

PRAXISPHASE – SS 2019

NICO PISTEL

BERICHT

**PRAXISPHASENBERICHT**

**Classcon Consulting GmbH**

**D.VELOP AG**

**NICO PISTEL**

**Sommersemester 2019**

**WESTFÄLISCHE HOCHSCHULE  
WIRTSCHAFT UND INFORMATIONSTECHNIK**

## ABSTRACT

---

Nope.

## INHALTSVERZEICHNIS

---

<b>1 EINLEITUNG</b>	<b>1</b>
1.1 Problemstellung und Zielsetzung	2
1.2 Aufbau der Arbeit	2
<b>2 OPENCV GRUNDLAGEN</b>	<b>4</b>
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
<b>3 METRIKEN ZUR BILDREKONSTRUKTION</b>	<b>14</b>
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)	25

## 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	Distanzberechnung bei direkter Zuordnung der Pixel	24
Abbildung 3.6	Backtracking im DTW-Algorithmus	24
Abbildung 3.7	Distanzberechnung bei direkter Zuordnung der Pixel	25

## TABELLENVERZEICHNIS

---

Tabelle 3.1 Räumlicher Zusammenhang zweier Puzzlestücke 18

## LISTINGS

---

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

# 1

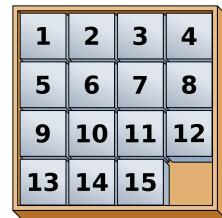
## EINLEITUNG

---

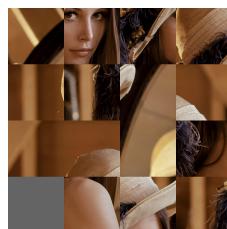
Schiebepuzzles wie das 15-Puzzle haben zum Ende des 19. Jahrhunderts das Interesse vieler amerikanischer Puzzle-Enthusiasten geweckt[12, 10] Heutzutage gibt es solche Schiebepuzzles in vielen Varianten. Dazu gehört die weitverbreitete Variante, bei der es nicht das Ziel ist, Zahlen aufsteigend zu sortieren, 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  $256 \times 256$  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  $256 \times 30$  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  $20 \times 20$  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.

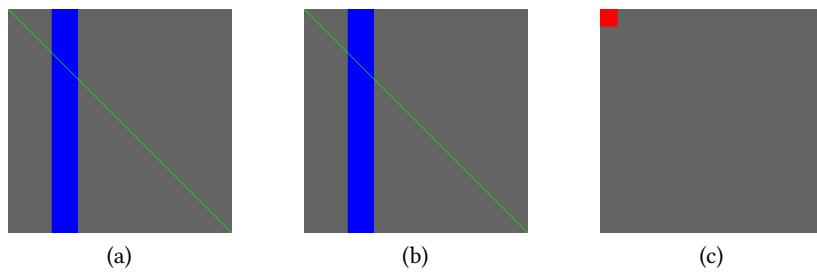


Abbildung 2.1: Ausgabe vom Mat und ROI Beispiel

## 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  $2 \times 3$  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  $90^\circ$  (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  $2 \times 3$  Matrix benötigt, wird die letzte Matrix auch als  $2 \times 3$  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  $2 \times 3$  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  $6 \times 6$  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.

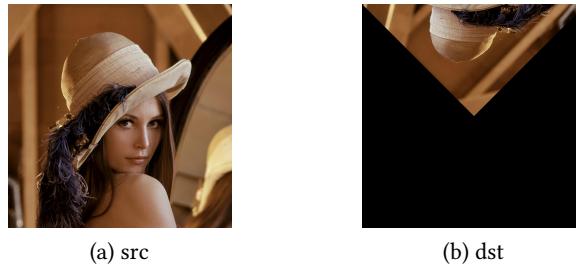


Abbildung 2.3: Abbildung zweier Dreiecke als affine Transformation

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  $3 \times 3$  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 Puzzle 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 [9] 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*[11] 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 Anordnung 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.

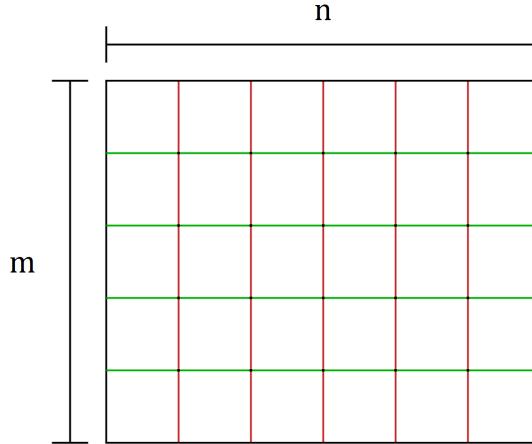


Abbildung 3.2: Anzahl der Kanten bei einem Type 3 Puzzle

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  $8 \times 8$  Type 1 Puzzle.

