Technische Universität München
Fakultät für Informatik,
Lehrstuhl 11
Angewandte Informatik / Kooperative Systeme

Kompression großer Bilddatenbanken mittels Ähnlichkeitsgruppierung

Matthias Kramm

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat) genehmigten Dissertation.

Vorsitzende:	Univ.-Prof. Gudrun J. Klinker, Ph.D.
Prüfer der Dissertation:	Univ.-Prof. Dr. Johann Schlichter
	Univ.-Prof. Dr. Uwe M. Borghoff, Universität der Bundeswehr München

Die Dissertation wurde am 12.03.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30.06.2008 angenommen.

Kurzfassung

Heutige Bilddatenbanken bestehen ohne weiteres aus mehreren hunderttausend Bildern, und bewegen sich bezüglich Speicherplatzanforderungen im Terabyte Bereich, auch wenn sie mit aktuellen Bildkompressionsverfahren arbeiten. In dieser Arbeit wird gezeigt, wie sich bestehende Redundanzen zwischen verschiedenen, aber ähnlichen Bildern in so einer Bilddatenbank gewinnbringend auszunutzen lassen, um Bildkompressionsraten zu erhöhen.
Hierzu werden eine Reihe von auf Einzelbildern arbeitende Verfahren (fraktale Kompression, Primärkomponentenanalyse, Textursegmentierung, zeilenbasierte Pixelvorhersagen) auf Bildmengen erweitert. Diese neuen Verfahren komprimieren ähnliche Bilder mit mehr Effizienz als bei der Einzelbildkompression.
Um eine bestehende Bilddatenbank in Gruppen zu zerlegen, auf denen man die genannten Verfahren möglichst gewinnbringend einsetzen kann, werden eine Reihe von Clustering-Algorithmen und -Metriken daraufhin untersucht, inwieweit sie mit den aus den Datenkompressionsansätzen gewonnenen Statistiken zusammenarbeiten und eine gewinnbringende Zerlegung der Datenbank in Gruppen von ähnlichen Bildern liefern.

Abstract

Image databases today easily contain several hundreds of thousands of images, and use several terabytes of disk space, even when tightly compressed with state-of-the-art algorithms.
In this thesis, it is shown how redundancies between different but similar images in such image databases can be exploited to enhance compression rates. In order to do that, a number of image compression algorithms which usually only work on single images (fractal compression, principal component analysis, texture segmentation, line-wise predictors) are extended to versions which work on image groups. The new algorithms are able to compress images better than their single-image variants if the images are similar enough. In order to find such groups of similar images, a number of clustering strategies are evaluated in order to estimate to which degree they are able to partition an existing image database into groups such that the new compression algorithms are able to accomplish the maximum of compression.

Inhaltsverzeichnis

1 Einleitung **7**

1.1 Motivation 7

1.2 Überblick 9

2 Bild/Video Kompression — Stand der Dinge **10**

2.1 Gängige Algorithmen bzw. Formate (verlustlos) 10

2.1.1 Geschichte 10

2.1.2 GIF / LZW 11

2.1.3 LZ77 / PNG sowie Prediktoren/Preprocessing 12

2.1.4 RLE / BMP 16

2.1.5 CCITT / TIFF 16

2.1.6 LOCO-I / Verlustloses JPEG 17

2.1.7 Verlustloses JPEG-2000 18

2.2 Gängige Algorithmen bzw. Formate (verlustbehaftet) 19

2.2.1 Geschichte 19

2.2.2 Transformations-basierte Algorithmen 20

2.2.3 Fourier / JPEG 24

2.2.4 Wavelets / JPEG2000 25

2.2.5 Motion-Estimation 26
2.3 Experimentelle bzw. (noch) nicht etablierte Algorithmen . . . 28
2.3.1 Karhunen-Loeve / PCA 29
2.3.2 Steerable Filters 30
2.3.3 Textons 30
2.3.4 Fraktale Kompression 36
2.3.5 Texturkompression 39
2.3.6 Polygonalisierung 42
2.3.7 Modellbasierte Kompression 45
2.4 Zusammenfassung der besprochenen Algorithmen 49

3 Kompression von Bildmengen 50

3.1 Fraktale Bildmengenkompression 52
3.1.1 Erweiterung ins Dreidimensionale 52
3.1.2 Einführung von Kreuzverweisen zwischen Bildern . . . 53
3.1.3 Algorithmus 54
3.1.4 Abschätzung der fraktalen Komprimierbarkeit 59
3.2 Videocodecs 64
3.3 Karhunen-Loeve auf Bildmengen 69
3.3.1 Kompressionsdistanz von KLT 72
3.4 PNG auf Bildmengen 76
3.4.1 Implementierung 79
3.4.2 Distanz-Relation zwischen Scanlines 80
3.5 Kompression von Bildmengen mittels Textursynthese und Segmentierung 83

3.5.1 Texturanalyse und -synthese 85

3.5.2 Segmentierung 88

3.5.3 Distanz-Relation zwischen Texturen 90

3.5.4 Exkurs: Nachkompression von JPEG Bildern mittels Texturen 91

3.6 Eigenschaften der neuen Kompressionsalgorithmen 95

4 Kompressionsoptimierung durch Bildgruppensuche 96

4.1 Kompressions-Distanzmetriken 97

4.2 Klassische Distanz-Metriken zwischen Bildern 100

4.2.1 Featureextraktion 101

4.2.2 Vergleich von Features 107

4.2.3 Approximation von Kompressionsdistanz mittels Bildähnlichkeitsfeatures 108

4.3 Clustering Algorithmen 110

4.3.1 Kleinster Spannbaum (MST) 110

4.3.2 SAHN 111

4.3.3 Greedy 114

4.3.4 nCut 114

4.3.5 Multidimensional Scaling 117

4.3.6 k-Means 119

4.3.7 MeanShift 120

4.3.8 Genetische Algorithmen 123

4.3.9 Perfektes Clustering 125

4.4 Bestimmung der Qualität von Clusteringalgorithmen 127

4.4.1 Teilmengen 128
4.4.2 Token-Sets 129
4.4.3 zlib auf Textbausteinen 130
4.5 Vergleich der Clustering-Algorithmen 133

5 Praxisbeispiel: Integration der neuen Algorithmen 135
5.1 Die mediaTUM Bilddatenbank der TU-München 135
5.1.1 Architektur 136
5.1.2 Datenmodell 136
5.1.3 Features 138
5.2 Datenbestand in mediaTUM 138

6 Zusammenfassung 142

7 Ausblick 144

Abbildungsverzeichnis 146

Index 150

Literaturverzeichnis 153

Kapitel 1

Einleitung

1.1 Motivation

Datenkompression ist eine der ältesten Disziplinen der Informatik. Bereits 1835 erfand Samuel Morse den nach ihm benannten Morse-Code [11], einen Code, der öfter benutzte Zeichen in natürlichsprachigen Texten mit wenigen „Bits“ (Punkten und Strichen) kodierte, und seltener benutzte mit dementsprechend mehr. Eine einfache „Variante“ des eigentlich erst 1952 erfundenen Huffman Kompressionsalgorithmus.

Heute hat die Datenkompression in fast alle Bereichen der Technik, insbesondere elektronische Audio-/Video- und Bildverarbeitung und -archivierung, Einzug erhalten, und ist dort nicht mehr wegzudenken.

Diese Arbeit beschäftigt sich mit dem Einsatz von gängigen und neuen Bildkompressionsalgorithmen im Bereich großer Bilddatenbanken. Dabei wird untersucht, inwieweit sich auch Redundanzen zwischen ähnlichen, aber verschiedenen Bildern (siehe z.B. Abbildung 1.1) ausnutzen lassen, um die Kompressionsrate zu erhöhen.

Die Idee hierbei ist, dass bisherige Bildkompressionsalgorithmen bis auf wenige spezielle Ausnahmen[1] nur die Redundanzen innerhalb ein und desselben Bildes speichern. Oftmals sind aber auch, gerade in großen Datenbanken

[1] Sowohl die Videokompression als auch die multispektrale Kompression behandelt das Komprimieren von stark korrelierten Bilddateien der gleichen Dimension. Dies ist ein Spezialfall, der für Bilddatenbanken allerdings nur bedingt eingesetzt werden kann (siehe auch Seite 64).

Abbildung 1.1: *Zwei ähnliche Bilder aus der zentralen Bilddatenbank der Technischen Universität München*

mit vielen thematisch geordneten Bildern (Häuser, Landschaften, ...) sehr viele Korrelationen zwischen Bildern vorhanden, die von den gängigen Algorithmen bisher so gut wie nicht ausgenutzt werden. Große Bilddatenbanken bedürfen gleichzeitig aber auch sehr guter Kompressionsverfahren, da sie sehr viele und teils auch sehr große Bilder (z.B. gescannte Landkarten, medizinische Fotos, Fotoreihen etc.) speichern müssen.

Gleichzeitig ist es oft wünschenswert, Teile solcher Datenbanken in sehr kompakter Form extrahieren zu können, um sie auf einem relativ kleinen Datenträger, wie z.B. einer CD oder einer DVD zu speichern.

Dies stellt neue Anforderungen an die verwendeten Kompressionsverfahren, und ist damit ein guter Grund, sich eingehender mit der Ausnutzung von Redundanzen in Bildmengen zu beschäftigen.

1.2 Überblick

Zuallererst gibt Kapitel 2 einen Überblick über den bisherigen Stand und die Standards in Bild-, sowie davon abgeleitet, Videokompression.

Als nächstes werden in Kapitel 3 Erweiterungen dieser Verfahren, sowie die Bildmengen-Versionen von einigen neuen, bisher nicht in einem standardisierten Dateiformat verfügbaren Algorithmen (wie beispielsweise Polygonalisierung oder Textursynthese), vorgestellt.

Kapitel 4 betrachtet nun, daraus motiviert, ein verwandtes Problem: Welche Möglichkeiten gibt es, große Bildmengen so in Cluster zu zerlegen, dass das Endresultat möglichst gut mit einem der erweiterten Verfahren komprimierbar ist? Hierzu werden in Kapitel 4.2 die verschiedenen Ähnlichkeitsmaße zwischen Bildern, und deren Zusammenhang mit der Komprimierbarkeit von Bildmengen, studiert, und in Kapitel 4.3 gängige Clusteralgorithmen vorgestellt. In Kapitel 4.4 wird dann der Einsatz von verschiedenen dieser Verfahren mit Kompressionsalgorithmen bewertet.

Kapitel 5.1 zeigt abschließend dann auf, inwieweit die in den vorherigen Kapiteln vorgeschlagenen Algorithmen in eine reale Bilddatenbank integriert werden können.

Kapitel 2

Bild/Video Kompression — Stand der Dinge

Der Überblick über die verschiedenen aktuellen Kompressionsverfahren in diesem Kapitel erfolgt zweistufig: Zuerst werden etablierte Algorithmen betrachtet, die sich in der Praxis bewährt haben, und für die für eines standardisierten Dateiformats genug Implementationen zur Verfügung stehen. Danach werden Algorithmen vorgestellt, die sich noch in der Forschung befinden, oder die sich auf dem Softwaremarkt nicht durchsetzten konnten. Hier werden insbesondere solche betrachtet, die als Basis für einen auf Bildmengen arbeitenden Kompressionsalgorithmus verwendet werden können.

2.1 Gängige Algorithmen bzw. Formate (verlustlos)

2.1.1 Geschichte

Datenkompression (in ihrer verlustfreien Variante) wurde gewissermaßen mit der Schöpfung der Shannon'schen Informationstheorie (1940, von Claude Shannon) geboren, in der untersucht wird, wie sich Datenströme durch Reduzierung der Bitzahlen pro Zeichen verkürzt darstellen lassen.

Die darin begründete Idee, Daten statistisch zu modellieren sowie die den einzelnen Symbolen zugrundeliegenden Wahrscheinlichkeiten durch Bitlängen

zu repräsentieren, wurde 1952 von D. A. Huffman aufgegriffen und maßgeblich vorangetrieben. Huffman erfand die nach ihm benannte Huffman-Kodierung [40].

