruf von **FindRoot** mit dem Parent als Parameter. Dann wird dort die Pfadkomprimierung angestoßen und p bekommt die Wurzel als Parent. Dies geht bis zum Basisfall der Rekursion weiter. Beim Abbau der Rekursion werden dann alle Vektoren aufaddiert. Für einen Pfad von v nach r mit m Zwischenelementen $q_1 \dots q_m$ baut sich durch die Rekursion folgende Summe auf:

$$\underbrace{\vec{0} + \vec{rq_m} + \vec{qm}q_{m-1} + \dots + \vec{q_2q_1}}_{\vec{rq_1}} + \vec{q_1v} = \vec{rv}$$

Damit lässt sich erklären, was genau in Algorithmus 6 passiert (und warum die Reihenfolge der einzelnen Statements genau so gewählt ist). Beim Aufruf von **FindRoot** mit einem Index der für Element v steht, wird in Zeile 3 der Index des Wurzelementes zwischengespeichert (k). Viel wichtiger noch ist, dass nach Abschluss dieser Zeile der rekursive Schritt fertig ist und für das Parent-Element p nun im Koordinaten-Feld der Vektor \vec{rp} steht. Da der Index des Parents noch bekannt ist (deswegen wurde der Index des Wurzelementes zunächst nur zwischengespeichert), lässt sich dieser Vektor einfach nach Zeile 4 auslesen. Die Addition in Zeile 5 berechnet dann $\vec{rp} + \vec{pv} = \vec{rv}$. Jetzt lässt sich ohne Probleme die Pfadkomprimierung abschließen, indem man das zuvor gefundene Wurzelement als Parent von v erklärt (Zeile 6).

Als simples Beispiel ist in Abbildung 4.1 ein Koordinatensystem dargestellt. Es wird ein Element v betrachtet mit einem Parent p , Grandparent g und Root r . Dabei ist r der Ursprung des Koordinatensystems. Die Vektoren $\vec{pv}, \vec{gp}, \vec{rg}$ beschreiben die relativen Position von einem Element bezüglich seines Parents. Ziel ist es $\vec{rv} = \vec{rg} + \vec{gp} + \vec{pv}$ zu berechnen. Die Rekursion bewirkt, dass zunächst $\vec{rp} = \vec{rg} + \vec{gp}$ berechnet wird und dann mit dem Ergebnis das gesuchte $\vec{rv} = \vec{rp} + \vec{pv}$.

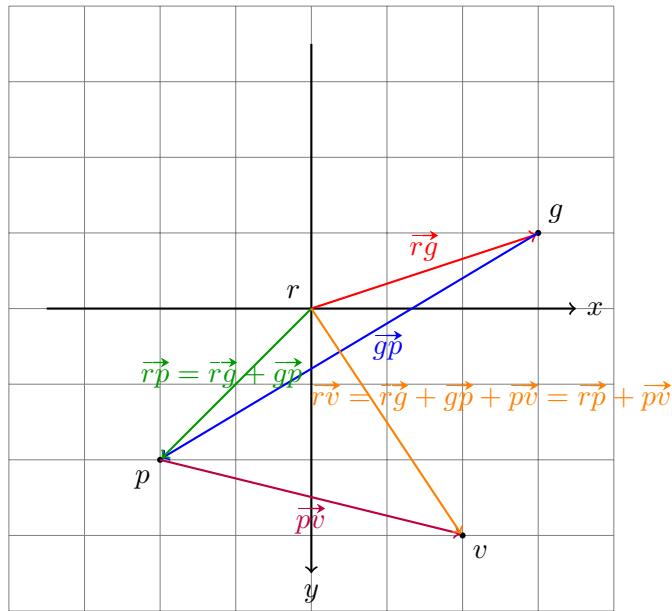


Abbildung 4.1: Koordinatentransformation im DSF bei Pfadkomprimierung

Bei der Union Operation gilt es nun die neuen implementierten Informationen (in Form von räumlichen Beziehungen der Elemente) so zu verwenden, dass nur dann vereinigt wird, wenn dabei keine zwei Puzzlestücke übereinander liegen. Dazu wird optimistischer Weise eine Vereinigung der beiden Mengen berechnet, welche die Paare von Elementindizes inklusive Koordinaten enthalten. Da diese Koordinaten sich jedoch auf zwei verschiedene Koordinatensysteme beziehen, muss zunächst in einer dieser Mengen eine Transformationen dieser Koordinaten vorgenommen werden. Diese Vereinigung wird dann nur bezüglich der Koordinate eines solchen Paares gebildet. Mit der Kardinalität dieser Vereinigungsmenge lässt sich dann feststellen, ob die beiden Bäume vereinigt werden können oder nicht. Da nun das Vereinigen zweier Bäume konkret das Zusammenstecken zweier Puzzlestücke darstellt ist es bei dieser Operation nun auch notwendig zu wissen, wie diese beiden Stücke zusammengesetzt werden. Dazu wird eine numerische Konstante *r* definiert nach Tabelle 3.1 mit übergeben.

Algorithm 7 Erweiterte Union Operation

```

1: function UNION( $i, j, r$ )
2:    $k \leftarrow \text{FindRoot}(i)$ 
3:    $l \leftarrow \text{FindRoot}(j)$ 
4:   if  $k = l$  then
5:     return false       $\triangleright v_i$  und  $v_j$  befinden sich in der selben Menge
6:   end if
7:   if nodes[ $k$ ].tree.size > nodes[ $l$ ].tree.size then
8:     Swap( $k, l$ )       $\triangleright k$  ist der Wurzelindex der kleineren Menge
9:     Swap( $i, j$ )       $\triangleright i$  ist Index vom Element der kleineren Menge
10:     $r \leftarrow (r + 2) \bmod 4$   $\triangleright$  Inverse Beziehung von  $v_i, v_j$  berechnen
11:   end if
12:   tree'_ $i \leftarrow \text{set } \{\}$ 
13:   for all  $(u, \overrightarrow{r_i v_u}) \in \text{nodes}[k].\text{tree}$  do
14:     tree'_ $i \leftarrow \text{tree}'_i \cup \{(u, \overrightarrow{r_j r_i} + \overrightarrow{r_i v_u})\}$ 
15:   end for
16:   if tree'_ $i \cap \text{nodes}[l].\text{tree} \neq \emptyset$  then
17:     return false           $\triangleright$  Stücke überschneiden sich
18:   end if
19:   nodes[ $l$ ].tree  $\leftarrow \text{nodes}[l].\text{tree} \cup \text{tree}'_i$ 
20:   nodes[ $k$ ].parent  $\leftarrow l$        $\triangleright$  Kleinere Menge an die größere anhängen
21:   nodes[ $k$ ].parent_coords  $\leftarrow \overrightarrow{r_j r_i}$ 
22:   nodes[ $k$ ].tree  $\leftarrow \text{set } \{\}$ 
23:   tree_count  $\leftarrow \text{tree\_count} - 1$ 
24:   return true
25: end function

```

Der interessante Teil des Algorithmus läuft zwischen Zeile 12 und 19 ab. Kommt der Algorithmus erfolgreich bis Zeile 12, so befinden sich die beiden Elemente v_i, v_j nicht in der selben Menge. Außerdem ist an dieser Stelle sichergestellt, dass v_i stets das Element der kleineren Menge ist (die Wurzel r_i des Baumes dieses Elementes soll an die Wurzel r_j des größeren Baumes des anderen Elementes angehängt werden). Beide Wurzeln r_i, r_j kennen alle Elemente ihrer Menge mit den Koordinaten in dem Koordinatensystem der Menge. Beim Vereinigen bekommt r_j nun alle Elemente von r_i . Dazu müssen aber die Koordinaten aller Elemente zunächst angepasst werden. Für ein Element v_u in der Menge von r_i ist die Koordinate dieses Elementes in der Form $\overrightarrow{r_i v_u}$ angegeben. Um v_u in die Menge aller Elemente aus r_j übernehmen zu können muss diese Koordinate in die Form $\overrightarrow{r_j v_u}$ gebracht werden. Dazu muss berechnet werden:

$$\overrightarrow{r_j v_u} = \overrightarrow{r_j r_i} + \overrightarrow{r_i v_u} \quad (4.1)$$

Es ist also der Vektor $\overrightarrow{r_j r_i}$ gesucht (die relative Position von der Wurzel r_i zu r_j). Diese standen zuvor in keinerlei Beziehung, da jede Menge ihr eigenes Koordinatensystem besitzt. Bei der Zusammensetzung von v_i und v_j lässt sich jedoch eine Beziehung der beiden Wurzeln beschreiben. Zunächst sei gesagt, dass der Parameter r der Methode (die räumliche Beziehung der beiden Elemente) exakt den Vektor $\overrightarrow{v_j v_i}$ bestimmt. Dieser lässt sich direkt der

Tabelle 3.1 entnehmen. Beispielsweise ist $\overrightarrow{v_j v_i} = (-1, 0)$ genau dann wenn $r = 0$.

Da beide Wurzeln alle Koordinaten ihrer Elemente kennen (und somit auch die Koordinaten der entsprechenden v 's), könnte man diese Menge durchsuchen. Dies dauert aber zu lange und ist dank der neuen erweiterten Pfadkomprimierung aus Algorithmus 6 nicht nötig. In Zeile 2 und 3 von Algorithmus 7 werden die Indizes von r_i, r_j bestimmt dabei werden v_i, v_j aber direkte Kinder ihrer entsprechenden Wurzeln. Dank der Koordinatentransformation nach Abbildung 4.1 stehen die Koordinaten $\overrightarrow{r_i v_i}$ und $\overrightarrow{r_j v_j}$ direkt als Attribut in den jeweiligen Elementen drin. Damit sind alle Informationen zur Berechnung von $\overrightarrow{r_j r_i}$ vollständig und es gilt:

$$\begin{aligned}\overrightarrow{r_j r_i} &= \overrightarrow{r_j v_j} + \overrightarrow{v_j v_i} + \underbrace{\overrightarrow{v_i r_i}}_{=-\overrightarrow{r_i v_i}} \\ \overrightarrow{r_j r_i} &= \overrightarrow{r_j v_j} + \overrightarrow{v_j v_i} - \overrightarrow{r_i v_i}\end{aligned}\tag{4.2}$$

Somit lässt sich $\overrightarrow{r_j r_i}$ noch vor den Eintritt in die Schleife in Zeile 13 berechnen. Diese Schleife verwendet dann $\overrightarrow{r_j r_i}$ zusammen mit Gleichung 4.1 lässt sich dann $\overrightarrow{r_j v_u}$ berechnen. Am Ende der Schleife ergibt sich die Menge aller Paare mit Elementen von r_i jedoch in den Koordinaten von r_j . Jetzt lässt sich bezüglich der Koordinaten dieser Paare die Durchschnittsmenge von dieser neu berechneten Menge und von der Menge bilden, die r_j bereits kennt. Ist diese Durchschnittsmenge nicht leer, so bekommen zwei Elemente beim Vereinigen der Bäume die gleiche Koordinate. Anders gesagt: Zwei Puzzlestücke würden sich überschneiden. Andernfalls kann bereits die Vereinigungsmenge der beiden Paar-Mengen gebildet werden, diese muss schließlich die alte Paar-Menge von r_j überschreiben. In der Implementation wird oftmals wie schon erwähnt ein optimistischer Ansatz gewählt und es wird direkt die Vereinigungsmenge gebildet, da die Durchschnittsmenge eigentlich nicht gebraucht wird (außer um auf eine nicht leere Menge zu überprüfen). Nach dem Inklusion-Exklusion-Prinzips lässt sich die Kardinalität der Durchschnittsmenge dann aus den Kardinalitäten der beiden Mengen und der Kardinalität der Vereinigungsmenge berechnen: $|A \cap B| = |A| + |B| - |A \cup B|$. Diese Kardinalität lässt sich dann auf 0 überprüfen und es muss stets nur eine Mengenoperation angewandt werden.