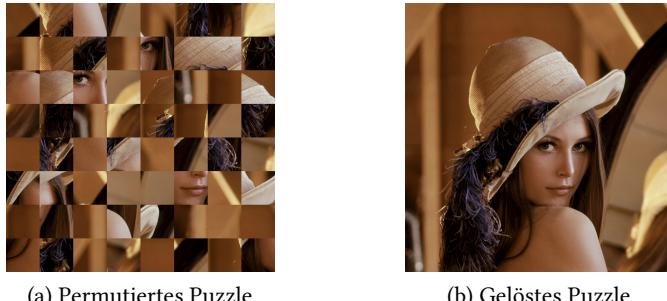


Abbildung 3.3: Beispiel eines Type 1 Puzzles

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 [8] 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 [8] wurde jedoch für Jigsaw Puzzles mit der  $(L_p)^q$  Norm eine weitere (allgemeinere) Norm eingeführt. Die  $(L_p)^q$  Norm berechnet sich zu

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

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 [8] (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 [8] 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:

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

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.

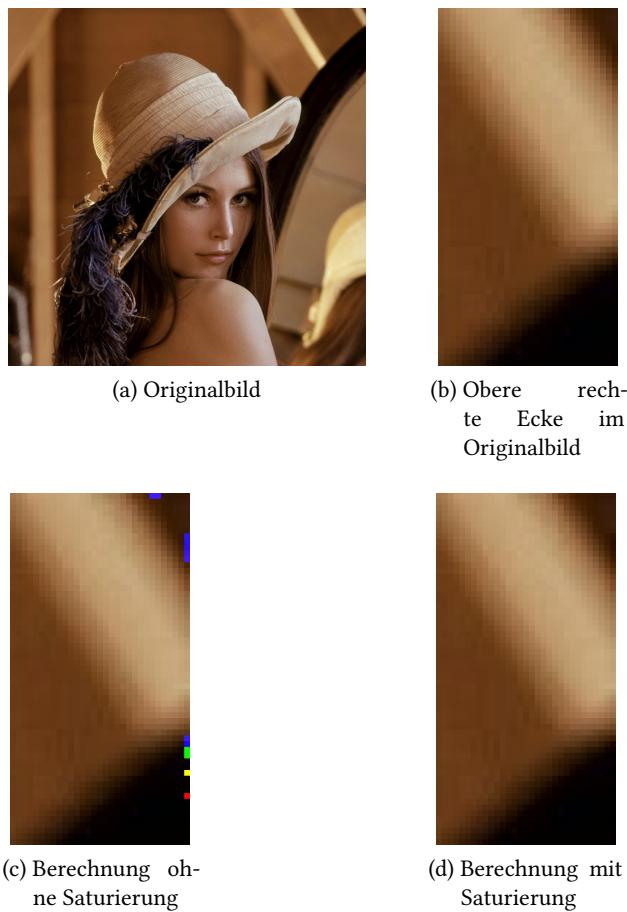


Abbildung 3.4: Lineare Approximierung der Pixel am rechten und am oberen Bildrand

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 [7] 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  $8 \times 1$  Bildränder  $\mathcal{A}$  und  $\mathcal{B}$ , welche 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 Distanzen eine Gesamtdistanz von 2246.

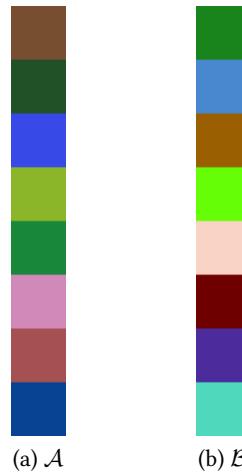


Abbildung 3.5: Distanzberechnung bei direkter Zuordnung der Pixel

Bitte töte mich

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

Abbildung 3.6: Backtracking im DTW-Algorithmus

Bro please kill me lol

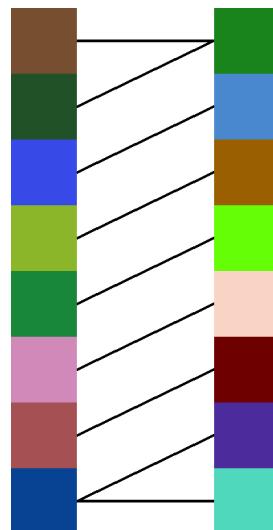


Abbildung 3.7: Distanzberechnung bei direkter Zuordnung der Pixel

### 3.5 MAHALANOBIS GRADIENT COMPATIBILITY (MGC)

aaaaaaaaaaaaaaaaaaaaaaa

## 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](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](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](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](https://docs.opencv.org/master/da/d54/group_imgproc_transform.html); abgerufen am 26. September 2019.
- [7] Naif Alajlan. Solving square jigsaw puzzles using dynamic programming and the hungarian procedure. *American Journal of Applied Sciences*, 2009.
- [8] 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.
- [9] Andrew C. Gallagher. Jigsaw puzzles with pieces of unknown orientation. 2012.
- [10] Dic Sonneveld Jerry Slocum. *The 15 Puzzle Book: How it Drove the World Crazy*. Slocum Puzzle Foundation, 2006.
- [11] David B. Cooper Kilho Son, James Hays. Solving square jigsaw puzzles with loop constraints. 2013.
- [12] Yakov Isidorovich Perelman. *Fun with Maths and Physics : Brain Teasers Tricks Illusions*. MIR Publishers (Moscow), 1988.