Fast 30 Jahre fand sämtliche Datenkompression durch eine derartige probabilistische Modellierung statt, d.h. durch eine Einzel-Zeichen Kompression ohne Kontext, oder, im Fall von verschachtelten Huffman-Bäumen, nur durch den Kontext der unmittelbar vorangegangenen n Zeichen.

Erst in den Jahren 1977/78 wurden von J. Ziv und A. Lempel zwei neue Algorithmen vorgestellt (LZ77 und LZ78) die nicht mehr (nur) auf diesem Modell der statistischer Modellierung basierten, sondern Daten durch einen Wörterbuch-Ansatz auf Basis von wiederkehrenden Zeichenketten komprimierten [112] [113]. Der Algorithmus LZ78 wurde 1984 durch Terry Welch weiter verfeinert [104], und bekam den Namen LZW.

Eine der ersten Implementierungen des LZW Algorithmus war das Unix-Programm „compress“, entwickelt 1984, das aufgrund guter Portabilität, Dokumentation und Performance schnell großflächig in der Unix-Welt eingesetzt wurde. Das Implementierungs-Äquivalent in der Welt der DOS-Computer, ARC, wurde erst 1985 von der Firma System Enhancement Associates präsentiert.

Durch die Programme gzip (Unix) sowie PKZIP, LHA und später RAR (DOS) wurden die Programme „compress“ und „ARC“ später abgelöst, und damit interessanterweise gleichzeitig auch der Algorithmus LZ78 bzw. LZW wieder durch seinen Vorgänger LZ77 ersetzt, der durch diverse Verfeinerungen besser optimiert werden konnte (und damit bessere Kompression lieferte sowie auch einfacher zu implementieren war) als sein großer Bruder LZ78.

Die Entwicklung zurück zu LZ77 wurde später noch weiter durch die Patent-Problematik rund um LZW vorrangetrieben, und bewirkte u.A., dass auch neue Bildformate (wie PNG, Kapitel 2.1.3) wieder von dem in älteren Bildformaten (GIF) eingesetzten LZW Codec zu LZ77 umschwenkten.

2.1.2 GIF / LZW

Das grundlegende Prinzip des Kompressionsalgorithmus LZW ist, dass implizit eine Zeichenbibliothek aufgebaut wird, auf die in jedem neuen Zeichen nur verwiesen werden muss. Je mehr Daten gelesen werden, desto größer wird die Zeichenbibliothek, und damit werden alle Daten, die statistisch ähnlich

den bisherigen sind, besser komprimiert. (Siehe Abbildung 2.1)

Das GIF Format speichert die Verweise auf die Zeichenbibliothek des Weiteren noch in einem Huffman-kodierten Format.

Für Bilder auf Webseiten war das GIF Format lange Standard, wenngleich es relativ wenig Formate und Farbräume beherrscht — nur Palettenindizierte Bilder mit 256 Farben werden unterstützt. Ein interessantes Feature von GIF sind allerdings Animationen, die durch Bereichsüberlagerungen sogar inkrementell komprimiert werden können. Gegenüber anderen gängigen Kompressionsformaten fehlt GIF aber insbesondere die Möglichkeit, Metadaten, Thumbnails sowie Bilder mit größeren Farbräumen abzulegen.

Da GIF gegenüber neueren Algorithmen wie beispielsweise PNG technisch zurücksteht, wird in den nachfolgenden Kapiteln dieses Format nicht weiter betrachtet werden.

2.1.3 LZ77 / PNG sowie Prediktoren/Preprocessing

Anders als beim LZW Algorithmus wird beim Vorgänger LZ77 nicht eine speziell angepasste Zeichenbibliothek bei der Kompression und Dekompression aufgebaut, sondern es wird direkt auf den Stream der bisher gelesenen Daten verwiesen.

Dies ist streng genommen ineffizienter als LZW, da im Datenstrom manche Zeichenketten mehrfach vorkommen können, und deswegen verschiedene Verweise auf diese Zeichenkette nicht in der kürzestmöglichen Form abgelegt werden (bei zwei gleichen Zeichenketten erzeugen zwei verschiedene Ausgabesymbole dieselbe Dekompression). Nichtsdestotrotz erzielen optimierte LZ77 Kompressionsverfahren (z.B. zlib [22]) sehr gute Resultate, was u.A. daran liegt, dass es einfacher ist, statistisch gute Kodierungen für die Verweise zu finden. Insbesondere sind Kovarianzen zwischen Rückverweislänge und Symbollänge kodierbar, was bei LZW nicht ohne weiteres möglich ist.

Das Format PNG, das den zlib Algorithmus und damit LZ77 zur Komprimierung benutzt, ist ein „tagged“ Dateiformat, d.h. das Datenfile besteht aus „Chunks“, (z.B. Header, Farbtabellen, Alphakanal-Daten, textuelle Informationen), die nicht zwangsläufig alle von einer Dekodierungsapplikation ausgewertet werden müssen, um eine PNG Datei dekomprimieren zu können. Zum Beispiel kann eine Applikation, die sich nicht für die Transparenzebenen interessiert, sämtliche Alphakanal-Tags weglassen.

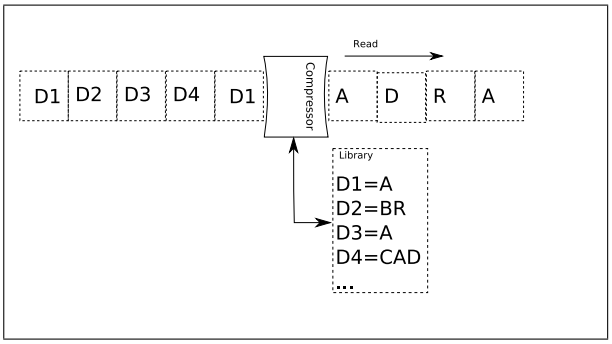


Abbildung 2.1: *Das Prinzip der LZ78 und LZW Verfahren — ein Dictionary wird verwaltet, welches durch jedes Eingabezeichen ergänzt wird. Der Ausgabestrom des Kompressors sind (im Fall von LZ77 und LZ78) Mischungen aus Zeichen und Dictionary-Einträgen bzw. (im Fall von LZW) nur Dictionary-Einträge.*

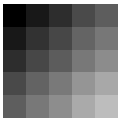
Zusätzlich zur eigentlichen Kompression verwendet PNG eine zeilenweise Umschaltung zwischen sogenannten Prediktoren, die versuchen, die Bildinformationen aus den vorangegangenen Pixeln vorherzusagen und nur die Differenzen zwischen dieser Vorhersage und dem tatsächlichen Wert zu speichern.

Ein Beispiel für Prediktoren ist in Bild 2.2 gegeben. Hier wird der Pixel in der rechten unteren Ecke vorhergesagt, entweder durch den in X- und Y- Richtung vorhergehenden, durch den Mittelwert der beiden, oder durch einen nicht-linearen Prediktor (Paeth-Prediktor) in 3 Richtungen. Nur die Differenz zwischen Vorhersagewert und eigentlichem Pixel wird gespeichert.

Diese Prediktor-basierte Kodierung verändert zwar nicht die Länge des Datenstroms, macht diesen aber besser komprimierbar.

Als Beispiel betrachte man das folgende 5×5 Bild eines Gradienten:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$



Wird dieses Bild durch einfache Prediktoren (Vorhersage durch den jeweils horizontal vorangehenden Pixel) abgelegt, so ergibt sich die folgende, leichter komprimierbare Version:

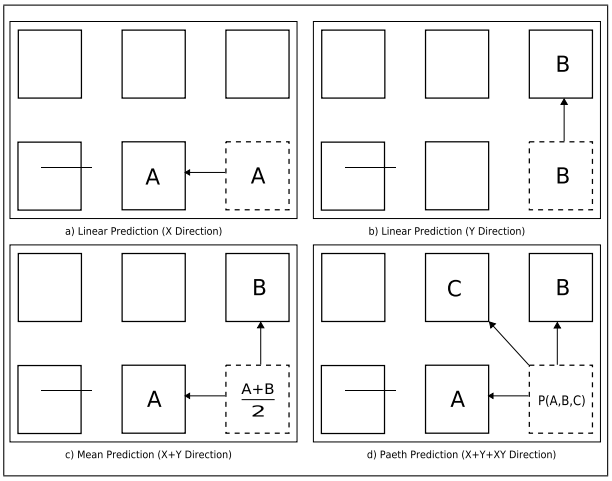


Abbildung 2.2: *Prediktoren in PNG Bildern*

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 & 1 \\ 3 & 1 & 1 & 1 & 1 \\ 4 & 1 & 1 & 1 & 1 \\ 5 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Um das ganze noch weiter zu illustrieren, ist in Bild 2.3 ein Bild und seine mit verschiedenen Methoden gefilterte Version zu sehen. Wie man sieht ist bei dem gefilterten Bild deutlich weniger Farbvarianz vorhanden, die Information konzentriert sich stärker auf die kontrastreichen Bereiche und macht die Daten dadurch besser komprimierbar.

Prediktoren machen bei den meisten Bildern einen Kompressionsunterschied von bis zu 50% aus, weshalb PNG sich momentan im Vergleich mit anderen verlustlosen Dateiformaten sehr gut schlägt. Nur bei sehr kleinen Bildern ist das (auf 256 Farben spezialisierte) GIF Format teilweise besser, bei Fotoähnlichen Bildern erzielen oftmals die verlustlosen Versionen von JPEG und JPEG2000 bessere Ergebnisse.

PNG unterstützt des Weiteren die Ablage von textuellen Informationen (Metadaten), die auch komprimiert gespeichert werden können. Erwähnenswert ist des Weiteren die Erweiterung MNG, die das verlustlose Speichern von Animationen erlaubt, mittels Mechanismen wie Sprites (bewegten Objekten) sowie lokalen Bildüberlagerungen [76].

Das Dateiformat PNG wird in Kapitel 3.4 nochmals aufgegriffen werden.