Das Zurücksetzen der Paar-Menge von dem (nun nicht mehr Wurzel-) Element r_i in Zeile 22 ist optional, bietet sich jedoch an um den Speicheraufwand der Datenstruktur zu verringern.

Da bei der Vereinigungsoperation nur ein Element (r_i) seinen Parent (von keinen Parent auf r_j) ändert (abgesehen von der Pfadkomprimierung die implizit im Algorithmus stattfindet), gibt es auch nur eine Parent-Koordinate, welche angepasst werden muss. Dies geschieht in Zeile 21 mit dem zuvor berechneten Vektor $\overrightarrow{r_j r_i}$, welcher genau diese relative Koordinate vom Element r_i relativ zu seinem neuen Parent r_j beschreibt. Alle Elemente unterhalb von r_i ändern zwar implizit ihr Wurzelement von r_i auf r_j , jedoch nicht ihr Parent-Element, womit alle Parent-Koordinaten weiterhin konsistent sind.

Damit ist der Hauptalgorithmus der Datenstruktur komplett beschrieben. An dieser Stelle sollte klar sein, dass iteratives Vereinigen der Bäume zu einer Zusammenhangskomponente führt, welche außerdem keine Puzzlestücke

enthält, die sich überschneiden. Damit ist jedoch noch nicht garantiert, dass das entstandene Bild auch die gewünschten Dimensionen des Puzzles ($m \times n$) besitzt. Auch dazu wird sich an Gallagher [16, 10] orientiert und noch zwei simple Schritte eingeführt: Trimming und Filling.

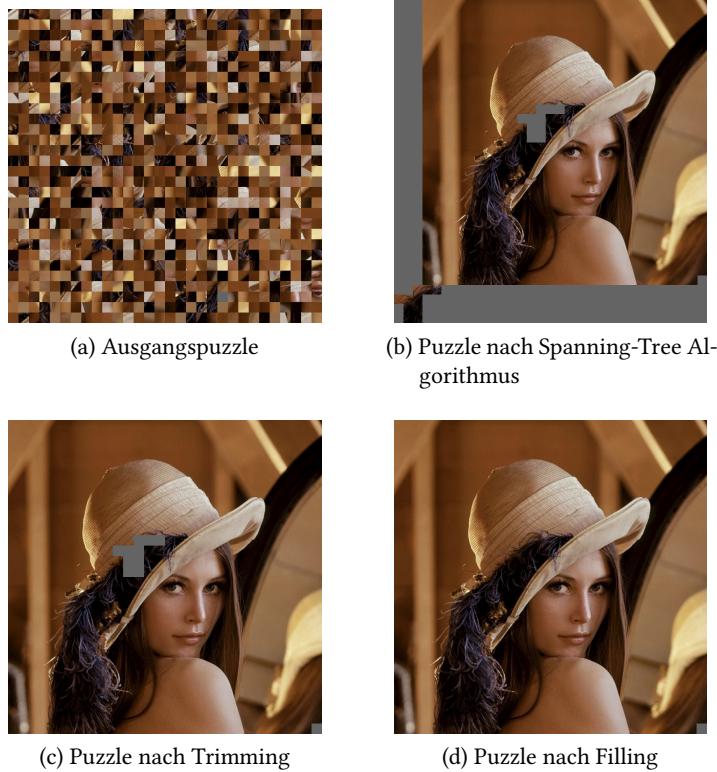


Abbildung 4.2: Die 4 Stages eines Puzzles im Gesamタルgorithmus

4.2 TRIMMING

Sobald nur noch eine Zusammenhangskomponente im Forest vorhanden ist, lässt sich die Wurzel dieses Spannbaums im Array auffinden. Diese Wurzel kennt alle Elemente inklusive den relativen Koordinate zu sich selbst. Mit den minimalen und maximalen x- und y-Koordinaten aller Elemente lässt sich die Anzahl der nötigen Zeilen und Spalten bestimmen, um alle Elemente in einem Rechteck darzustellen. Der Punkt mit der minimalen x-Koordinate aller Elemente und der minimalen y-Koordinate aller Elemente (die obere-linke Ecke des Rechtecks) wird Ursprung des neuen Koordinatensystems. Durch eine simple Vektordifferenz zwischen den Wurzelkoordinaten eines Elementes und den Koordinaten dieses Ursprungs (auch in Wurzelkoordinaten) ergibt sich die Koordinate eines Elementes in dem neuen Frame. Das Ergebnis dieser Rekonstruktion ist in Abbildung 4.2b dargestellt.

Aus diesem Zwischenergebnis soll dann ein Rechteck extrahiert werden, welches die Dimensionen des Puzzles besitzt. Dazu wird dem Algorithmus diese Information mitgeteilt (Anzahl der Zeilen und Spalten vom Puzzle). Dieser legt dann eine Art Sliding-Window über das zuvor rekonstruierte Puzzle und summiert die Anzahl der belegten Puzzlefelder auf. Die Position,

die ein Puzzle ergeben würde, was am dichtesten gefüllt ist (also am meisten Stücke platziert sind), wird als beste Position definiert.

Im nächsten Schritt wird das Puzzle aus 4.2b so getrimmt, dass der Teil des Puzzles, welcher sich innerhalb dieses Sliding-Windows (welches an der zuvor bestimmten Position angebracht wird) befindet, in ein neues Zwischenergebnis übernommen wird. Das dabei entstehende Bild besitzt nun die gewünschten Dimensionsgrößen, enthält aber noch nicht belegte Plätze (vgl. Abbildung 4.2c). Alle Stücke die nicht übernommen wurden, werden in einer Menge zwischengespeichert. Es gilt nun im nächsten Schritt, alle Stücke dieser Menge bestmöglich auf die freien Plätze des Puzzles zu verteilen.

4.3 FILLING

Bevor das Puzzle gefüllt wird, sollen zunächst die möglichen Plätze für Stücke analysiert werden. Die freien Plätze werden dabei in 5 Buckets eingeteilt: Buckets 0 … 4. Die Nummer des Bucket steht für die Anzahl der benachbarten belegten Felder. Die Idee dabei ist, zunächst die Buckets in absteigender Reihenfolge abzuarbeiten, da die Felder mit mehr Nachbarn auch mehr Informationen darüber liefern können, wie gut ein Stück in das Feld passt. Außerdem gilt zu beachten, dass im Laufe dieses Verfahren Puzzlestücke aus den niedrigen Buckets durch das Füllen der Plätze wiederum Nachbarn bekommen und somit solange *promoted* werden, bis diese Stücke selber platziert werden.

Dieses Verfahren funktioniert wie von [16, 10] gezeigt, gut für Puzzles, die keine fehlenden Puzzlestücke haben. Damit ist dies der erste (und auch einzige Schritt) im ganzen Algorithmus, der hier leicht abgeändert werden muss. Schließlich fehlt bei Schiebepuzzles immer exakt ein Puzzlestück. Der Algorithmus wird dementsprechend angepasst. Es sei aber gesagt, dass dieses Verfahren nicht gut erweiterbar ist auf mehrere fehlende Puzzlestücke. Diese Problemstellung ist ein viel schwierigeres Problem, welches auch schon entsprechend Aufmerksamkeit in der Literatur bekommen hat. [17]

Die ursprüngliche Idee ist es, alle Kombinationen aus noch nicht platzierten Stücken und freien Plätzen eines Buckets durchzuprobieren. Dabei wird in jedem Durchgang der höchste Bucket ausgewählt, der mindestens ein Element enthält. Durch die gleichen Distanzen welche auch schon für den Hauptalgorithmus genommen wurden, lässt sich vergleichen, wie gut ein Puzzlestück in ein freies Feld passt. Es wird wieder Greedy vorgegangen, womit das beste Paar aus Puzzlestück und Platz kombiniert werden. Das Stück wird platziert und aus der Menge entfernt. Auch der (nun nicht mehr) freie Platz wird aus seinem entsprechenden Bucket platziert. Außerdem werden dabei ggf. die Nachbarn des Platzes aktualisiert. Befinden sich dort freie Plätze, so können diese alle einen Bucket höher platziert werden (da ein freier Nachbarplatz nun gefüllt wurde). Dieser Vorgang wird so lange wiederholt, wie sich noch nicht platzierte Stücke in der Menge befinden.

An dieser Stelle ist klar, dass dies bei einem Schiebepuzzle ziemlich schief gehen kann. Ist das freie Feld des Schiebepuzzles beispielsweise von keinen freien Feldern umgeben, hat dieses (selbst wenn es schlechte Kompatibilitäten mit den Puzzlestücken aufweist) eine so hohe Priorität bearbeitet zu werden,

dass es mit großer Wahrscheinlichkeit trotzdem fälschlicherweise gefüllt wird. Abbildung 4.3 zeigt ein Beispiel solch eines Fehlers.