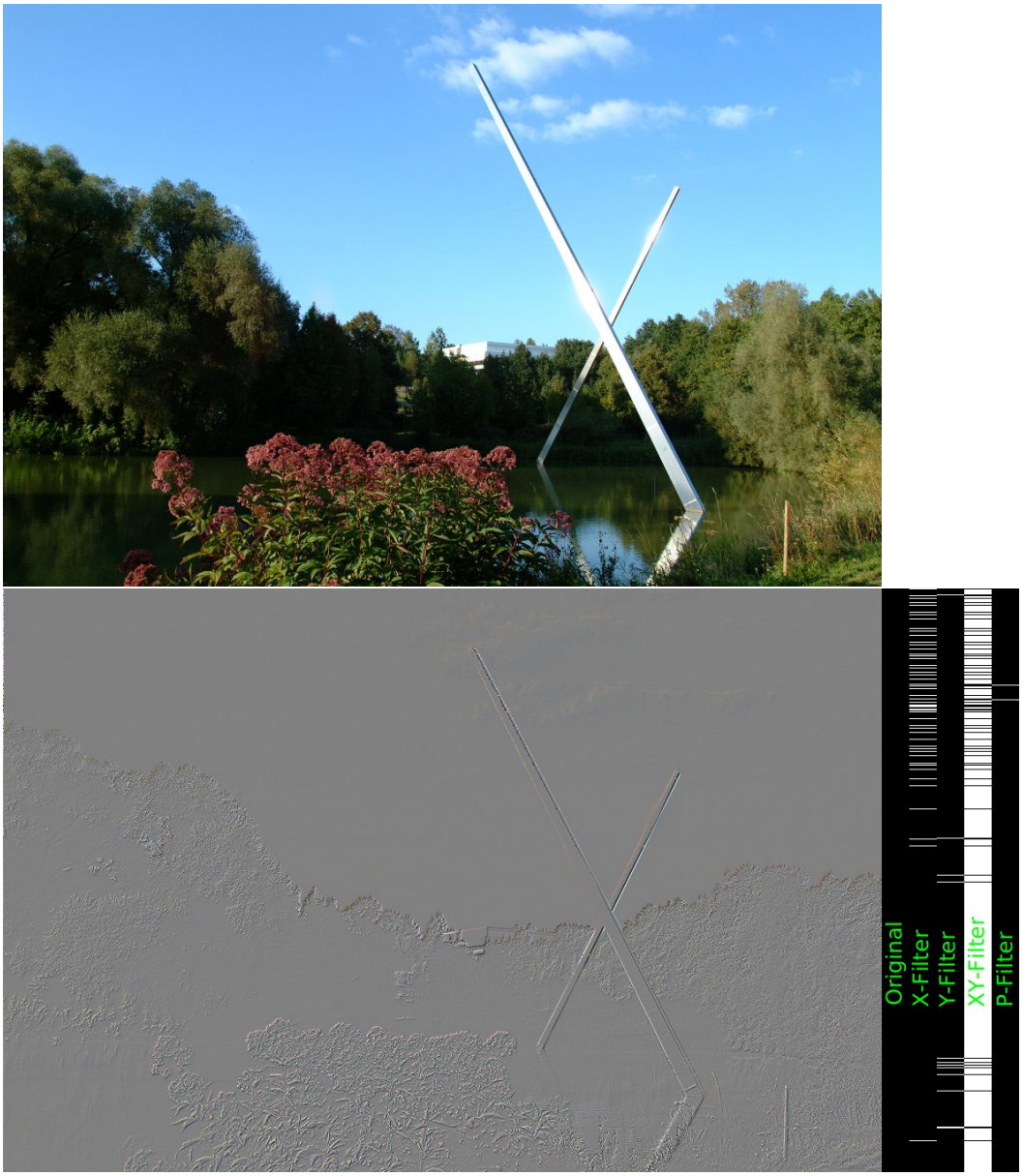


Abbildung 2.3: *Ein Bild, und sein PNG-gefiltertes Äquivalent — die Beschriftung auf der rechten Seite zeigt den in der jeweiligen Zeile gewählten Filtertyp an*

2.1.4 RLE / BMP

RLE ist ein sehr einfaches Kompressionsformat, dessen Geschichte nicht genau bekannt ist, das hier der Vollständigkeit halber aber kurz erwähnt werden soll: Es speichert Aneinanderreihungen von gleichen Bytes (d.h. z.B. Graustufenbilder mit monotonen Bereichen) komprimiert durch Längenangaben.

Dies ist eine Art Untermenge des LZ77 Algorithmus, der auch „implizit“ in der Lage ist, derartige Byteketten zu komprimieren — siehe Abbildung 2.4

Benutzt wird es u.A. im (sehr alten) BMP Format. BMP unterstützt auch Bilder im vollen RGB Farbraum, hat allerdings keine Unterstützung für Metadaten.

2.1.5 CCITT / TIFF

CCITT ist ein von Faxmaschinen importierter Algorithmus, der auf der Speicherung von Lauflängen nach dem RLE Prinzip (Group 3) sowie vertikalen Verweisen (Group 4) basiert. Durch dieses Verfahren können Schwarzweiß-Bilder sehr effizient komprimiert werden.

Einsatz findet CCITT außer auf Faxmaschinen noch im Containerformat TIFF. TIFF ist das mit Abstand komplexeste Format in der momentanen Welt der verlustlosen Bilddatenkompression, das insbesondere als einziges hier betrachtetes verlustloses Format mehr als einen Kompressionsalgorithmus unterstützt, und Farbräume außer RGB (YCbCr, L*a*b*, CMYK) ab-

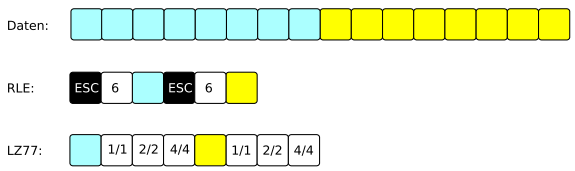


Abbildung 2.4: *Vergleich von LZ77 und RLE: RLE speichert ein sich wiederholendes Zeichen explizit mittels eines Escape Zeichens im Datenstrom, gefolgt vom Zeichen und einer Anzahl. LZ77 speichert erst das Zeichen, und darauf folgend Rückverweise (Länge 1, auf das Zeichen, danach Länge 2, auf zwei Zeichen, usw.)*

zubilden vermag.

TIFF Bilder können außer mittels RLE u.A. auch verlustbehaftet (mittels JPEG) komprimiert sein, ohne dass sich etwas an der Dateinamenendung (.tif) ändert.

Des Weiteren enthält TIFF einen verlustlosen LZW RGB Codec (für fotografische Bilder), der aber leider nur eine sehr mangelhafte Kompressionseffizienz erzielt, was u.A. an der (im Vergleich zu PNG) kleinen Anzahl an Präprozessor-Algorithmen liegt — nur der X-Prediktor wird unterstützt.

2.1.6 LOCO-I / Verlustloses JPEG

JPEG, welches in Kapitel 2.2 (über Verlustbehaftete Algorithmen) beschrieben wird, enthält auch einen verlustlosen Algorithmus, der allerdings relativ unbekannt ist und in der momentanen Softwarewelt nicht annähernd so oft eingesetzt wird wie sein JPEG-Gegenstück, die verlustbehaftete JPEG-Kompression.
Diese verlustlose Variante benutzt denselben JFIF-Container zur Speicherung des komprimierten Streams wie das „normale“ JPEG, verwendet dann allerdings intern den sogenannten LOCO-I Algorithmus [103].

Dieser Algorithmus besteht aus einem Vorhersageschritt (Prediction), der ähnlich zum Paeth-Prediktor des PNG Algorithmus arbeitet, sowie einer Huffman-Kodierung der Differenz zum Vorhersagewert. Des Weiteren werden die Lauflängen anhand eines vorberechneten Binärbaums (dem Golumb-Rice Baum [82]) abgespeichert.

Die Wahl dieser Kombination macht klar, dass verlustfreies JPEG insbesondere für die gute Kompression von Fotos und anderen natürlichen Bildern konzipiert wurde. Typische Redundanzen von künstlichen Bildern, wie z.B. Webgrafiken, (beispielsweise wiederholende Bildausschnitte) werden von LOCO-I nicht erfasst.
Eine Gemeinsamkeit zum „klassischen“ JPEG besteht darin, dass auch LOCO-I das Bild in 8×8 Blöcke zerlegt, um bestimmte Parameter für lokale Bildbereiche zu optimieren.

2.1.7 Verlustloses JPEG-2000

Auch das „neue JPEG“, JPEG 2000, enthält die Möglichkeit, Bilder verlustlos abzuspeichern. Bemerkenswert ist, dass bei der verlustlosen Speicherung genau die selben Algorithmen eingesetzt werden wie bei der verlustbehafteten. Einzig auf den Information zerstörenden Quantisierungsschritt wird verzichtet, und für die Wavelet-Transformation wird auf einfachere, umkehrbare, Filter umgeschaltet [1].

Bezüglich der Kompressionsrate ist JPEG-2000 in etwa vergleichbar [17] mit dem LOCO-I Algorithmus von JPEG, hat diesem gegenüber allerdings den Vorteil, dass kein eigener Kompressor benötigt wird, sondern die Implementierung analog zum „normalen“ JPEG-2000 funktioniert.

Eine etwas genauere Beschreibung der JPEG-2000 Funktionsweise findet sich im Kapitel 2.2.4

2.2 Gängige Algorithmen bzw. Formate (verlustbehaftet)

Die in den vorangegangenen Kapitel vorgestellten Methoden erreichen aufgrund ihrer Verlustlosigkeit in der Regel nur relativ niedrige Kompressionsraten von (typischerweise) bis zu 10 Bits pro Pixel. Benötigt man bessere Kompression, so ist man gezwungen, auf ein verlustbehaftetes Format zu wechseln.

2.2.1 Geschichte

Verlustbehaftete Kompression (auf Bilddateien) wurde durch die beginnende starke Verbreitung von Desktop-Computern in den 70er Jahren notwendig und sinnvoll. Da die bis dahin etablierten Verfahren der klassischen verlustlosen Kompression nicht mehr ausreichten, um die wachsende Zahl von Grafikdateien platzsparend abzulegen, wurde damals im Rahmen eines Zusammenschlusses zwischen den Standardisierungs-Gremien CCITT und ISO die Joint Photographics Expert Group geschaffen, die im Verlauf einiger Jahre das Format JPEG entwickelte. JPEG macht seit viele Jahren den de-facto Standard der verlustbehafteten Bildformate aus, und wird erst seit einigen Jahren teilweise durch die Neuentwicklung JPEG2000 abgelöst, die aus der Feder des selben Gremien-Zusammenschlusses stammt.

Die diesen Formaten zugrunde liegende Idee ist, dass Bilder nicht mehr 1:1 abgelegt werden, sondern es dem Kompressionsalgorithmus erlaubt wird, gezielt die Bilddaten in einer Art und Weise zu verändern, die dem menschlichem Beobachter möglichst nicht auffallen soll, und die die Daten gleichzeitig besser komprimierbar macht.
D.h. der neue, den verlustbehafteten Verfahren innewohnende Aspekt ist es, dass die Eigenheiten des menschlichen Sehvermögens mit in Erwägung gezogen werden. Daraus spross auch im Laufe der Jahre ein komplett eigener Forschungszweig, fußend auf Wertheimers „Gestalt“ Theorie aus dem Jahre 1936 [105]. Er analysiert die biologischen und psychologischen Aspekte des Sehens und leitet daraus Empfehlungen für präferierte Arbeitsweise von Bild-(kompressions-)algorithmen ab. [1]

[1] Nennenswert sind in diesem Kontext insbesondere auch die Arbeiten von Julesz [44] [46] [45], die sich mit der Wahrnehmung von regelmäßigen und quasiregelmäßigen Mustern, sog. Texturen beschäftigen. Mehr dazu in Kapitel 2.3.5.

2.2.2 Transformations-basierte Algorithmen

Das Prinzip der Transformations-basierten Kompression ist es, Quelldaten (Pixeldaten) des Bildes durch einen oder mehrere Transformationsschritte in ein Format zu bringen, von dem erwartet wird, dass es besser komprimierbar ist, und in dem Quantifizierungen dem menschlichen Beobachter weniger auffallen.

Die bessere Komprimierbarkeit der transformierten Daten wird in der Regel damit in Verbindung gebracht, dass die neuen Daten stärker dekorreliert sind. Durch die stärkere Dekorrelation sind die transformierten Daten unabhängiger, und Redundanz zwischen ihnen wird entfernt [82], dadurch sinkt die Varianz und damit der Wertebereich der abzuspeichernden Daten, wodurch sie durch weniger Informationseinheiten ausgedrückt werden können.

Auch wenn nicht alle dieser Verfahren momentan in der Praxis der Datenkompression eingesetzt werden (siehe auch Kapitel 2.3 über experimentelle und noch in der Forschung befindliche Verfahren), so sei hier doch bereits jetzt ein Überblick über die bekannten Transformationen in der Datenkompression (bzw. allgemeiner in der Bildverarbeitung) gegeben:

- Lineare Transformationen
  - Fourier (Kapitel 2.2.3)
  - Karhunen-Loeve (KLT), auch bekannt als Principal Component Analysis (PCA) (Kapitel 2.3.1),
- Filter-basierte Verfahren
  - Wavelets (Kapitel 2.2.4)
  - Steerable Pyramids (Kapitel 2.3.2)

Hierbei sind die Transformationen der ersten Klasse blockbasiert, und lassen sich über eine Matrix darstellen. Dazu werden die Bilder zuerst in quadratische $n \times n$ Blöcke geeigneter Größe geschnitten, und die resultierenden (aus n^2 Pixelwerten bestehenden) Vektoren dann über eine Matrizenoperation (mit einer $n^2 \times n^2$ Matrix A) transformiert. Teilweise (im Fall von Fourier-Transformationen) ist diese Transformation auch durch zwei Transformationen mit einer $n \times n$ Matrix darstellbar, und auch teilweise noch weiter optimierbar, wie z.B., wieder im Falle Fourier, durch die Fast-Fourier-Transformation (FFT).