Abbildung 4.3: Fehler beim Filling Algorithmus

Um dieses Problem zu lösen, wird das Verfahren nur leicht angepasst. Anstatt den höchsten Bucket mit mindestens einem Element zu nehmen (was dazu führt, dass dieser Platz auf jeden Fall belegt wird, was wiederum zu den Fehler in Abb. 4.3 führt), wird der höchste Bucket mit mindestens zwei Elementen genommen und sofern ein höherer Bucket mit exakt einem Element existiert, dann konkurriert dieses Element mit allen Elementen des niedrigeren Buckets. Existiert der freie Platz des Schiebepuzzles etwa in Bucket 4, so würden zunächst alle anderen freien Plätze mit vier Nachbarn vergeben werden sofern die Metrik stimmt (da es kein passendes Stück für diesen Platz gibt). Danach ist dieses Element alleine im Bucket, wird aber nicht automatisch ausgewählt, sondern wird jetzt zusammen mit den Elementen aus Bucket 3 gegen die noch nicht platzierten Stücke verglichen. Somit ist sichergestellt, dass das Element was am Ende der Prozedur über bleibt, nicht irgendein Element ist, sondern das Element, was im Vergleich zu allen anderen Elementen immer die schlechteste Kompatibilität mit den Puzzlestücken hatte.

4.4 KANTEN-VORVERARBEITUNG

An dieser Stelle sind noch zwei Dinge zu erledigen, die einerseits die Qualität des Algorithmus verbessern und andererseits die Laufzeit durchschnittlich verringern.

Da der Algorithmus keine Form von Backtracking hat (außer in Form von Trimming & Filling), kann selbst eine einzige falsche Zuordnung von Puzzleteilen, welche sehr früh im Algorithmus auftritt, später zu sehr falschen Ergebnissen führen. Intuitiv lässt sich das damit begründen, dass der Fehler nie behoben werden kann (zwei verbundene Stücke bleiben immer verbunden) und der Algorithmus somit um den Fehler herumarbeiten muss, wodurch weitere falsche Entscheidungen getroffen werden (die in diesem Zeitpunkt

als das geringste Übel aussehen und der Algorithmus schließlich greedy vorgeht). Damit pflanzen sich diese Fehler immer weiter fort. Dies ist besonders bei Bildern der Fall, wo viele Stücke ähnlich gute Kompatibilitäten zueinander besitzen. Beispielsweise Puzzlestücke mit Teilen vom Himmel oder wo nur Wasser abgebildet ist. Um diesem Effekt entgegenzuwirken werden die Distanzen der Puzzlestücke nach der initialen Berechnung angepasst. Dazu wird zu jedem Paar von Puzzlestück und Position des Puzzlestücks die Kante mit der zweitniedrigsten Distanz aller dieser Kanten ermittelt (beispielsweise die Kante mit der zweitbesten Distanz von allen Kanten die bei denen ein Teil x_i links vom anderen Puzzlestück liegt).[16, 10] Damit werden die Kantengewichte so normalisiert, dass nur die besten Kanten eine Distanz < 1 aufweisen und alle anderen Distanzen nun ≥ 1 sind. Damit würde der Algorithmus in einem Bild mit einem klaren Himmel also nicht probieren zunächst diesen Himmel komplett anzutragen (obwohl die Kanten dort sehr niedrige absolute Distanzen aufweisen), was mit hoher Wahrscheinlichkeit nicht fehlerlos passieren würde.

Da die Datenstruktur fast den kompletten Algorithmus übernimmt, muss der eigentliche Algorithmus (Kruskal) nur über die Kanten iterieren und die entsprechende Union Operation der Datenstruktur aufrufen, sofern noch kein Spannbaum entstanden ist. Da die Kanten in aufsteigender Reihenfolge (bezüglich der Distanz) abgearbeitet werden sollen, wird dafür oft zunächst die Kantentabelle sortiert (was $\Theta(|E| \log |E|)$ ist). Da der Graph des Puzzles jedoch unglaublich dicht ist (jedes Paar zweier Vertices wird durch 4 Kanten verbunden), werden nur ein Bruchteil dieser Kanten benötigt, um den Graphen zu spannen. Daher bietet es sich an eine Priority-Queue zu verwenden, womit zunächst nur ein $\Theta(|E|)$ Aufwand nötig ist, um die Heapbedingung der Kantentabelle herzustellen und dann das Entfernen einer Kante dann $\mathcal{O}(\log |E|)$ ist. Selbst in dem Fall, dass doch fast alle Kanten betrachtet werden müssen ist dieser Ansatz von der Komplexität her trotzdem nicht schlechter als der Ansatz mit vorherigem Sortieren, da es sich dann lediglich ein Heapsort ergibt, welcher optimales Laufzeitverhalten besitzt.

4.5 ERGEBNIS

Wie bereits in Abbildung 4.2 gezeigt wurde, ist der Algorithmus in der Lage Puzzles von bis zu 1000 Puzzlestücken (die Abbildung zeigt in 30×30 Beispiel) perfekt zu lösen. Diese Genauigkeit ist zu einem großen Teil der MGC-Metrik 3.5 zu verdanken. Puzzles mit bis zu 1000 Stücken werden aufgrund einer effizienten Implementation der Datenstruktur in wenigen Sekunden zusammengefügt. Mögliche Verbesserungen und Erweiterungen bezüglich der Genauigkeit, Laufzeit und auch Speicherverbrauch (zur Speicherung der berechneten Distanzen zwischen Puzzlestücken) sind für das Lösen von sehr großen Puzzles (mehrere tausend Stücke) möglich, jedoch hier nicht nötig. Mit der Hauptmotivation diese Assemblierung als Zwischenschritt für das Lösen von Bild-Schiebepuzzles zu verwenden, sollte bewusst sein, dass die Anzahl der Puzzlestücke von Schiebepuzzles in der Regel nicht viel mehr als etwa 20 sein dürfen, sofern dieses Puzzle optimal von einem Suchalgorithmus gelöst werden soll.

5

LÖSBARKEIT EINES SCHIEBEPUZZLES

In diesem Kapitel wird kurz erklärt, wie geprüft werden kann, ob ein Schiebepuzzle mit einem bestimmten Anfangszustand und einem bestimmten Endzustand lösbar ist. Mit den Ergebnissen der vorherigen Kapitel lässt sich ein $m \times n$ Puzzle aus einer Menge von $N - 1 = mn - 1$ Puzzlestücken zusammensetzen. Diese Zusammensetzung wird als Endzustand anerkannt. Der Startzustand sei beliebig, weshalb deshalb von Interesse ist, ob und wann es überhaupt möglich ist durch eine Reihe von legalen Zügen einen Startzustand in den Endzustand zu bringen. Im Zusammenhang damit ist dann auch interessant, wie sich aus einem Bild (und damit mit einem festen Endzustand) Startzustände generieren lassen, welche immer lösbar sind (dies ist nützlich um Algorithmen zur Lösung des Puzzles zu testen).

Dazu wird angenommen, dass die $N - 1$ Puzzlestücke von 1 bis $N - 1$ nummeriert sind (das leere Feld im Puzzle wird als 0 definiert). Sowohl der Anfangs- als auch der Endzustand ist dann lediglich eine Permutation dieser Zahlen. Es werden dann für beide Zustände die Permutation betrachtet, die beim abtragen der Zahlen in Leserichtung (also von oben nach unten und innerhalb einer Zeile von links nach rechts) entsteht. Das freie Feld wird dabei ignoriert. Damit entstehen für beide Zustände eine Permutation der Form $\alpha, \beta : \{1, \dots, N - 1\} \rightarrow \{1, \dots, N - 1\}$.

Zwischen diesen beiden Permutationen wird die Kendall-Tau-Distanz betrachtet, welche die Summe aller Paare der beiden Permutationen zählt, welche nicht in der gleichen Reihenfolge in den Permutationen vorkommt. Es wird also jedes Paar gezählt, dessen Reihenfolge in den beiden Permutationen verschieden ist. Die Kendall-Tau-Distanz zwischen den beiden Permutationen $(2, 1, 4, 3)$ und $(4, 1, 3, 2)$ ist beispielsweise 4, da die folgenden Paare an Zahlen in verschiedener Reihenfolge in den beiden Permutationen auftreten: $(1, 2), (1, 4), (2, 3), (2, 4)$. Formal lässt sich die Kendall-Tau-Distanz folgendermaßen für die beiden Permutationen α, β berechnen:[27]

$$K(\alpha, \beta) = \sum_{0 < i < j < N} [\beta^{-1}(\alpha(i)) > \beta^{-1}(\alpha(j))] \quad (5.1)$$

Wobei $[\cdot]$ für das Iverson-Bracket steht, welches 1 ergibt, wenn die Bedingung in den eckigen Klammern erfüllt ist und 0 wenn nicht.

Substituiert man nun $\pi = \beta^{-1} \circ \alpha$, erhält man die Definition der Anzahl der Inversionen einer Permutation:[32]

$$inv(\pi) = \sum_{0 < i < j < N} [\pi(i) > \pi(j)] \quad (5.2)$$

Der Zusammenhang dieser beiden Operationen ist, dass die Anzahl der Inversionen einer Permutation die Kendall-Tau-Distanz dieser Permutation mit der Identitätspermutation $id(x) = x$ darstellt.

Die Substitution bewirkt quasi, dass man nun zwei andere Zustände der Schiebepuzzle hat (bezüglich der Zahlenwerte her, das leere Feld bleibt unverändert), indem man den Endzustand als Identitätspermutation definiert hat. Der Startzustand wird durch die Permutation π repräsentiert. Ein $\pi(k)$ ist dann anschaulich der Index aus β an dem der Wert $\alpha(k)$ steht. Hat unser alte Anfangszustand beispielsweise die Zahl 5 im ersten Feld (erste Zeile, erste Spalte) und befindet sich die 5 im Endzustand im dritten Feld (erste Zeile, dritte Spalte), so steht der Wert 3 im ersten Feld des neu definierten Anfangszustands. Diese Umbenennung dient lediglich der einfacheren Handhabung einer Permutation (und einer impliziten Identitätspermutation).

Um das Schiebepuzzle zu lösen, ist es notwendig die Permutation π in die Identitätspermutation zu transformieren. Dies ist noch nicht hinreichend, weil dann noch sichergestellt werden muss, dass nicht nur die Reihenfolge der Zahlenwerte stimmt, sondern auch das leere Feld sich an der richtigen Stelle befindet.

Da die Anzahl der Inversionen einer Permutation nur dann gleich 0 sind, wenn es sich um die Identitätspermutation handelt, lässt sich auch sagen, dass das Ziel ist $inv(\pi)$ auf 0 zu bringen.

Beim Schiebepuzzle sind zwei Arten von Zügen zugelassen: Züge innerhalb einer Zeile und Züge innerhalb einer Spalte. An dieser Stelle lässt sich der Vorteil der Repräsentation als Permutation ohne leeres Feld erkennen, denn ein Zug innerhalb einer Zeile des Puzzles ändert nichts an der Permutation und somit auch nichts an der Anzahl der Inversionen der Permutation.

Für Spaltenzüge wird ein Puzzlestück betrachtet mit Koordinaten (i, j) , wobei i der nullbasierte Zeilenindex ist und j der nullbasierte Spaltenindex. Der nullbasierte Index k in Leserichtung (und diesmal mit dem leeren Element) berechnet sich zu $k = ni + j$. Soll dieses Stück innerhalb seiner Spalte bewegt werden, so muss sich das leere Element an der Stelle $(i \pm 1, j)$ befinden oder auch $l = n(i \pm 1) + j$. Die Anzahl der Elemente zwischen dem Puzzlestück und dem leeren Feld ist $|k - l| - 1 = |ni + j - [n(i \pm 1) + j]| - 1 = |\mp n| - 1 = n - 1$. In der Permutation bedeutet dies, dass das Puzzlestück um $n - 1$ Stellen nach links oder rechts rutscht. Damit ändert sich die relative Position von diesem Puzzlestück mit jedem der $n - 1$ Elementen: Hat eines der Elemente vorher eine Inversion mit dem Stück gebildet, so ist dies jetzt nicht mehr der Fall und umgekehrt auch. Die Anzahl der Inversionen ändert sich also. Es gilt

$$inv(\pi') = inv(\pi) \underbrace{\pm 1 \pm 1 \pm \cdots \pm 1 \pm 1}_{n-1 \text{ mal}} \quad (5.3)$$

An dieser Stelle wird unterschieden zwischen Schiebepuzzles mit einer geraden Anzahl von Spalten geraden n und Schiebepuzzles mit einer ungeraden Anzahl an Spalten.

Für ungerade n ist $n - 1$ gerade. Da eine Summe mit einer geraden Anzahl an ungeraden Summanden stets gerade ist, ist der Ausdruck über der geschweiften Klammer in 5.3 somit auch gerade. Dann ist die Parität von $\text{inv}(\pi')$ aber gleich der Parität von $\text{inv}(\pi)$. Anders gesagt gilt

$$\text{inv}(\pi') \equiv \text{inv}(\pi) \pmod{2}$$

Damit ändert sich die Anzahl der Inversionen für ungerade n bei einem Zeilenzug garnicht womit die Parität der Anzahl der Inversionen trivialerweise gleich bleibt. Und da diese auch bei einem Spaltenzug gleich bleibt, lässt sich sagen, dass die Parität der Anzahl der Inversionen einer Permutation unter allen legalen Zügen des Schiebepuzzles invariant ist. Um die Anzahl der Inversionen also auf 0 (eine gerade Zahl) bringen zu können (und das Puzzle damit zu lösen), muss die Permutation des Anfangszustand eine gerade Anzahl an Inversionen aufweisen.

Im anderen Fall (n ist ungerade), lässt sich sagen, dass die Summe über der geschweiften Klammer in 5.3 stets ungerade ist, wodurch sich die Parität mit jedem Spaltenzug ändert. Deshalb werden für ungerade n auch noch die Zeilenindizes r, s der leeren Felder in beiden Zuständen betrachtet. Um das Puzzle zu lösen ist es auch notwendig das leere Feld an der richtigen Stelle zu haben (und damit besonders in der richtigen Zeile). Betrachtet man den Zeilenabstand der beiden leeren Felder, also $|r - s|$, so ist klar, dass auch dieser notwendigerweise 0 sein muss. Dieser Zeilenabstand ändert sich aber bei jedem Spaltenzug um ± 1 . Damit ändert sich die Parität von $|r - s|$ bei jedem Spaltenzug. Zusammengefasst lässt sich schließen, dass die Puzzles mit ungeraden n die Parität der Anzahl der Inversionen gleich der Parität des Zeilenabstand beider leeren Felder sein muss. Da beide Attribute auf 0 gebracht werden müssen beim Lösen und somit am Ende auf eine gerade Parität kommen müssen, beide aber in jedem Spaltenzug ihre Parität ändern, würde eine ungleiche Parität der beiden dazu führen, dass in jedem Schritt genau einer von beiden dafür sorgt, dass das Puzzle nicht gelöst sein kann.

Die obigen Bedingungen müssen damit notwendigerweise von lösbarren Schiebepuzzles erfüllt sein. Die andere Richtung (dass diese Bedingungen tatsächlich auch hinreichend sind) wurde von Johnson & Story [33] gezeigt. Dementsprechend gilt für alle Schiebepuzzle, dass diese dann und nur dann lösbar sind, wenn diese die obigen Bedingungen erfüllen. Tatsächlich wurde in diesem Paper auch gezeigt, dass die Menge aller Puzzlezustände sich in zwei gleich große Äquivalenzklassen partitionieren lässt. Die Zustände dieser Äquivalenzklassen lassen sich stets unter Verwendung legaler Züge nur in andere Zustände der selben Äquivalenzklasse transformieren. Damit ist immer genau die Hälfte aller Permutationen eines Schiebepuzzles lösbar.

Um die Lösbarkeit eines Schiebepuzzles zu bestimmen muss also lediglich die Parität der Anzahl der Inversionen der Permutation des Schiebepuzzles (und je nach Spaltenanzahl auch noch die Parität des Zeilenabstandes der beiden leeren Felder) überprüft werden. Da diese Überprüfungen alle konstant sind, hängt die Laufzeit- und Speicherkomplexität lediglich von der Bestimmung der zusammengesetzten Permutation π ab (dies ist ein linearer Zeit mit linearem Speicher möglich) und der Berechnung der Parität.

Eine triviale Möglichkeit die Parität zu bestimmen basiert auf der Berechnung der Inversionen nach Gleichung 5.2. Da die Summe jedoch über $\binom{N-1}{2}$

geht, ist die Laufzeitkomplexität hier mit $\Theta(N^2)$ quadratisch. Dafür wird aber an dieser Stelle nicht noch weiterer zusätzlicher Speicher benötigt.

Eine Möglichkeit die Anzahl der Inversionen zu bestimmen ergibt sich durch eine Erweiterung des Mergesort-Algorithmus.[27] Die Permutation wird in zwei gleich große Hälften zerlegt. Die gesamte Anzahl an Inversionen lässt sich dann in drei Summanden zerlegen: Alle Inversionen die in der linken Hälfte auftreten, alle Inversionen die in der rechten Hälfte auftreten und alle Inversionen, die übergreifend der Hälften auftreten. Die ersten beiden Summanden ergeben sich durch die Rekursion (der Basisfall der Rekursion sind Permutation mit weniger als zwei Elementen, welche trivialerweise keine Inversionen haben können). Die Inversionen, welche zwischen den beiden (nun auch sortierten) Hälften auftreten, lassen sich in dem Merge-Schritt des Algorithmus mitzählen. Da jedes Element aus der linken Hälfte in der Permutation vor jedem Element aus der rechten Hälfte kam, so entstehen genau dann Inversionen, wenn das aktuell betrachtete Element aus der rechten Hälfte kleiner ist als das aktuell betrachtete Element aus der linken Hälfte. Dieses Element der rechten Hälfte bildet dann mit allen verbleibenden Elementen der linken Hälfte eine Inversion, da es in der Permutation zwar rechts von diesen Elementen stand, jedoch kleiner als alle dieser Elemente ist. Es folgt ein Beispiel mit der Permutation $\pi = (3, 4, 7, 9, 2, 5, 6, 8)$. Beide Hälften $\lambda = (3, 4, 7, 9), \mu = (2, 5, 6, 8)$ sind bereits sortiert, womit innerhalb der beiden Hälften keine Inversionen auftreten. Es gilt nun diese zusammenzufügen, um die Inversionen zwischen den beiden Hälften zu zählen.

$$\lambda = (3, 4, 7, 9), \quad \mu = (\underline{2}, 5, 6, 8), \quad \pi' = (), \quad \text{inv}(\pi) = 0$$

2 ist das Minimum und bildet mit allen restlichen Elementen aus λ eine Inversion. Damit ergeben sich 4 Inversionen beim Merge.

$$\lambda = (\underline{3}, 4, 7, 9), \quad \mu = (\underline{2}, 5, 6, 8), \quad \pi' = (2), \quad \text{inv}(\pi) = 4$$

Das nächste Minimum kommt aus der linken Hälfte und bildet somit keine neuen Inversionen.

$$\lambda = (\underline{3}, \underline{4}, 7, 9), \quad \mu = (\underline{2}, 5, 6, 8), \quad \pi' = (2, 3), \quad \text{inv}(\pi) = 4$$

$$\lambda = (\underline{3}, \underline{4}, 7, 9), \quad \mu = (\underline{2}, \underline{5}, 6, 8), \quad \pi' = (2, 3, 4), \quad \text{inv}(\pi) = 4$$

5 bildet mit den restlichen beiden Elementen aus λ (Elemente 7, 9) Inversionen.

$$\lambda = (\underline{3}, \underline{4}, 7, 9), \quad \mu = (\underline{2}, \underline{5}, \underline{6}, 8), \quad \pi' = (2, 3, 4, 5), \quad \text{inv}(\pi) = 6$$

$$\lambda = (\underline{3}, \underline{4}, \underline{7}, 9), \quad \mu = (\underline{2}, \underline{5}, \underline{6}, 8), \quad \pi' = (2, 3, 4, 5, 6), \quad \text{inv}(\pi) = 8$$

$$\lambda = (\underline{3}, \underline{4}, 7, 9), \quad \mu = (\underline{2}, \underline{5}, \underline{6}, \underline{8}), \quad \pi' = (2, 3, 4, 5, 6, 7), \quad \text{inv}(\pi) = 8$$

$$\lambda = (\underline{3}, \underline{4}, 7, \underline{9}), \quad \mu = (\underline{2}, \underline{5}, \underline{6}, \underline{8}), \quad \pi' = (2, 3, 4, 5, 6, 7, 8), \quad \text{inv}(\pi) = 9$$

$$\lambda = (\underline{3}, \underline{4}, 7, \underline{9}), \quad \mu = (\underline{2}, \underline{5}, \underline{6}, \underline{8}), \quad \pi' = (2, 3, 4, 5, 6, 7, 8, 9), \quad \text{inv}(\pi) = 9$$

Es ergibt sich für π eine Inversionszahl von 9.