Die Umkehrung (d.h. Dekompression) erfolgt dann durch Transformation mit der inversen Matrix; im Fall von orthonormalen Transformationen ist diese gleich der Transponierten.
Abbildung 2.5 zeigt ein Bild sowie eine in jedem 8×8 Block auf hohe oder niedrige Frequenzen quantisierte Version. Wie man sieht, sind die niedrigen Frequenzen deutlich entscheidender für die visuelle Qualität eines Bildes als die hohen.

Die Filter-basierten Verfahren der zweiten Klasse wiederum sind Pixel-basiert — die Transformation eines Bildes I funktioniert hier durch eine Konvolution

$$R_k = I \star f_k$$

mit einer Reihe von Filtern $f_k, k = 1 \ldots n$ mit anschließendem Subsampling von R_k.
Auch hier gibt es für bestimmte Wavelets bzw. Filter schnellere Implementierungen, z.B. „Lifting“ [92].

Die Umkehrung erfolgt durch umgekehrte Konvolution mit der Filter-Inversen (bei bestimmten Filtern, den sogenannten „tight frames“, ist dies der Filter selber [59]), und schrittweiser Vergrößerung der Auflösung sowie abschließender Addition aller Filter-Komponenten.

Filter- bzw. Wavelet-basierte Verfahren haben gegenüber Fouriertransformationen den Vorteil, dass sich die Koeffizienten immer nur auf ein relativ kleines Zeitfenster beziehen, und nicht - wie bei der Fouriertransformation - auf die gesamte zeitliche Achse. (Siehe Abbildung 2.6)
D.h. ein Wavelet-Koeffizient gibt sowohl Frequenz als auch Position an. Durch diese Eigenschaft nehmen Filter-basierte Verfahren nicht nur globale sondern auch lokale Informationen auf [12].

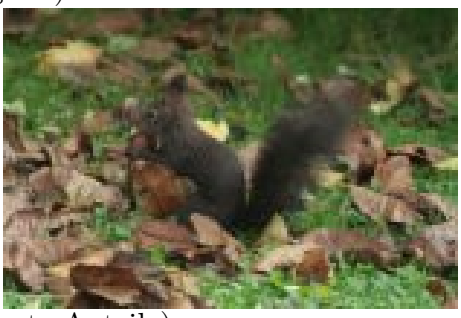
Abbildung 2.7 zeigt eine einfache Wavelet Transformation eines Bildes in die verschiedenen Wavelet-Anteile. An diesem Bild ist auch eine typische Eigenschaft von Wavelet-Transformationen sichtbar: Sie sind in der Regel hierarchisch, d.h. ein Bild einer bestimmten Größe wird stufenweise in eine Reihe von immer kleineren Bildern zerlegt.

In den folgenden beiden Kapiteln werden nun die beiden momentan etablierten Verfahren (JPEG und JPEG2000) beschrieben, die bisher Fourier- bzw. Wavelet-Transformationen einsetzen.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

(Original)

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



(Niederfrequente Anteile)

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

(Hochfrequente Anteile)

Abbildung 2.5: *Ein Bild (oben), bei dem einmal 48 der hohen Frequenzen (mitte), und einmal 48 der niedrigen Frequenzen (unten) entfernt (auf Null gesetzt) wurden. Der DC Koeffizient wurde bei beiden konstant gelassen. Auf der linken Seite ist die jeweilige Quantisierungsmatrix zu sehen.*

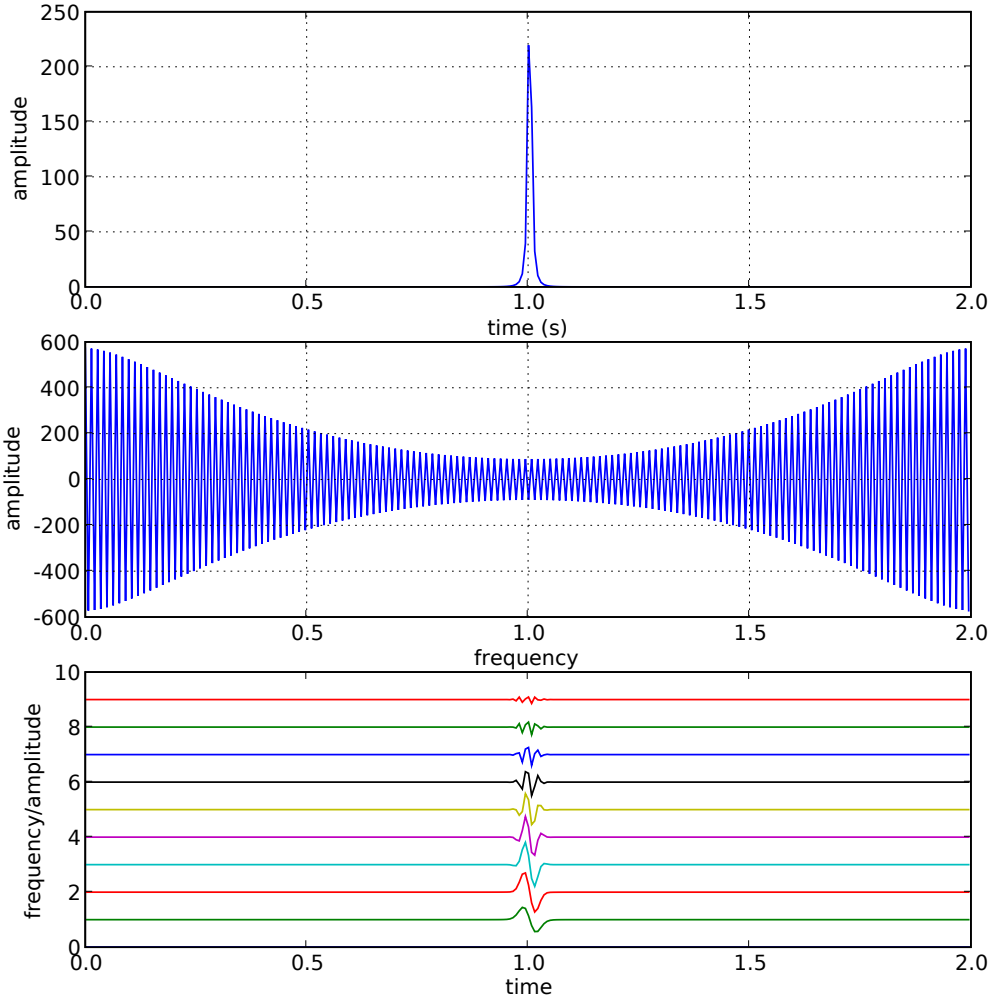


Abbildung 2.6: *Vergleich einer Fouriertransformation (mitte) eines zeitlich lokalen Signals (oben) mit der entsprechenden Gabor-Wavelettransformation (unten). Bei der Fouriertransformation wirken sich lokale Schwankungen auf das gesamte Spektrum aus, während bei der Gabortransformation nur ein kleiner Teilbereich der Koeffizienten betroffen ist.*

Abbildung 2.7: *Transformation eines Bildes (links) mittels 8 Stufen von Le Gall 5/3 Wavelets. Da das Ursprungsbild eine Größe von* $2^8 \times 2^8$ *Pixeln hat, ist das Residuum nur einen Pixel groß.*

2.2.3 Fourier / JPEG

Der JPEG Codec (üblicherweise eingefasst in das JFIF Dateiformat) benutzt die Block-basierte Fouriertransformation, um ein in 8×8 Bildblöcke zerlegtes Bild in leichter komprimierbare Koeffizienten zu überführen.

Hierzu wird jeder 8×8 Block $p_{ij}, i = 0 \ldots 7, j = 0 \ldots 7$ eines bestimmten Farbkanals (JPEG kann Bilder sowohl in RGB als auch in YUV oder CMYK Kanäle zerlegt kodieren) durch die diskrete Kosinustransformation

$$f_{ij} = \frac{1}{4} T(i) T(j) \sum_{m=0}^{7} \sum_{n=0}^{7} p_{ij} cos(\frac{\pi i(2m+1)}{16}) cos(\frac{\pi j(2n+1)}{16})$$

mit

$$T(x) = \begin{cases} \frac{1}{\sqrt{2}} & : \text{ für } x = 1, \\ 1 & : \text{ sonst} \end{cases}$$

in einen 8×8 großen Koeffizienten-Block $f_{ij}, i = 0 \ldots 7, j = 0 \ldots 7$ transformiert [101].

Dieser hat in der Regel die Eigenschaft, dass die Werte stärker voneinander dekorreliert sind als im Ursprungsbild , und sie sich deswegen besser komprimieren lassen. D.h. es findet eine Reduktion der „Energie“ der Ursprungsdaten statt [82]. Einer der Gründe dafür ist, dass die optimal dekorrelierende Transformation (KLT) von statistischen Daten gegen die DCT konvergiert — siehe Kapitel 3.3.

Im nächsten Schritt werden die Werte nun quantisiert (niederfrequente Anteile werden mit mehr Bits abgespeichert als hochfrequente), und durch Run-Length-Encoding (RLE) und Huffman (oder arithmetische Kodierung) in Bitstrukturen gepackt.
Das genannte RLE findet hier nur innerhalb der einzelnen Blöcke statt. Die einzige Koeffizientenkodierung, die sich auch Blockübergreifend beeinflusst (durch Differenzkodierung), ist die des Basiswertes (f_{00}) [72].

Durch das „Wegschneiden“ von hochfrequenten Anteilen innerhalb der 8×8 Blöcke erfolgt innerhalb dieser Blöcke eine für das menschliche Auge kaum wahrzunehmende Kompression (Siehe Bild 2.5), allerdings entstehen auch sog. „Blockartefakte“, d.h. vertikale und horizontale Konturlinien. Diese Artefakte sind einer der Hauptgründe, weshalb sich bei höheren Kompressionsraten eine sichtbare Verschlechterung des Bildes ergibt.
Ein weiteres JPEG typisches Artefakt ist „Ringing“ (das die künstliche Erzeugung von Schwingungen innerhalb eines 8×8 Blockes bezeichnet), das dadurch entsteht dass sich die quantisierten Koeffizienten nicht mehr in derselben Weise gegenseitig auslöschen wie das im Ursprungsbild der Fall war, und deswegen geriffelte Muster entstehen [47], siehe Abbildung 2.8.

2.2.4 Wavelets / JPEG2000

Eines der Ziele des „neuen JPEG“, JPEG2000, war es die Anwesenheit von Artefakten wie Blocking und Ringing zu verhindern, auch bei niedrigen Bitraten.

Dazu wurde u.A. die im „alten JPEG“ verwendete Fourier-Transformation durch eine Wavelet-Transformation abgelöst.
Genauer gesagt werden Bilder mittels der (umkehrbaren, verlustfreien) Legall 5/3 Transformation [1] oder der (nicht umkehrbaren, verlustbehafteten) Daubechies 9/7 Transformation [4] in Wavelet-Koeffizienten zerlegt, die dann mittels Quantisierung sowie arithmetischer Kodierung abgespeichert werden.

Abbildung 2.8: *Blocking und Ringing Artefakte bei JPEG komprimierten Bildern. Links: Blocking, Rechts: Ringing*

Die Abspeicherung bzw. Redundanzkodierung findet über sogenannte „Zerotrees“ statt, die eine effiziente Abbildung von Nullen in den verschiedenen Wavelet-Schichten erlauben [84].

JPEG 2000 kann (wie auch JPEG) mit mehreren Farbräumen umgehen, hat aber auch Unterstützung für palettenindizierte sowie monochrome Bilder.

Des Weiteren erlaubt es JPEG 2000, Bilder mittels „Tiling“ in beliebig große Blöcke zu zerlegen (anders als JPEG, wo die Blockgröße 8×8 hardkodiert war), und für jeden Block eine eigene Wavelet-Zerlegung und Kodierung durchzuführen. Auch das gesamte Bild kann als ein großer Block gespeichert werden, d.h. Tiling ist optional.

2.2.5 Motion-Estimation

Motion-Estimation ist ein Verfahren aus der Videokompression, das als erstes der in dieser Arbeit soweit referenzierten Verfahren speziell auf die Kompression von hintereinanderfolgenden Bildern (Videosequenzen) ausgerichtet ist.