Der Algorithmus besitzt somit die gleichen Eigenschaften wie ein normaler Mergesort, womit sich eine Laufzeitkomplexität von $\Theta(N \log N)$ ergibt. Es ist zusätzlich noch linearer Speicheraufwand nötig (für das temporäre Array, welches zum Mergen benötigt wird). Da die Berechnung der zusammengesetzten Permutation π jedoch bereits linearen Speicher braucht, ändert dies nichts an der Speicherkomplexität der kompletten Prozedur.

Um die komplette Prozedur sowohl in Zeit, als auch in Speicher linear zu halten, muss ein anderer Ansatz her. Die entscheidende Beobachtung dafür ist, dass die exakte Anzahl der Inversionen garnicht nötig ist und lediglich die Parität dieser Anzahl eine Rolle spielt. Diese Parität der Inversionen wird tatsächlich als Parität der Permutation definiert und es gibt viele äquivalente Definitionen der Parität einer Permutation. Eine dieser Definitionen definiert die Parität einer Permutation als die Parität der Anzahl der Transpositionen*, die nötig sind, um die Identitätspermutation in diese Permutation zu transformieren. Es sei zu beachten, dass die Anzahl der Transpositionen dabei (im Gegensatz zu der Anzahl der Inversionen) für eine Permutation nicht eindeutig ist. Die Parität dieser Anzahl ist jedoch immer gleich. Eine Permutation die beispielsweise 5 Transpositionen benötigt, lässt sich auch in 7 Transpositionen erstellen, indem am Ende der ersten 5 noch eine willkürliche Transposition durchgeführt wird und dann sofort die inverse dieser Transposition angewendet wird.

Um die Äquivalenz der beiden Definitionen zu zeigen wird ein induktiver Ansatz gewählt. Offensichtlich lässt sich die Identitätspermutation durch das anwenden von 0 Transpositionen in die Identitätspermutation transformieren. Außerdem hat die Identitätspermutation auch 0 Inversionen, beide Definitionen liefern also die gleiche Parität. Sind beide Paritäten für eine Permutation π nun gleich und wird eine beliebige Transposition $\tau_{\pi(i), \pi(j)}$ mit $i < j$ auf π angewandt, so erhöht sich die Anzahl der Transpositionen um 1 und $\tau_{\pi(i), \pi(j)} \circ \pi$ besitzt die inverse Parität von π . Nun gilt es zu betrachten, was mit $inv(\pi)$ bei der Anwendung der Transposition passiert. Alle Inversionen der Permutation mit Elementen links von i oder rechts von j bleiben bei der Transposition erhalten. Nun befinden sich zwischen i und j genau $h = j - i - 1$ Elemente. Jedes dieser Elemente könnte sowohl mit dem Element an der Stelle i , als auch mit dem Element an der Stelle j eine Inversion gebildet haben. Sei nun p die Anzahl dieser Elemente, welche eine Inversion mit dem Element an der Stelle i gebildet haben und q die Anzahl der Elemente, die eine Inversion mit j formen. Beim Durchführen der Transposition gehen somit $p + q$ Inversionen verloren, es entstehen aber dafür $(h - p) + (h - q)$ neue Inversionen. Nun ist noch zu beachten, dass die Elemente an den Stellen i und j selber ihre relative Position ändern und somit die Anzahl der neuen Inversionen entweder inkrementieren oder dekrementieren. Die neue Anzahl der Inversionen ist also $inv(\tau_{\pi(i), \pi(j)} \circ \pi) = inv(\pi) - (p + q) + (h - p) + (h - q) \pm 1 = inv(\pi) + 2h - 2p - 2q \pm 1 = inv(\pi) + 2(h - p - q) \pm 1$. Es wird also eine ungerade Zahl auf die (alte) Anzahl der Inversionen addiert. Damit ändert sich die Parität der Anzahl der Inversionen bei jeder Anwendung einer Transposition. Da beide Definition für die Identitätspermutation die gleiche Parität liefern, die Anwendung jeder Transposition auf eine Permutation beide Paritäten invertiert und jede Permutation sich als die Komposition endlich vieler Transpositionen darstellen lässt, sind beide Definitionen der Parität einer Permutation für alle Permutationen gleich.

Es reicht also, einen Algorithmus zu finden, der eine (beliebige) Zerlegung einer Permutation in Transpositionen findet. Die Anzahl dieser Transpositio-

* Eine Transposition $\tau_{a,b}$ ist eine Permutation, welche zwei Elemente tauscht (also a auf b abbildet und b auf a abbildet) und alle anderen Elemente gleich lässt (also alle anderen Elemente c wiederum auf c abbildet).

nen könnte sich zählen lassen, womit die Parität bekannt wäre. Dazu wird noch eine finale Eigenschaft von Permutationen betrachtet.

Permutationen lassen sich (da diese bijektive Funktionen sind) in disjunkte Zyklen zerlegen. Die Permutation $\pi = (3, 6, 1, 2, 5, 4)$ bildet etwa 1 auf 3 ab, 3 wird aber wiederum auf 1 abgebildet, womit der Zyklus $(1, 3)$ entsteht (dies entspricht einer Transposition). Auch gilt $2 \mapsto 6, 6 \mapsto 4, 4 \mapsto 2$ womit der nächste Zyklus $(2, 6, 4)$ entsteht. Als letztes wird 5 auf 5 abgebildet, was auch ein Zyklus darstellt (5) (ein sogenannter *fixed-point*). Die Permutation kann dann als das Produkt solcher Zyklen ausgedrückt werden (Zyklenschreibweise). Eine Möglichkeit dies auszudrücken wäre nach oben dann $\pi = (1, 3)(2, 6, 4)(5)$.

Ein Zyklus $\sigma = (a_1, a_2, \dots, a_k, \dots, a_{n-1}, a_n)$ der Länge n lässt sich aber als die Komposition von Transpositionen darstellen. Von besonderem Interesse ist die Anzahl der Transpositionen die dafür nötig sind. Diese ist zwar nicht klar definiert, die Parität der Anzahl aber schon, womit es hinreichend ist eine solche Komposition zu finden. Zwei solcher Kompositionen wären beispielsweise $\sigma = \tau_{a_1, a_n} \circ \tau_{a_1, a_{n-1}} \cdots \circ \tau_{a_1, a_k} \cdots \circ \tau_{a_1, a_3} \circ \tau_{a_1, a_2}$ oder auch $\sigma = \tau_{a_1, a_2} \circ \tau_{a_2, a_3} \circ \dots \tau_{a_k, a_{k+1}} \cdots \circ \tau_{a_{n-2}, a_{n-1}} \circ \tau_{a_{n-1}, a_n}$. Beide Zerlegungen haben die Länge $n - 1$.

Eine Permutation lässt sich also in disjunkte Zyklen zerlegen, welche sich alle wiederum in Transpositionen zerlegen lassen können. Die Parität der Anzahl aller Transpositionen ist dann die gesuchte Parität der Permutation. Da die Parität einer Summe durch die Anzahl der ungeraden Summanden bestimmt ist, ist die Parität der Permutation die Parität der Anzahl der Zyklen mit einer ungeraden Anzahl an Transpositionen. Da eine mögliche Zerlegung eines Zyklus der Länge n aber $n - 1$ Transpositionen aufweist, ist die Anzahl der Transpositionen für einen Zyklus genau dann ungerade, wenn die Länge des Zyklus gerade ist. Zusammengefasst ist die Parität einer Permutation also die Parität der Anzahl der Zyklen mit gerader Länge. Das Beispiel $\pi = (1, 3)(2, 6, 4)(5)$ ist eine ungerade Permutation, da es genau einen Zyklus mit gerader Länge gibt $(1, 3)$ und die Anzahl der Zyklen mit gerader Länge somit ungerade ist.

Ein Algorithmus zur Bestimmung der Zerlegungen einer Permutation in disjunkte Zyklen geht so vor, dass beim ersten Element i der Menge gestartet wird und dies als *gesehen* markiert wird. Dann wird das Element $\pi(i)$ betrachtet, dann $\pi(\pi(i))$ und so weiter, bis die Permutation ein Element wieder auf i abgebildet hat. Auf dem Weg werden alle betrachteten Elemente als *gesehen* markiert. Außerdem wird mitgezählt wie viele Kompositionen von π nötig waren, um den Zyklus zu schließen. Diese Anzahl ist die Länge des Zyklus. Außerdem wird insgesamt mitgezählt wieviele dieser Zyklen eine gerade Länge hatten. Nachdem der erste Zyklus traversiert wurde, werden die nächsten Elemente der Menge betrachtet und sofern diese noch nicht als *gesehen* markiert wurden (diese Elemente sind Teil eines bereits betrachteten Zyklus), wird die beschriebene Prozedur von diesem Element aus durchgeführt. Am Ende ergibt sich die Anzahl der Zyklen mit gerader Länge. Die Parität dieser Anzahl ist die gesuchte Parität der Permutation.

Dieser Algorithmus benötigt zwar für das *gesehen*-Array linearen Speicher (was jedoch nicht schlechter ist als der Mergesort-Ansatz), da jedoch jeder Zyklus nur ein einziges mal traversiert wird, werden die Elemente der Per-

mutation auch nicht mehr als zwei mal betrachtet, womit der Algorithmus linear in der Länge der Permutation ist.

Damit lässt sich in linearer Zeit und mit linearem Speicher sagen, ob ein gegebenes Puzzle lösbar ist oder nicht. Um zusätzlich noch gezielt Anfangszustände generieren zu können, die (bezüglich eines gegebenen Endzustand) immer lösbar oder nicht lösbar sein sollen, so lässt sich einfach eine zufällige Permutation generieren. Diese wird dann auf Lösbarkeit überprüft. Ist diese Lösbarkeit verschieden von der gewünschten Lösbarkeit, so wird exakt eine Transposition durchgeführt. Es werden also zwei beliebige Elemente vertauscht (wobei keines der beiden Elemente das leere Feld ist und die beiden Elemente auch verschieden sind). Wie bereits erwähnt, ändert jede Transposition die Parität einer Permutation. Damit hat der Anfangszustand nun die gewünschte Lösbarkeit.

6

LÖSEN DES SCHIEBEPUZZLES

Nachdem die einzelnen Puzzlestücke zu einem gewünschten Endzustand des eigentlichen Schiebepuzzles zusammengefügt worden sind und nachdem außerdem sichergestellt wurde, dass das Puzzle unter einem gegebenem Anfangszustand auch lösbar ist, soll nun die eigentliche Lösung des Schiebepuzzles ermittelt werden. Gesucht ist dabei eine Lösung (also eine Sequenz von legalen Zügen), die den Anfangszustand in den Endzustand überführt. Diese Lösung soll außerdem optimal sein. Es darf also keine andere Lösung existieren, die das Schiebepuzzle in weniger Schritten löst.

Dazu werden in diesem Kapitel kurz bekannte Suchalgorithmen aus dem Feld der künstlichen Intelligenz angesprochen und gezeigt, wie diese genutzt werden können, um das Schiebepuzzle optimal zu lösen.

Das Problem des Schiebepuzzles lässt sich mit einem sogenannten Zustandsgraphen beschreiben. Ein Zustand des Schiebepuzzles entspricht einer möglichen Anordnung der Felder. Bei einem $N = m \times n$ Puzzle sind $N!$ Anordnungen möglich. Da vom Startzustand aus aber nur die Hälfte dieser Anordnungen auch wirklich erreichbar sind (vgl. Kapitel 5), besitzt der Zustandsgraph eines Puzzles $\frac{N!}{2}$ Vertices. Jeder Zustand lässt sich durch das Schieben eines Puzzlestückes in das leere Feld in einen anderen Zustand überführen. Diese Verbindung wird als Kante im Zustandsgraphen dargestellt. Diese Kante ist im konkreten Fall hier ungerichtet (da jeder Zug wieder umkehrbar ist). Da als Kriterium für das Optimum einer Lösung die Anzahl der nötigen Züge genommen wird, lässt sich sagen, dass es sich um einen gewichteten Graphen handelt, wobei jede Kante ein Kantengewicht von 1 besitzt. Damit entspricht die Anzahl der Züge dann die Kantengewichtssumme eines Weges durch den Graphen. Beim Schiebepuzzle besitzt jeder Zustandsvertex einen Grad (Anzahl der ausgehenden Kanten des Vertex) der Werte zwischen 2 (bei Zuständen, wo das leere Feld in einer Ecke ist) und 4 (bei Zuständen, wo das leere Feld sich nicht an der Seite des Puzzles befindet) annehmen kann. In der künstlichen Intelligenz wird dieser Wert auch als *Branching-Faktor b* bezeichnet. Es ist außerdem von Bedeutung zu erwähnen, dass der Zustandsgraph des Schiebepuzzles Zyklen enthält.

Das Problem der Lösungsfindung im Schiebepuzzle lässt sich nun als ein Suchproblem in einem Graphen formulieren. Es ist ein Suchalgorithmus gefragt, der den kürzesten Weg vom Startzustand zum Endzustand im Zu-

standsgraphen findet. Dafür werden im Folgenden zunächst zwei triviale uninformierte Suchverfahren betrachtet und daraufhin zwei informiert Suchverfahren, welche auf einer Heuristikfunktion basieren.

6.1 BREADTH-FIRST SEARCH (BFS)

Der bekannte Breadth-First Search (BFS) Algorithmus zur Traversierung eines Graphen kann genutzt werden, um eine optimale Lösung des Schiebe-puzzles aufzufinden. Dazu wird der Zustandsgraph $G = (V, E)$ betrachtet mit Startzustand s und Endzustand g (Goal-State). Alle entdeckten, aber noch nicht abgearbeiteten Knoten (die Nachbarn der Knoten wurden noch nicht *expanded*) werden in einer Datenstruktur verwaltet. Beim BFS ist diese Datenstruktur eine (FIFO-) Queue. Diese Menge der Knoten, welche noch entdeckt worden sind, aber nicht untersucht wurden, wird auch als *Fringe*, *Open-Set* oder *Frontier* bezeichnet. Obwohl BFS immer eine Lösung findet (sofern diese existiert) und nicht in Schleifen gefangen werden kann (wie etwa DFS) wird in der Regel auch in *Closed-Set* eingeführt, welches verhindern soll, dass ein Zustand mehr als ein mal expanded wird. Um später den eigentlichen Lösungsweg ausgeben zu können, lässt sich dies damit verbinden, das *Closed-Set* zur einer *Parent-Map* erweitert wird, wo auch noch zu jedem Zustand mitgespeichert wird, von welchem Zustand aus dieser Zustand entdeckt wurde. Dann lässt sich vom Endzustand aus dieser Weg (rückwärts) zurückverfolgen. Es folgt ein Ausschnitt an Pseudocode, welcher BFS darstellt.

Algorithm 8 Breadth-First Search (BFS)

```

1: function BFS( $s, g$ )
2:   frontier  $\leftarrow$  Queue{ $s$ }
3:   parent  $\leftarrow$  Map{ $(s, \emptyset)$ }
4:   repeat
5:      $v \leftarrow$  frontier.dequeue()
6:     if  $v = g$  then
7:       return SUCCESS
8:     end if
9:     for all  $(v, u) \in E$  do
10:      if  $\forall w \in V : (u, w) \notin \text{parent}$  then
11:        parent  $\leftarrow$  parent  $\cup \{(u, v)\}$ 
12:        frontier.enqueue( $u$ )
13:      end if
14:    end for
15:   until frontier  $= \emptyset$ 
16:   return FAILURE
17: end function

```

BFS findet zwar eine optimale Lösung, dafür wächst aber sowohl das Laufzeit- als auch das Speicherverhalten des Algorithmus exponentiell mit der Suchtiefe. Bezüglich einer Suchtiefe d im Graphen hat BFS ein exponentielle Laufzeitverhalten der Form $\mathcal{O}(b^d)$. Die Laufzeit ist an dieser Stelle

nicht das Problem (es wird auch mit besseren Algorithmen asymptotisch exponentiell bleiben). BFS benötigt aber bereits zur Erhaltung des Frontiers auch einen exponentiellen Speicheraufwand (in der letzten Tiefe befinden sich b^d Elemente, welche zu einem Zeitpunkt alle gleichzeitig im Frontier liegen werden). Deshalb wird im nächsten Schritt eine simple Alternative zum BFS untersucht.

6.2 ITERATIVE DEEPENING DEPTH-FIRST SEARCH (IDDFS)

Die Tiefensuche (Depth-First Search - DFS) stellt das Gegenstück zur Breitensuche (BFS) dar. Da diese jedoch generell keine optimalen Lösungen liefert, ist der reine Algorithmus hier ungeeignet. Wird dieser Suchalgorithmus jedoch mit Iterative Deepening kombiniert, so stellelt der entstehende Algorithmus (IDDFS) eine gute Alternative zu BFS dar. Dazu wird die gängige DFS Implementation (welche in der Regel rekursiv implementiert wird) durch einen weiteren Parameter d_{max} erweiter, welcher das Tiefenlimit darstellt. Wird dieses innerhalb der Suche erreicht, so wird nicht rekursiv noch tiefer gesucht. Dann lässt sich diese Version von DFS mit immer größer werdender Tiefe aufgerufen. Es entsteht quasi eine Art Pseudo-BFS. Dazu wird zunächst ein limitierter DFS Algorithmus implementiert und dann der eigentliche IDDFS.

Algorithm 9 Iterative Deepening Depth-First Search (IDDFS)

```

1: function DFS( $v, g, d, d_{max}, \text{parent}$ )
2:   if  $d > d_{max}$  then
3:     return FAILURE
4:   end if
5:   if  $v = g$  then
6:     return SUCCESS
7:   end if
8:   for all  $(v, u) \in E$  do
9:     if  $\forall w \in V : (u, w) \notin \text{parent}$  then
10:       $\text{parent} \leftarrow \text{parent} \cup \{(u, v)\}$ 
11:      if DFS( $u, g, d + 1, d_{max}, \text{parent}$ ) then
12:        return SUCCESS
13:      end if
14:       $\text{parent} \leftarrow \text{parent} \setminus \{(u, v)\}$ 
15:    end if
16:   end for
17:   return FAILURE
18: end function
19: function IDDFS( $s, g$ )
20:    $\text{parent} \leftarrow \text{Map}\{(s, \emptyset)\}$ 
21:   for  $d_{max} = 0 \dots \infty$  do
22:     if DFS( $s, g, 0, d_{max}, \text{parent}$ ) = SUCCESS then
23:       return SUCCESS
24:     end if
25:   end for
26:   return FAILURE
27: end function

```

Man könnte meinen, dass das Laufzeitverhalten darunter leidet, dass Knoten in den höheren Tiefen mehrmals besucht werden. Tatsächlich stellt sicher aber heraus, dass der Aufwand lediglich um einen konstanten Faktor (abhängig von b , nicht jedoch von d) wächst.[11]

Im Gegensatz zum DFS ist nun aber die Speicherkomplexität nicht mehr exponentiell, da es einerseits kein Frontier mehr gibt und andererseits nicht alle besuchten Knoten in der Parent-Map gespeichert werden, sondern nur die Knoten des Pfades, der aktuell in der Rekursion untersucht wird (man beachte Zeilen 10 und 14 aus Algorithmus 9). Damit ist die Speicherkomplexität linear bezüglich der Tiefe d und somit $\mathcal{O}(d)$.

6.3 HEURISTIKEN

Um eine Lösung des Puzzles doch schneller finden zu können, werden an dieser Stelle informierte Suchalgorithmen eingeführt, also Suchalgorithmen die auf Heuristiken basieren. Dies bewirkt keine Veränderung der Laufzeitkomplexität, sorgt aber dafür, dass der richtige Weg zum Ziel in der Regel schneller gefunden wird, indem schlecht aussehende Wege erst dann weiter verfolgt werden, wenn auch die guten Wege beim weiterverfolgen kein bes-

seres Ergebnis liefern. Abstrakt lässt sich sagen, dass der Branching-Faktor b in einen neuen kleineren *effective* Branching-Faktor b^* überführt wird.

Eine Heuristikfunktion $h(n)$ ordnet jedem Vertex im Zustandsgraphen eine nicht-negative Zahl zu ($\forall n \in V : h(n) \geq 0$). Der Wert dieser Zahl gilt als Abschätzung dafür, wie weit n vom Endzustand entfernt ist. Die Heuristikfunktion nimmt beim Endzustand den Wert 0 an, also $h(g) = 0$.

Es sind zwei Eigenschaften von Heuristikfunktionen zu untersuchen, welche bei der Optimalität von informierten Suchalgorithmen von großer Bedeutung sind: *Admissibility* (Zulässigkeit) und *Consistency* (Konsistenz).