Die Idee hierbei ist, dass Videos zwischen den einzelnen Frames oft ähnliche und wiederkehrende Bildausschnitte enthalten, diese allerdings durch Kameraschwenks, sich bewegende Objekte etc. nicht immer genau an derselben

Stelle liegen.

Dazu wird (analog zu JPEG) das Bild in (üblicherweise 8×8 oder 16×16) Blöcke aufgeteilt, und in jedem dieser Blöcke ein Richtungsvektor gespeichert, der angibt welcher $n \times n$ Block des direkt vorangegangenen Frames (oder, bei manchen Videocodecs, eines beliebigen, zeitlich weiter zurückliegenden Frames, einer sog. „bookmarked frame reference“) am ehesten der im aktuellen Block gespeicherten Bildinformation entspricht.

Aktuelle Codecs erlauben es nicht nur, auf reale $n \times n$ Blöcke zu verweisen, sondern auch auf interpolierte Blöcke zwischen Pixeln (sub-pixel precision) [106]. (Siehe Bild 2.9)

In der Regel wird dann die Differenz zwischen dem referenzierten Block und dem tatsächlich zu kodierenden Block abgespeichert, meist über ein JPEG / JPEG2000 ähnliches Transformation-Kompression basiertes Verfahren.

Bei günstiger Wahl des Referenzblocks ist diese Differenz („P-Block“) kleiner komprimierbar als der ursprüngliche Block. Ist die Differenz größer, so hat der Codec auch die Wahl, den Block nicht als Differenz sondern in seiner Originalform als „I-Block“ abzuspeichern. Für Keyframes, sowie Frames die zu viele I-Blocks enthalten, kann auch ein sogenannter „I-Frame“ erstellt werden. Bei diesem wird im Vornherein davon ausgegangen, dass er nur aus I-Blocks besteht, was pro Block ein Bit Information spart. (Bei einem 720x576 großen Videostream sind das bei einer Blockgröße von 8×8 Pixeln immerhin 810 Bytes)

Sind die Differenzen zwischen zwei hintereinanderfolgenden Frames zu hoch, so wird jeder Video-Codec einen Großteil der Blöcke als „I-Blocks“ (d.h. nicht Motion-Estimation basiert) kodieren, und eventuell sogar den gesamten Frame als I-Frame, d.h. als unabhängiges Bild.

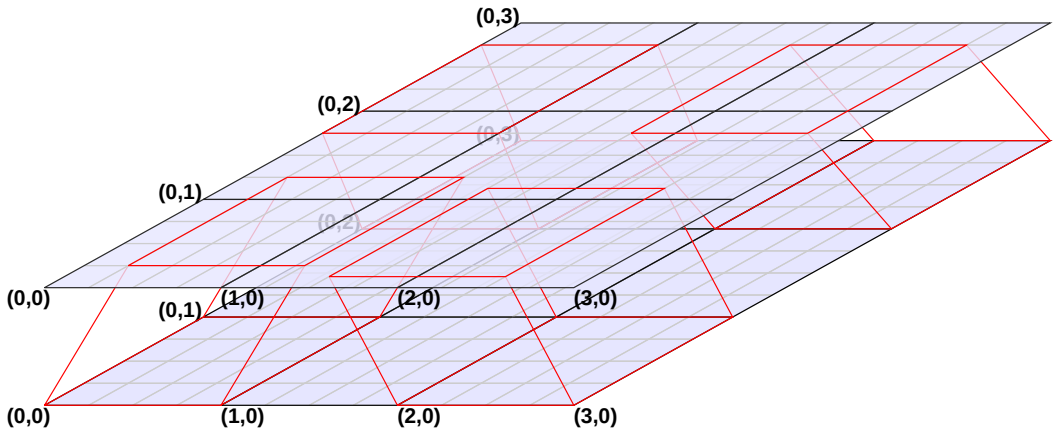


Abbildung 2.9: *Beispiel für P-Frame Referenzen zwischen Videoframes. Der linke untere Block hat den relativen Bewegungsvektor (1,1), d.h. verweist auf einen Block der um einen Pixel in beiden Richtungen verschoben ist. Der rechte untere Block verweist auf einen Subpixel-Block: er hat den Bewegungsvektor (6, 0.5)*

2.3 Experimentelle bzw. (noch) nicht etablierte Algorithmen

Nachdem in den vorhergegangenen Kapiteln eine Reihe von bereits etablierten, in festen Dateiformaten „verdrahteten“ Algorithmen beschrieben wurden, geht dieses Kapitel nun auf Verfahren ein, die sich noch in der Forschung befinden, oder die bisher im Bereich der „offiziellen“ Formate keinen festen Stand etablieren konnten.

Die Auswahl der Verfahren ist motiviert durch das Anliegen, Algorithmen zu finden, die sich auf Bildcluster erweitern lassen und dort erst zur Effizienzsteigerung neigen.

Insbesondere sind deswegen auch Kompressionsstrategien interessant, die bisher auf Einzelbildern noch keinen ernst zu nehmenden Gewinn abwerfen, da sie diese „Effizienzschwelle“ dann evtl. bei Bildmengen überschreiten.

Von den hier vorgestellten Algorithmen werden dann Karhunen-Loeve, die fraktale Kompression sowie die Textur-basierten Methoden in späteren Kapiteln unter dem Aspekt der Bildmengenkompression nochmals aufgegriffen.

2.3.1 Karhunen-Loeve / PCA

Die Karhunen-Loeve-Transformation ist (genauso wie die in JPEG verwendete Fourier-Transformation aus Kapitel 2.2.3) eine lineare Transformation: Das Bild wird in $n \times n$ Blöcke zerlegt, und jeder dieser Blöcke wird mittels einer einfachen Matrizenoperation in eine leichter transformierbare Version zerlegt.

Anders als bei der Fouriertransformation sind allerdings die Transformationskoeffizienten nicht fest vorgegeben, sondern werden aus der Kovarianzmatrix des Ursprungsbildes errechnet.

Die Idee hierbei ist, dass eine optimale Kompression dann erreicht wird, wenn die Blöcke des transformierten Bildes eine diagonalisierte Kovarianzmatrix haben, da dann die Pixel maximal unabhängig voneinander sind, also alle Redundanzen so gut wie möglich entfernt werden.
Also gilt für die Karhunen Loeve Matrix K, wenn die untransformierten Daten die Kovarianzmatrix C haben, dass die Kovarianzmatrix C_T der transformierten Daten ($C_T = (KD)(KD)^T = KCK^T$) eine Diagonalmatrix ist. Hierdurch werden die Daten optimal dekorreliert, und sind deswegen oft besser komprimierbar als mit statischen Matrizen [82].

Zur Berechnung der Matrix K benutzt man in der Regel eine Eigenwertzerlegung (SVD, Singular Value Decomposition) der Kovarianzmatrix C, d.h. man zerlegt sie in $C_T = K^T \Lambda K$ mit Λ eine Diagonalmatrix[2] (mit den Eigenwerten in der Diagonale). Hierfür gibt es auch schnellere Verfahren [62].

Nachteil dieses Verfahren ist allerdings die numerische Instabilität bei kleinen Bildern sowie der Overhead, der dadurch entsteht, dass die Kovarianzmatrix bzw. Karhunen-Loeve Matrix bei jedem Bild immer mit abgespeichert werden muss. Bei einer Zerlegung des Bildes in z.B. 8×8 Blöcke (analog zu JPEG) enthalten sowohl Kovarianzmatrix als auch Karhunen-Loeve Matrix $(8 \cdot 8)^2 = 4096$ Fliesskomma-Werte. D.h. die Erweiterung eines Verfahrens auf KLT lohnt sich nur dann wenn der Kompressionsgewinn durch die andere

[2] Eine alternative (aber äquivalente) Sichtweise ist die, dass jedes Bild (bzw. jeder Bildblock) aus der linearen Überlagerung von kleinen „Basiselementen“ besteht, die eine Art Baukastensystem für Bilder dieses Typs darstellen. Während die Fouriertransformation davon ausgeht, dass diese „Bausteine“ immer die zweidimensionalen Sinus- und Kosinusfunktionen sind, so geht die KLT bzw. die PCA (Principal Components Analysis) davon aus, das jedes Bild eigene (optimierte) Bausteine besitzt. Die dominanten Bausteine sind dann genau die Eigenvektoren der Matrix C, die zu den betragsmäßig größten Eigenwerten gehören [36].

Transformation größer als die kodierte Binärform dieser Matrix ist, was bei Bildern normaler Größe normalerweise nicht der Fall ist. (Bei Bildmengen wiederum schon, siehe Kapitel 3.3)

2.3.2 Steerable Filters

Wie bereits in Kapitel 2.2.2 angesprochen, ist der Ansatz von neueren Transformationsbasierten Verfahren, das Bild zuerst anhand einer auflösungs- und positionsabhängigen Wavelet-Transformation in verschiedene Schichten zu zerlegen, die sich danach besser komprimieren lassen.

Eine relativ neue Weiterentwicklung solcher Wavelet-Transformationen sind die „steerable Filters“ [29] bzw. Pyramids [89], die zusätzlich zur Position auch noch eine Orientierung analysieren bzw. speichern, und die die Eigenschaft haben, dass sich aus der Summe einer Reihe von richtungsbasierten Filtern für jeden Pixel (und jede Auflösungsstufe) die Richtung des stärksten Anstiegs (Gradient) ablesen lässt.
D.h. auch die in diesem Verfahren verwendeten Filter lassen sich durch lineare Kombination in jede beliebige Richtung „drehen“ und es werden damit nicht nur Richtungskomponenten der Basisfilter erfasst, sondern, durch das Zusammenspiel der Filter einer bestimmen Auflösung, auch noch alle übrigen Richtungen.

Ein Beispiel für die Zerlegung eines Bildes in eine Steerable Pyramid ist in 2.10 zu sehen.
Eine Steerable Pyramid ist allerdings in der Regel hochgradig überbestimmt [73]. D.h. Information wird hier redundant gespeichert, weshalb sich diese Transformation nur bedingt für den direkten Einsatz in der Bildkompression eignet. Sie hat allerdings gegenüber normalen Wavelets deutliche Vorteile in den Bereichen Bildanalyse und Textur-Synthese (siehe Kapitel 2.3.5), da sie mehr Informationen über lokale Komponenten des Bildes erfasst als Wavelets.

2.3.3 Textons

Genauso, wie man mittels KLT für blockbasierte Transformationsalgorithmen eine optimale Transformation berechnen kann, kann man auch in Filterbasierten Verfahren versuchen, die Transformation optimal auf die Eigenschaften des zu transformierenden bzw. zu komprimieren Bildes anzupassen.