Eine *admissible* Heuristik überschätzt niemals die echte Distanz, die ein Knoten n zum Ziel hat. Sei $h^*(n)$ die *perfekte* Heuristikfunktion (die Heuristik, die jedem Knoten die exakte Distanz zum Ziel zuordnet), dann ist eine Heuristik dann und nur dann Zulässig, wenn

$$\forall n \in V : h(n) \leq h^*(n) \quad (6.1)$$

Eine *consistent* Heuristik erfüllt in gewisser Maßen eine Form der Dreiecksungleich. Bei solchen Heuristiken sind die totalen Kosten (Summe der absoluten Kosten vom Startzustand bis zu einem Knoten und der geschätzten Kosten der Heuristikfunktion an diesem Knoten) entlang eines Weges im Graphen stets monoton steigend. Dafür darf die Änderung in der Heuristik beim traversieren einer Kante nicht größer sein als die Kosten dieser Kante. Wenn $C(a, b)$ die Kosten einer Kante (a, b) sind, gilt also:

$$\forall(a, b) \in E : h(a) - h(b) \leq C(a, b) \quad (6.2)$$

$$\iff \forall(a, b) \in E : h(a) \leq C(a, b) + h(b) \quad (6.3)$$

Nun lässt sich induktiv zeigen, dass die *Consistency* einer Heuristik die *Admissibility* dieser Heuristik impliziert (der umgekehrte Fall ist nicht unbedingt wahr).

Es ist offensichtlich, dass eine Heuristik h am Endzustand stets Kriterium 6.1 erfüllt, da dort $h(g) = 0 = h^*(g)$ und damit $h(g) \leq h^*(g)$ wahr ist. Für einen Zustand a , der den Endzustand optimal in exakt $n+1$ Schritten erreicht, existiert ein Zustand b , der den Endzustand optimal in n Schritten erreicht (der direkte Nachfolger von a auf diesem Weg). Ist die Heuristik nun admissible für alle Zustände, die den Endzustand optimal in n Schritten erreichen, dann gilt nach 6.1 $h(b) \leq h^*(b)$. Wenn h aber auch konsistent ist, dann gilt nach 6.3 $h(a) \leq C(a, b) + h(b)$. Dann gilt insgesamt $h(a) \leq C(a, b) + h^*(b)$ und somit $h(a) \leq h^*(a)$. Die Heuristik ist also auch für a admissible. Da a beliebig war, ist die Heuristik h für alle Vertices admissible, die den Endzustand optimal in $n+1$ Schritten erreichen, wenn dies bereits für alle Vertices gilt, die den Endzustand optimal in n Schritten erreichen. Da aber der Endzustand sich selber optimal in 0 Schritten erreicht und die Heuristik dort admissible ist, ist die Induktion vollständig und eine konsistente Heuristik erfüllt das Admissibility-Kriterium für alle Vertices des Graphen.

An dieser Stelle werden noch zwei simple Heuristiken für das Schiebe-puzzle betrachtet.

6.3.1 Hamming-Distanz

Die Hamming Distanz zählt lediglich wie viele der Puzzlestücke sich nicht an der richtigen Stelle befinden. Bei einem $N = m \times n$ Schiebepuzzle nimmt die Heuristikfunktion also stets Werte zwischen 0 (das Puzzle ist gelöst) und $N - 1$ (kein Stück befindet sich an der richtigen Stelle) an.

Um zu zeigen, dass diese Heuristik admissible und konsistent ist, reicht es zu zeigen, dass die Heuristik konsistent ist. Dazu wird Gleichung 6.2 betrachtet, welche aussagt, dass die Änderung in der Heuristik beim Durchführen eines Zuges nicht größer sein darf, als die Kosten dieses Zuges. Da die Kosten beim Schiebepuzzle für alle Züge stets 1 ist, muss also $h(a) - h(b) \leq 1$ gelten. Die Heuristik darf also beim Zug um nicht mehr als 1 verringert werden. Da es beim Schiebepuzzle nicht möglich ist, innerhalb von nur einem Zug zwei Puzzlestücke, welche zuvor beide an der falschen Stelle waren, an die richtige Position zu bringen, kann die Differenz der Heuristik nicht größer als 1 sein. Damit ist die Hamming-Distanz eine konsistente und auch admissible Heuristik für das Schiebepuzzle.

6.4 SUMME DER MANHATTEN-DISTANZEN

Es wird die Manhattan-Distanz zwischen einem Puzzlestück im Anfangszustand und dem gleichen Puzzlestück im Endzustand gebildet. Diese Distanz ist die Summe der absoluten Zeilendifferenz und der absoluten Spaltendifferenz dieser Stücke. Diese Summe wird weiterhin für alle Puzzlestücke gebildet und aufsummiert.

Auch diese Heuristik ist konsistent (und admissible), da ein Zug diese Summe niemals um mehr als 1 verringern kann, weil bei jedem Zug schließlich nur die Manhattan-Distanz von einem Puzzlestück beeinflusst wird und dies nur entlang einer Achse, womit die Manhattan-Distanz dieses einen Stücks sich um ± 1 verändert.

Nun lässt sich weiterhin noch sagen, dass die Summe der Manhattan Distanzen niemals kleiner sein kann als die Hamming-Distanz des gleichen Puzzles. Während die Hamming-Distanz für jedes Paar an Puzzlestücken, welches nicht an der richtigen Stelle ist, 1 aufsummiert, wird bei der Manhattan-Distanz ein Wert aufsummiert, welcher auch mindestens 1 ist (da das Stück nicht an der richtigen Stelle steht und somit mindestens entlang einer Achse ein Summand größer 0 entsteht). Da dieser Wert in den meisten Fällen sogar echt größer als 1 ist (das Stück befindet sich also nicht direkt links oder rechts oder unten oder oben von dem richtigen Platz), lässt sich sagen, dass die Summe der Manhattan-Distanzen eine strengere Heuristik ist. Da diese aber weiterhin noch admissible ist und somit die echte Distanz einer Zustands nicht überschätzt, ist diese Heuristik also insgesamt genauer (entspricht eher der Wahrheit als die Hamming-Distanz). Man sagt auch, dass eine die Summe der Manhattan-Distanzen die Hamming-Distanz als Heuristik *dominiert*. Damit ist die Summe der Manhattan-Distanzen als Heuristik nie schlechter (und in der Regel besser) als die Hamming-Distanz. Diese Heuristik wird für die folgenden informierten Suchalgorithmen implementiert.

6.5 A*

Der A*-Suchalgorithmus ist ein optimaler Suchalgorithmus, welcher vom algorithmischen Aufbau her dem BFS-Algorithmus in 8 ähnelt. Der wichtige Unterschied dabei ist, dass Anstatt einer normalen FIFO-Queue eine Priority-Queue als Frontier genutzt wird und die Knoten somit entsprechend eines A*-Scores aus dieser Queue entfernt werden. Dieser A*-Score eines Knotens n wird durch die Evaluationsfunktion $f(n) = g(n) + h(n)$ berechnet. Dabei ist h die Heuristikfunktion (etwa die Manhatten-Distanz) und g die Summe der Kosten vom Startzustand bis zum entsprechen Knoten.*

Diese Evaluationsfunktion wurde so gewählt, dass A* unter bestimmten Voraussetzungen mit Sicherheit eine optimale Lösung findet. Die erste Bedingung dafür ist, dass die Heuristikfunktion h admissible ist.[11] Dies garantiert, dass der erste Knoten der vom Frontier runtergeholt wird und den Endzustand enthält auch der Knoten ist, der den optimalen Pfad vom Anfangszustand zum Endzustand abschließt. Diese Bedingung ist hinreichend für die Optimalität von *A*-Tree-Search*. Diese Implementation von A* benutzt kein Closed-Set, womit die gleichen Zustände öfters aus dem Frontier geholt werden können (und zum Teil auch müssen). Denn Admissibility ist nicht hinreichend, um zu garantieren, dass auch der erste Zwischenknoten, welcher aus dem Frontier geholt wird und einen bestimmten Zustand enthält, auch der Knoten ist, welcher den optimalen Weg vom Anfangszustand zum Zustand dieses Knotens repräsentiert. Eine Implementation mit Closed-Set würde dies falscherweise annehmen und somit trotz Admissibility zu einer nicht optimalen Lösung führen.

Dementsprechend ist die Optimalitätsbedingung für die sogenannte *A*-Graph-Search* Implementation (mit Closed-Set) die Konsistenz dieser Heuristik (welche schließlich auch die Admissibility impliziert).[11] Da die hier verwendete Heuristik (Summe der Manhatten-Distanzen) konsistent ist, wird eine Implementation mit Closed-Set vorgenommen. Es folgt eine Darstellung des Algorithmus in Pseudocode. Eine wichtige Änderung die im Vergleich zu BFS vorgenommen werden muss, ist, dass Knoten erst dann zum Closed-Set hinzugefügt werden, wenn diese von dem Frontier runtergeholt wurden.

* Es wichtig zu beachten, dass ein Knoten des Frontiers zwar einen bestimmten Zustand aus dem Zustandsgraphen enthält, der Knoten jedoch genau genommen repräsentativ für einen ganzen Weg durch den Graphen steht. Nämlich ein Weg von Anfangszustand bis zum Zustand des Knotens. Es können viele verschiedene solcher Wege im Frontier sein.

Algorithm 10 A*

```

1: function A*( $s, g$ )
2:   frontier  $\leftarrow$  PriorityQueue{ $(s : 0)$ }
3:   parent  $\leftarrow$  Map{ $\{\}$ }
4:   repeat
5:      $v \leftarrow$  frontier.dequeue()
6:     if  $\exists w \in V : (v, w) \in \text{parent}$  then
7:       Continue Loop
8:     end if
9:     parent  $\leftarrow$  parent  $\cup$  { $(v, v.parent)$ }
10:    if  $v = g$  then
11:      return SUCCESS
12:    end if
13:    for all  $(v, u) \in E$  do
14:      frontier.enqueue( $u : g(v) + C(v, u) + h(u)$ )
15:    end for
16:    until frontier  $= \emptyset$ 
17:    return FAILURE
18: end function

```

Leider leidet A* an dem gleichen Problem wie BFS: Das Frontier wächst exponentiell bezüglich der Suchtiefe. Dementsprechend ist auch die Speicherkomplexität von A* exponentiell (wenn auch mit einem besseren effektiven Branching-Faktor $b^* < b$). Es wird daher wie auch beim BFS ein alternativer Ansatz gewählt, welcher auf Iterative Deepening basiert.