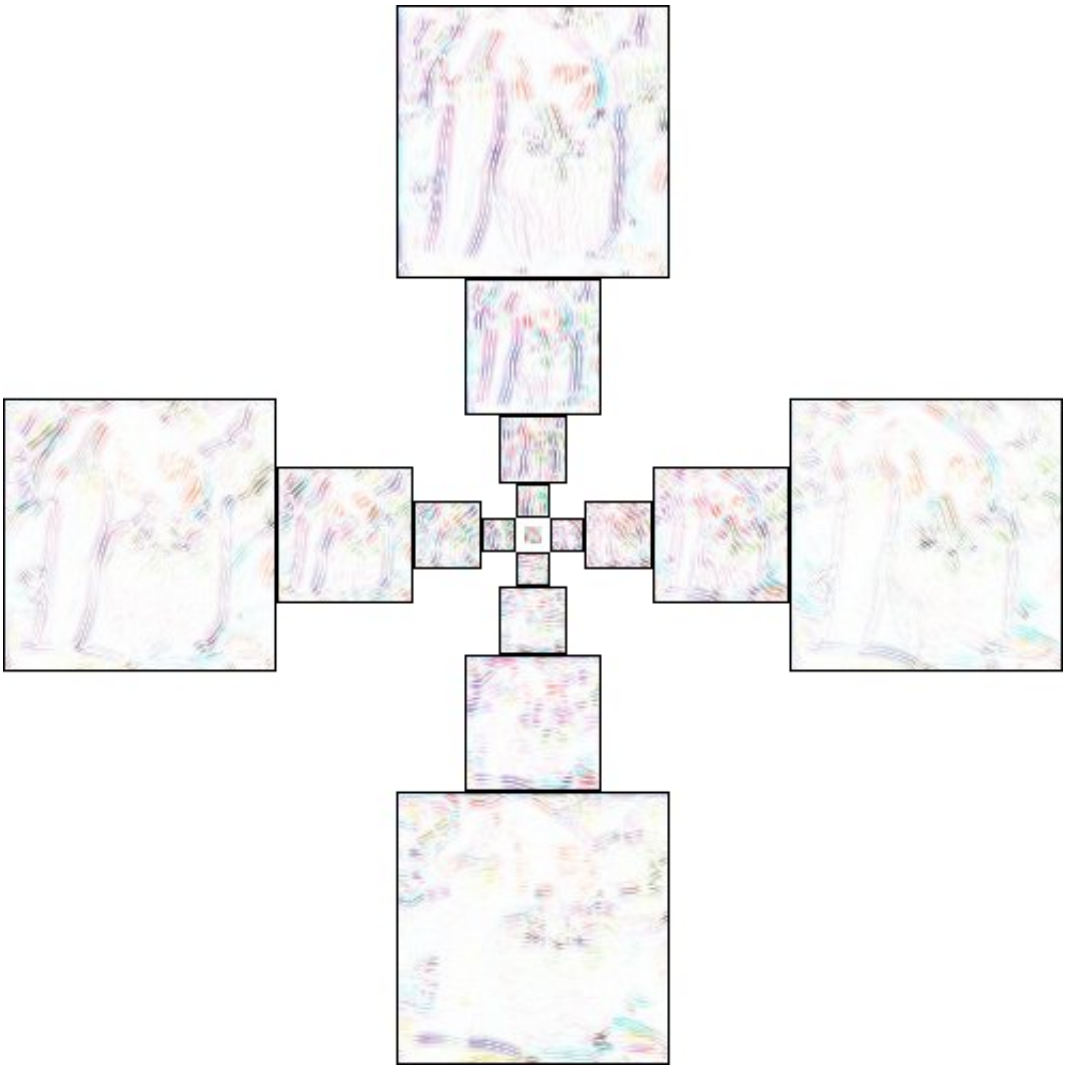


Abbildung 2.10: *Die Zerlegung des Bildes aus Abbildung 2.7 in eine „steerable Pyramid“ aus vier Richtungen und vier Skalen. Nicht dargestellt: Das Hochfrequenz-Residuum.*

Textons sind gewissermaßen das Filter-Äquivalent der bereits angesprochenen KLT Kompression. Auch bei Textons werden spezielle Filter berechnet, die genau auf das jeweilige Bild zugeschnitten sind.

Grafisch betrachtet sind Textons bestimmte Bildsymbole, von denen jedem Pixel genau eines zugeordnet wird. D.h. zu jedem Pixel wird nicht mehr die einfache Helligkeit gespeichert, sondern eine Umgebungsinformation [111]. Die Abbildung von Pixel zu Texton nennt sich „Texton-Map“.

Zur Erzeugung von Textons werden Bilder zuerst mit einer Filterbank gefaltet. Ein Beispiel für eine Filterbank, bestehend aus Gauß'schen Filtern, Differenz-Gauß-Filtern und verschieden orientierten Gabor Filtern ist in Bild 2.11 zu sehen. Daraus entsteht ein „hyperspektraler“ Vektor von Filterantworten („filter responses“) an jedem Punkt [64].

Diese Vektoren werden sodann durch einen Clustering-Algorithmus (meist K-Means, siehe Kapitel 4.3.6) zu Cluster-Zentren zusammengefasst. Diese Clusterzentren bestehen aus n-komponentigen Vektoren v, von denen jede Komponente der Ausprägung v_i eines Eintrags f_i der Filterdatenbank entspricht.

Umgekehrt können nun aus jedem Cluster-Zentrum Bildicons (Textons) generiert werden, durch eine „kleinste Quadrate“ Lösung der Formel:

$$I = \underset{I}{argmin} \sum_i (I \cdot f_i - v_i)^2$$

Da diese Gleichung überbestimmt ist, muss sie durch quadratische Optimierung gelöst werden.

Die Icons dienen allerdings primär der Anschauung; sie werden in den weiteren Schritten des Segmentierungsalgorithmus nicht weiter verwendet.

Durch erneute Faltung des Bildes mit allen Clusterzentren werden des Weiteren die bereits erwähnten Texton Maps generiert. Es sei angemerkt, dass die Clusterzentren — je nach Clusteralgorithmus — nicht zwangsläufig Bestandteil der ursprünglichen Filterdatenbank sein müssen.

Zur Veranschaulichung zeigt Abbildung 2.12 ein Bild und seine zugehörigen Textons sowie Texton Maps. Abbildung 2.13 zeigt außerdem ein Bild, bei dem für jeden Pixel nur das jeweilige Texton abgespeichert wurde.

Wie man sieht, erzeugt eine Kompression über Texton-Maps andere Artefak-

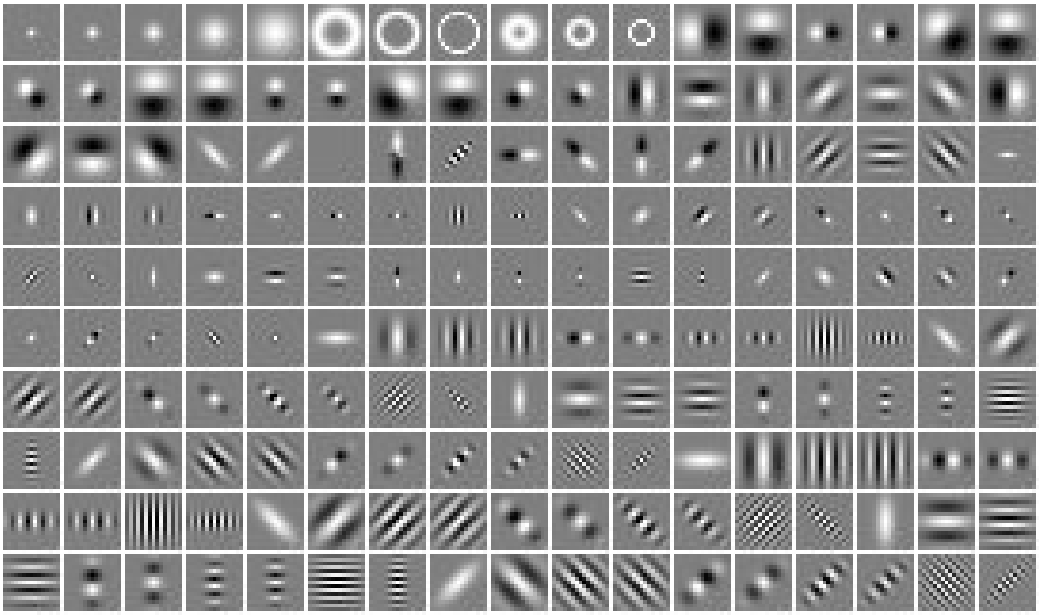


Abbildung 2.11: *Ein Beispiel für eine Filterbank, bestehend aus Gauß-Filtern, Differenzen von Gauß-Filter sowie verschiedenen Gabor-Filtern.*

te wie beispielsweise JPEG. Insbesondere ist eine künstliche Kantenschärfung erkennbar, was durch die höhere Granularität der Filterantworten zu erklären ist.
Allerdings erzeugt auch die Texton-Dekompression — je nach Dekompressor — u.U. das bereits bekannte Blocking, da immer rechteckige Pixelgruppen zusammen zurückgerechnet werden. Es ist allerdings wichtig zu erkennen, dass dieses Artefakt nur durch die Laufzeit- oder Speicherbeschränkungen des Dekompressors erzeugt wird, und nicht bereits dem ursprünglichen komprimierten Datenmaterial „innewohnen“.

Das Kompressionsverhältnis, das eine Texton-Kompression erzielen kann, errechnet sich über die Anzahl der verwendeten Textons (für jedes Pixel muss nur ein Texton-Index abgespeichert werden, z.B. bei 16 Textons benötigt man pro Pixel nur 4 Bit), sowie der Kompressionsrate der Texton-Map, die mit gängigen Verfahren (z.B. RLE oder PNG-ähnliche Algorithmen) komprimiert werden kann. Die Texton-Map ist in der Regel sehr gut komprimierbar, da sie viele Bereiche mit konstanten Werten enthält.

Der Nachteil des Texton-Verfahrens liegt allerdings in der visuellen Qualität (bei gleicher Bitrate erzeugen JPEG oder KLT „schönere“ Bilder), sowie des Berechnungsaufwandes beim Dekomprimieren, da die Pixel, ähnlich wie die

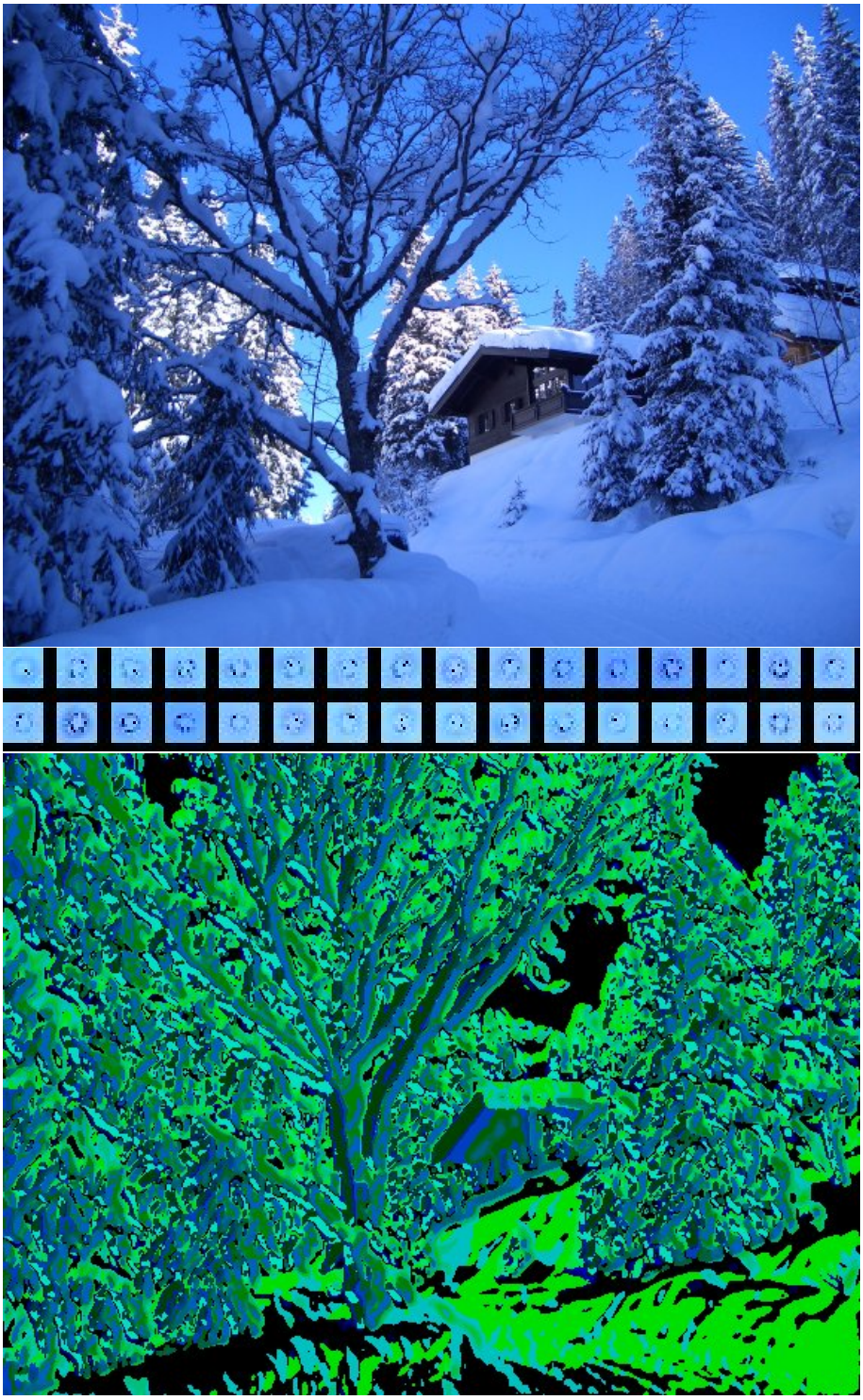


Abbildung 2.12: *Ein Bild (oben), aus dem eine Reihe von Textons (mitte) extrahiert wurde. Die Positionen der einzelnen Textons lassen sich aus der Texton-Map (unten) ablesen.*

Abbildung 2.13: *Verlustbehaftete Texton-Kompression des Bildes aus Abbildung 2.12. Die Pixel des Ursprungsbildes wurden aus der (mit rund 6 Bit/Pixel gut komprimierbaren) Texton-Map (Version mit 255 Einträgen) sowie den verwendeten Filtern zurückberechnet.*

Texton-Icons, durch eine quadratische Optimierung ermittelt werden müssen.

Aus diesen Gründen gibt es praktisch kaum Kompressionsverfahren, die auf Textons basieren, Texton-Maps sind allerdings nützlich u.A. zur Bildsegmentierung [64] sowie zur Bildklassifikation [87]. Insbesondere könnte für die in Kapitel 2.3.7 vorgestellte und in Kapitel 3.5.1 weiterverfolgte Methode der Textur-basierten Kompression die nötige Segmentierung auch mittels Textons durchgeführt werden.

2.3.4 Fraktale Kompression

Der Begriff „Fraktal“ wurde ursprünglich von Bernoit Mandelbrot geprägt, um Objekte zu beschreiben, die Selbstähnlichkeiten aufweisen, siehe Abbildung 2.14. Wenngleich Mandelbrot nicht an Datenkompression dachte, als er in seinem Buch „The Fractal Geometry of Nature“ [65] Fraktale beschrieb, so zeigte er aber zumindest, wie sich Fraktale zur Darstellung von natürlichen Objekten wie Wolken oder Pflanzen eignen.

Einzug in die Welt der Datenkompression erhielten Fraktale, als Michael Barnsley 1988 die Theorie der „Iterated Function Systems“ (IFS) [8] entwickelte, die sich zur fraktalen Repräsentation und Kompression von beliebigen Bildern eignete.

Die Idee eines IFS ist es, eine Funktion $f : R^2 \rightarrow R^2$ zu suchen, die ein Bild in einer geeigneten Weise transformiert, so dass der Fixpunkt I mit $f(I) = I$, der durch rekursive Funktionsanwendung ($I \approx f \cdot f \cdot f ... f(0)$) bestimmt werden kann, dem zu komprimierenden Bild entspricht.

Die Abbildung f ist bei einer optimalen Kompression dann so kurz wie möglich. Ist ein solches f gefunden, so lässt sich daraus das ursprüngliche Bild sehr schnell errechnen — allerdings kann es beliebig lange dauern, eine passende Abbildung zu finden.

Eine Weiterentwicklung (oder Vereinfachung) der Iterated Function Systeme stellen die „Partioned Iterated Function Systems“ (PIFS) dar, bei denen das Bild in rechteckige Blöcke zerlegt wird. Für jeden dieser Blöcke wird eine eigene Abbildung (meist eine Translation und Skalierung, siehe Abbildung 2.15) gesucht. [26]

Das Problem, für ein gegebenes Bild eine solche Abbildung zu finden, ist deutlich einfacher und weniger Rechenzeit-intensiv, weshalb es für derart

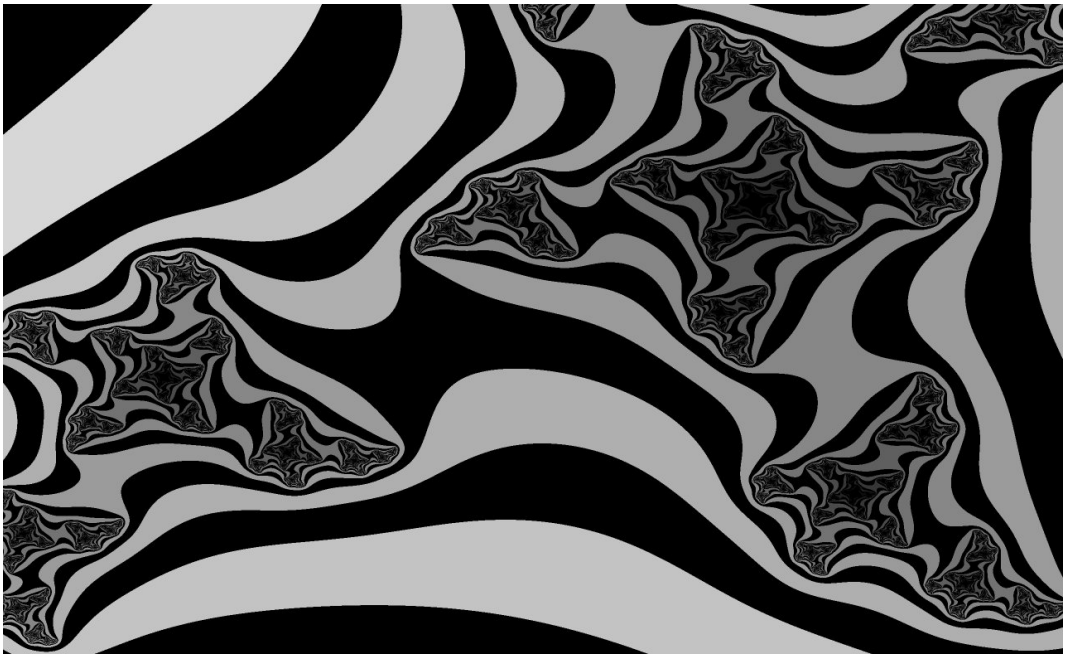


Abbildung 2.14: *Ein Mandelbrot-Fraktal*

aufgebaute Bilder auch schon Kompressoren und Datenformate gibt.

Das Format „FIF“, entwickelt von der Firma Iterated Systems, kodiert Bilder mittels einer Abwandlung von PIFS. Allerdings ist das Kodierungsformat und der genaue Algorithmus von Iterated Systems nicht offengelegt worden, weshalb es sich in der Praxis nicht durchgesetzt hat.

Fraktale Kompression und PIFS-Systeme sind aber immer noch ein aktives Forschungsgebiet, und werden neben der Datenkompression unter anderem auch zur Rauschunterdrückung [31], zur Textursynthese [49] und zur Gesichtserkennung [93] eingesetzt.

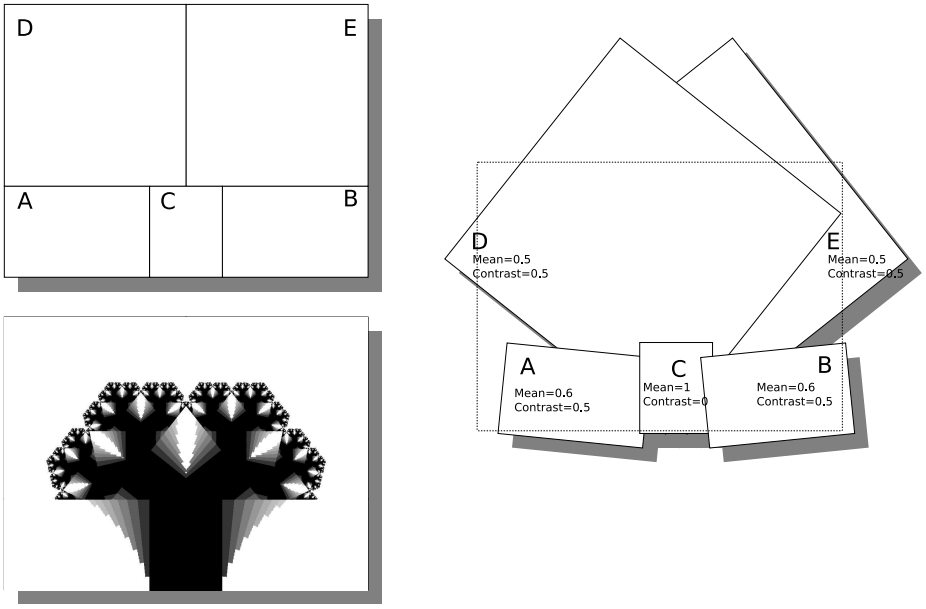


Abbildung 2.15: *Ein einfaches Beispiel einer PIFS Abbildung. Die Bereiche A,B,C,D und E werden durch Translation, Drehung und Helligkeits/Kontrastanpassungen auf andere Regionen des Bildes abgebildet (rechts). Daraus entsteht bei rekursiver Anwendung das Bild unten links. Die Gradienten am „Stamm“ des Baumes entstehen durch die rekursive Helligkeitsanpassung. Hierbei wird deutlich, dass fraktale Algorithmen mittels relativ weniger Parameter u.A. auch Farbverläufe darstellen können.*

2.3.5 Texturkompression

Die Texturkompression, d.h. die Kompression von kleinen, regelmäßig aussehenden Bildausschnitten, ist ein Bereich, der sich (entgegen den anderen bisher vorgestellten Verfahren) nicht mehr für generische Bilder eignet, sondern nur noch für Bilder eines ganz speziellen Typs, den besagten Texturen.

Haupteinsatz ist im Bereich der 3D Grafik, in der sehr große Mengen von Texturen auf möglichst wenig (Grafikkarten-)Speicher abgebildet werden müssen. Verwendung findet das der Texturkompression zugrundeliegende Gebiet der Analyse und Synthese von Texturen aber auch in der Bilderkennung, 3D-Grafik und Videobearbeitung [102] [80].

Bei der besagten Synthese von Texturen wird grundsätzlich versucht, eine durch (über Texturanalyse ermittelte) Parameter oder durch ein Beispielbild beschriebene Textur auf einer zweidimensionalen Fläche, oder auf einem 3D-Modell [98] zusammen zu bauen. Hierbei ist wünschenswert, dass die generierte Textur keine Kanten oder Artefakte aufweist, die auf ein „Tiling“ (regelmäßiges Zusammenbauen) des Beispielbildes hinweisen würde, und auch dass keine zu regelmäßigen Muster erkennbar sind, die nicht auch der ursprünglichen Textur innewohnten.

Das Interessante an der Textursynthese für die Datenkompression ist, dass Textursynthese in der Lage ist, Texturen relativ „kompakt“ (durch Texturparameter oder eine relativ kleine Beispieltextur) zu beschreiben, was eine kleinere Darstellung und Kodierung ermöglicht. Will man nicht nur Texturen, sondern auch „normale“ Bilder komprimieren, so lässt sich die Textursynthese des Weiteren auch in modellbasierte Verfahren einbetten, siehe Kapitel 2.3.7.

Man unterscheidet grundsätzlich drei Ansätze der Textursynthese:

Modelling: Ein statistisches Modell (wie z.B. ein neuronales Modell oder ein Hidden Markov Modell) wird anhand der Beispieldaten trainiert und dann dazu benutzt, die Textur zu generieren [15] [7]. Die momentanen Ansätze dieses Forschungszweigs sind zwar vielversprechend, hinken aber den alternativen Methoden bisher hinterher, weshalb sie für praktische Anwendungen momentan noch ungeeignet sind und in dieser Arbeit deswegen auch nicht weiter betrachtet werden.

Patching: Teile der Beispieltextur werden unregelmäßig so zusammengeflickt, dass ein neues Bild entsteht, das möglichst wenig sichtbare Kan-

ten enthält [25] [12] [24] [58]. Die Vorteile dieses Verfahrens sind sehr hochwertige Ergebnisse, und Applikationsmöglichkeiten im Bereich der Synthese von Fotos. Nachteil ist, dass die Textur durch ein Beispielbild anstatt eines Parametersatzes definiert wird, was dieses Verfahren insbesondere für die Datenkompression eher uninteressant macht.