6.6 ITERATIVE DEEPENING A* (IDA*)

Die Iterative Deepening Version vom A*-Algorithmus (IDA*) weist zwar ähnliche Eigenschaften auf wie der A*-Algorithmus (etwa die nötige Admissibility der Heuristik für die Optimalität), programmiert sich aber eher wie ein DFS. Der hauptsächliche Unterschied zum IDDFS besteht in der Art und Weise, wie die Rekursion eingeschränkt wird. Während beim IDDFS alle Knoten bis zu einer bestimmten Tiefe betrachtet wurden, wird beim IDA* die Suche durch einen maximalen A*-Score (f -Wert) eingeschränkt. Dabei startet der IDA* die Suche mit dem f -Wert des Startknotens. Da die g -Kosten am Startknoten 0 sind ist $f(s) = h(s)$. Der maximale f -Wert der dann bei der nächsten Suche verwendet wird, ergibt sich aus der vorherigen Suche. Dabei wird der minimale f -Wert aller f -Werte genommen, welche bei der vorherigen Suche die Rekursion abgebrochen haben (weil diese f -Werte größer als das vorherige Maximum waren).[11]

Algorithm 11 Iterative Deepening A* (IDA*)

```

1: function IDA*-SEARCH( $v, goal, g_{cost}, f_{max}, parent$ )
2:    $f \leftarrow g_{cost} + h(v)$ 
3:   if  $f > f_{max}$  then
4:     return  $f$ 
5:   end if
6:   if  $v = goal$  then
7:     return SUCCESS
8:   end if
9:    $f_{min} \leftarrow \infty$ 
10:  for all  $(v, u) \in E$  do
11:    if  $\forall w \in V : (u, w) \notin parent$  then
12:       $parent \leftarrow parent \cup \{(u, v)\}$ 
13:       $f_{min} \leftarrow \min \begin{cases} f_{min} \\ IDA^*-\text{Search}(u, goal, g_{cost} + C(v, u), f_{max}, parent) \end{cases}$ 
14:      if  $f_{min} = \text{SUCCESS}$  then
15:        return SUCCESS
16:      end if
17:       $parent \leftarrow parent \setminus \{(u, v)\}$ 
18:    end if
19:  end for
20:  return  $f_{min}$ 
21: end function
22: function IDA*( $s, g$ )
23:    $parent \leftarrow Map\{(s, \emptyset)\}$ 
24:    $f_{max} \leftarrow h(s)$ 
25:   while  $f_{max} < \infty$  do
26:      $f_{max} \leftarrow IDA^*-\text{Search}(s, g, 0, f_{max}, parent)$ 
27:     if  $f_{max} = \text{SUCCESS}$  then
28:       return SUCCESS
29:     end if
30:   end while
31:   return FAILURE
32: end function

```

Ähnlich wie beim IDDFS-Algorithmus ist die Speicherkomplexität auch beim IDA* lediglich linear in der Rekursionstiefe, womit der IDA*-Algorithmus einen guten Kompromiss zwischen Laufzeit- (langsamer als ein normaler A* aufgrund von Iterative Deepening, aber auch nur um einen konstanten Faktor) und Speicherverhalten trifft.

6.7 ERGEBNIS

Die besprochenen Algorithmen wurden zunächst an zufälligen (lösbar den quadratischen Puzzles der Größen 3×3 und 4×4 getestet. Zwar war das 3×3 Puzzle lösbar durch BFS, kompliziertere Anfangszustände haben jedoch den Speicherverbrauch des Algorithmus auf bis zu 2 GB ansteigen lassen (man beachte, dass jeder Knoten im Frontier oder im Open-Set den kompletten

Zustand eines Schiebepuzzles repräsentiert). Es hat sich tatsächlich gezeigt, dass IDDFS nicht sonderlich viel länger gebraucht hat als BFS zur Bestimmung einer Lösung. Das Lösen des 4×4 Puzzles mit den uninformierten Methoden hat sich als reine Glückssache rausgestellt (sofern das Puzzle nicht sehr einfach war, haben beide Verfahren in einem Zeitrahmen von mehreren Minuten keine Lösung gefunden). An dieser Stelle wurde erfolgreich der A*-Algorithmus eingesetzt, welcher zwar bei komplizierten Problemen schnell mehrere Gigabyte an Speicher benötigt hat, aber die meisten 4×4 Probleme in akzeptabler Zeit lösen konnte. IDA* ist dann in der Lage alle 4×4 Probleme zu lösen ohne dabei Unmengen an Arbeitsspeicher zu benötigen. Auch hier hat IDA* Probleme nicht viel langsamer gelöst als der normale A*. Das größte Problem war mit dem IDA* (und der Manhattan-Distanz als Metrik) im Verlauf des Projektes erfolgreich gelöst wurde, war ein 4×5 Puzzle. Dies war jedoch nicht mit allen Anfangszuständen reproduzierbar. Die Lösung eines 5×5 Puzzle steht damit an dieser Stelle außer Frage. Es sei jedoch erwähnt, dass anspruchsvollere Heuristikfunktionen und Verfahren, welche bekannte Teillösungen mit in die Heuristik einbeziehen (*Pattern-Datenbanken*) es ermöglichen, auch das 5×5 Puzzle optimal zu lösen.[21]

7

FAZIT & AUSBLICK

Rawr X3 *nuzzles* How are you? *pounces on you* you're so warm o3o *notices you have a bulge* someone's happy! *nuzzles your necky wecky* murr hehe ;) *rubbles your bulgy wolgy* you're so big! *rubbles more on your bulgy wolgy* it doesn't stop growing ///. *kisses you and licks your neck* daddy likes ;) *nuzzle wuzzle* I hope daddy likes *wiggles butt and squirms* I wanna see your big daddy meat! *wiggles butt* I have a little itch o3o *wags tails* can you please get my itch? *put paws on your chest* nyea it's a seven inch itch *rubs your chest* can you pwease? *squirms* pwetty pwease? :(I need to be punished *runs paws down your chest and bites lip* like, I need to be punished really good *paws on your bulge as I lick my lips* I'm getting thirsty. I could go for some milk *unbuttons your pants as my eyes glow* you smell so musky ;) *licks shaft* mmmmmmmmmmmmmmmmm so musky ;) *drools all over your cawk* your daddy meat. I like. Mister fuzzy balls. *puts snout on balls and inhales deeply* oh my gawd. I'm so hard *rubbles your bulgy wolgy* *licks balls* punish me daddy nyea *squirms more and wiggles butt* I9/11 lovewas an yourinside muskyjob goodness *bites lip* please punish me *licks lips* nyea *suckles on your tip* so good *licks pre off your cock* salty goodness *eyes roll back and goes balls deep*

LITERATURVERZEICHNIS

- [1] About opencv. Website, . Online erhältlich unter <https://opencv.org/about/>; abgerufen am 16. September 2019.
- [2] Opencv: cv::mat class reference. Website, . Online erhältlich unter https://docs.opencv.org/master/d3/d63/classcv_1_1Mat.html; abgerufen am 17. September 2019.
- [3] Opencv: Introduction. Website, . Online erhältlich unter <https://docs.opencv.org/master/d1/dfb/intro.html>; abgerufen am 19. September 2019.
- [4] Opencv: Hardware acceleration layer - interface. Website, . Online erhältlich unter https://docs.opencv.org/master/d1/d1b/group_core_hal_interface.html; abgerufen am 20. September 2019.
- [5] Opencv: Load, modify, and save an image. Website, . Online erhältlich unter https://docs.opencv.org/master/db/d64/tutorial_load_save_image.html; abgerufen am 23. September 2019.
- [6] Opencv: Geometric image transformations. Website, . Online erhältlich unter https://docs.opencv.org/master/da/d54/group_imgproc_transform.html; abgerufen am 26. September 2019.
- [7] Opencv: Operation on arrays - calcCovarMatrix. Website, . Online erhältlich unter https://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html#calcCovarMatrix; abgerufen am 23. Oktober 2019.
- [8] Naif Alajlan. Solving square jigsaw puzzles using dynamic programming and the hungarian procedure. *American Journal of Applied Sciences*, 2009.
- [9] Michael J. Fischer Bernard A. Galler. An improved equivalence algorithm. *Communications of the ACM*, 1964.
- [10] Jim Andress Cris Zanoci. Making puzzles less puzzling: An automatic jigsaw puzzle solver. 2016.
- [11] Alan K. Mackworth David L. Poole. Artificial intelligence: Foundations of computational agents. 2017.
- [12] Stephane Durocher Debajyoti Mondal, Yang Wang. Robust solvers for square jigsaw puzzles.
- [13] Ohad Ben-Shahar Dolev Pomeranz, Michal Shemesh. A fully automated greedy square jigsaw puzzle solver. *IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, 2011.

- [14] Nathan S. Netanyahu Dror Sholomon, Omid David. A genetic algorithm-based solver for very large jigsaw puzzles. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2013.
- [15] Martin L. Demaine Erik D. Demaine. Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. 2007.
- [16] Andrew C. Gallagher. Jigsaw puzzles with pieces of unknown orientation. 2012.
- [17] Ayellet Tal Genady Paikin. Solving multiple square jigsaw puzzles with missing pieces.
- [18] J. D. Ullman J. E. Hopcroft. Set merging algorithms. *SIAM Journal on Computing*, 1973.
- [19] Dic Sonneveld Jerry Slocum. *The 15 Puzzle Book: How it Drove the World Crazy*. Slocum Puzzle Foundation, 2006.
- [20] David B. Cooper Kilho Son, James Hays. Solving square jigsaw puzzles with loop constraints. 2013.
- [21] Richard E. Korf. Recent progress in the design and analysis of admissible heuristic functions. 2000.
- [22] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 1956.
- [23] M. Saks M. Fredman. The cell probe complexity of dynamic data structures. *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, 1989.
- [24] Roey Nagar Nadav Erell. Robotic jigsaw puzzle solver. 2012.
- [25] Yakov Isidorovich Perelman. *Fun with Maths and Physics : Brain Teasers Tricks Illusions*. MIR Publishers (Moscow), 1988.
- [26] Jan van Leeuwen Robert Endre Tarjan. Worst-case analysis of set union algorithms. *Journal of the ACM*, 1984.
- [27] Kevin Wayne Robert Sedgewick. Algorithms 4th edition: 2.5 sorting applications. pages 336–360, 2011.
- [28] Lourdes Agapito Rui Yu, Chris Russell. Solving jigsaw puzzles with linear programming. 2015.
- [29] William T. Freeman Taeg Sang Cho, Shai Avidan. A probabilistic image jigsaw puzzle solver. *Computer Vision and Pattern Recognition (CVPR)*, 2010.
- [30] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 1975.

- [31] Robert Endre Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 1979.
- [32] Eric W. Weisstein. Permutation inversion. Website. From MathWorld—A Wolfram Web Resource. Online erhältlich unter <http://mathworld.wolfram.com/PermutationInversion.html>; abgerufen am 16. September 2019.
- [33] William E. Story Wm. Woolsey Johnson. Notes on the "15" puzzle. *American Journal of Mathematics*, 1879.