Analysis and Synthesis: : Hier wird die Beispieltextur analysiert, und diverse Statistiken daraus extrahiert. Danach wird versucht, einem komplett neuen Bild diese Statistiken zu versehen [73]. Der Vorteil dieses Vorgehens ist die sehr kompakte Darstellungsform der Textur durch abstrakte Parameter.

Im folgenden eine detailliertere Beschreibung der letzteren zwei Verfahren.

Patching Einer der ersten Ansätze, Texturen auf Basis eines Beispielbildes neu zu synthetisieren geht zurück auf Efros & Leung [25]. Bei diesem Verfahren wird ein Bild Pixelweise zusammengestellt, wobei die Farbe eines Pixels im Zielbild dadurch entschieden wird, welche der Regionen des Quellbildes dem Arbeitsbereich im Zielbild am ehesten entspricht.

Efros optimierte diesen Ansatz später zusammen mit Freeman, indem er nicht mehr Pixel für Pixel arbeitete, sondern überlappende rechteckige Blöcke des Quellbildes stückweise ins Zielbild kopierte, und unregelmäßige Nahtstellen zwischen den Blöcken zuließ, die durch dynamische Programmierung ermittelt wurden [24].

Eine Forschungsgruppe aus Georgia entwickelte nun schließlich ein Verfahren, welches basierend auf minimalen Schnitten auf Graphen nicht mehr nur rechteckige Tiles, sondern beliebig geformte Bildausschnitte iterativ solange übereinanderstapelt, bis das zusammengebaute Bild keine sichtbaren Nahtstellen mehr aufweist [58].

Das letztgenannte Verfahren scheint momentan den „state of the art“ bzgl. Patching-basierten Syntheseverfahren darzustellen, und wird u.A. auch zur automatischen Synthese von Fotos und Kollagen benutzt [43].

Vorteile dieses Verfahrens ist u.A., dass es von vornherein auf Farbbildern arbeitet (und nicht, wie andere Verfahren, erst entsprechend erweitert werden muss), sowie dass es auch bei relativ komplexen Texturen immer gute Ergebnisse liefert.

Nachteil ist, dass keine Texturparameter extrahiert werden (die Synthesequelle ist das Beispielbild), d.h. es gibt kein explizites Modell einer Texturbeschreibung, was einem insbesondere bei Interpolation zwischen Texturen gewisse Freiheitsgrade nimmt.

Analyse & Synthese Die Idee, Texturen anhand von Parametern zu modellieren, geht auf Bela Julesz zurück [44], der 1962 folgende Vermutung anstellte:
Es gibt eine Menge von Funktionen $\theta_k(T), k = 1...n$, so dass zwei Textur-Samples T_1 und T_2 genau dann gleich aussehen, wenn gilt:

$$\forall k : \theta_k(T_1) = \theta_k(T_2)$$

Eine vielversprechende und relativ einfach zu implementierende Konkretisierung wurde von Heeger & Bergen entwickelt [38], in der durch schrittweise Anpassung von Texturparametern auf verschiedenen Auflösungsstufen und unter Einsatz verschiedener Filter Graustufentexturen entwickelt werden, die in vielen Fällen ähnlich aussehen wie das Original.

Eine sehr ausgebaute Weiterentwicklung davon geht auf Portilla und Simoncelli zurück. [73] Die Anzahl der zur Texturbeschreibung verwendeten Parameter ist hier stark nach oben gegangen, dadurch erzeugt der Algorithmus erstaunlich gute Syntheseresultate, siehe Abbildung 2.16.

Die folgenden Textur-Parameter werden hierbei modelliert:

1. Autokorrelation der Ausgabe von Hibert-erweiterten richtungsabhängigen High-Pass Filtern auf verschiedenen Auflösungsstufen
2. Kreuz-Korrelation dieser Filterfunktionen auf denselben Auflösungsstufen sowie zwischen Auflösungen — sowohl zwischen den reellen Anteilen als auch zwischen den Absolutwerten.
3. Schräge und Kurtosis des Bildes in verschiedenen Auflösungen.
4. Varianz und Erwartungswert der ungefilterten Bilder sowie der Hochfrequenzanteile (Filter-Residuum).
5. Pixelminima und -maxima des Bildes in der Originalauflösung.

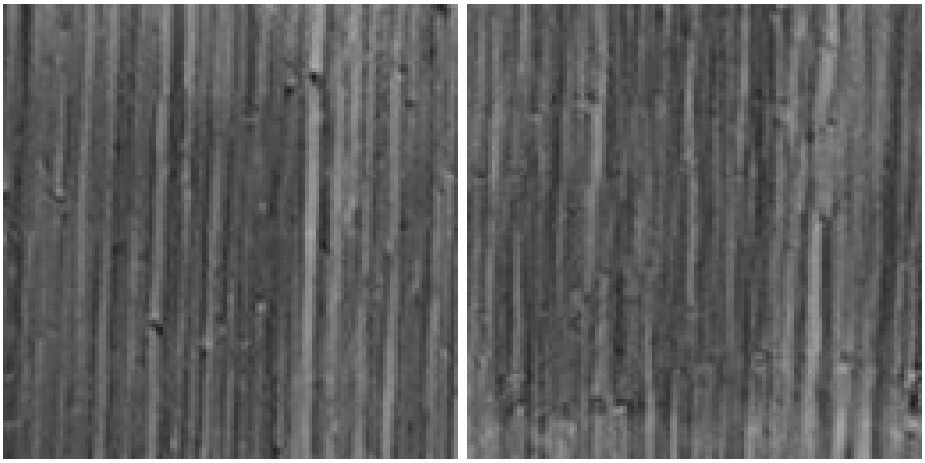


Abbildung 2.16: *Textursynthese nach Portilla & Simoncelli — links die Original-Textur, rechts das Syntheseresultat*

Diese Texturparameter werden aus der Ursprungstextur ermittelt, und eine synthetische Textur wird erzeugt, indem ausgehend von einem Zufallsbild (Gauß'sches Rauschen) solange die verschiedenen Texturparameter durch orthogonale Projektion angepasst werden, bis die Textur konvergiert ist.

Vorteil dieses Verfahrens ist, dass ein Bildunabhängiges Modell der Textur durch die verschiedenen Parameter erstellt wird. Nachteilig ist aber die Komplexität sowie die Laufzeit des Algorithmus.

2.3.6 Polygonalisierung

Polygonalisierung zerlegt ein Bild in eine Reihe von gleichfarbigen Polygonen. Diese können dann durch ihre Eckpunkte abgespeichert und mit Verfahren aus der 3D-Grafik komprimiert werden [78].

Insbesondere bei Bildern mit vielen gleichfarbigen Bereichen (wie z.B. Cartoons) wird dadurch ein Kompressionsgewinn erzielt.

Segmentierung nach Farben läuft in zwei Schritten ab [79]: Im ersten Schritt werden die n dominanten Farben des Bildes ermittelt, z. B. mittels eines Clustering-Algorithmus, siehe Kapitel 4.3. Danach wird allen Pixel des Bildes genau eine dieser Farben zugewiesen. Alle zusammenhängenden Bildbereiche einer gemeinsamen Farbe werden danach zu einem Segment zusammengefasst.

Um diese Bitmaps nun in „echte“ Polygone zu zerlegen, gibt es eine Reihe von Verfahren, zu denen u.A. in [10] ein guter Überblick gegeben wird. Diese Verfahren haben in der Regel auch einen „fitness“ Parameter, mit dem bestimmt werden kann, wie genau die generierten Polygone die Pixelgrenzen zwischen Regionen einhalten sollen.

Zur Illustration von Polygonalisierung wurden die beiden Bilder in Abbildung 2.17 nach Farben segmentiert. Wie man sieht, wurden heterogene Bereiche wie die Textur im rechten Bild in sehr viele kleine einzelne Polygone zerlegt. Im linken Bild wurden allerdings nur relativ wenig Polygone benötigt, da es sehr viele gleichfarbige Bereiche enthält.

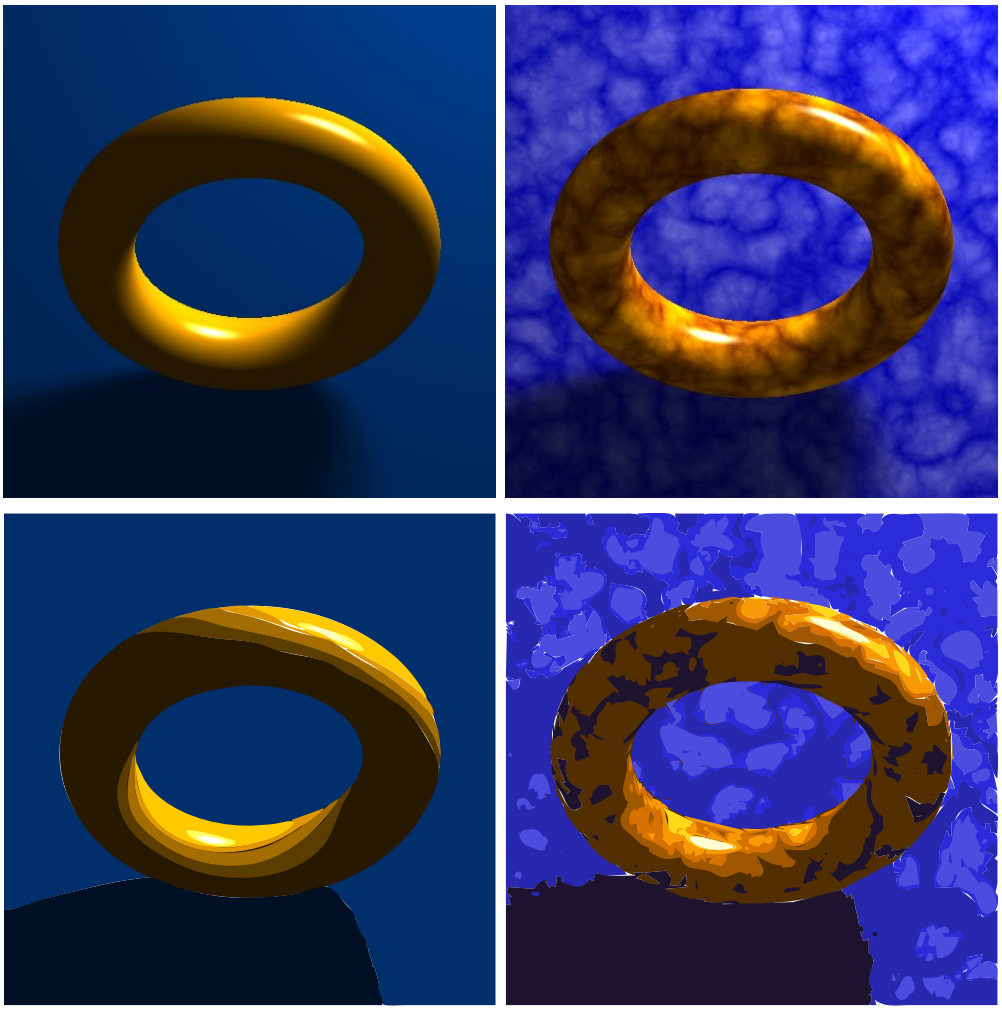


Abbildung 2.17: *Polygonalisierung von zwei Bildern. Oben: Die Originalbilder. Unten: Die in 10 Farben segmentierten Bilder*

2.3.7 Modellbasierte Kompression

Geht man ausgehend von der Polygonalisierung noch einen Schritt weiter, und zerlegt ein Bild in texturierte Polygon-Regionen, so ist man bei der sogenannten „Modellbasierten Kompression“ [85] [42] [80] [100].

Diese hat zum Ziel, ein Bild über Segmentierung und Texturanalyse möglichst parametrisiert zu beschreiben.

Bei der diesem Verfahren zugrundeliegenden Segmentierung wird das Bild aufgrund Texturcharakteristiken in unterschiedliche Regionen zerlegt. D.h. auch Bereiche, deren Pixel höchst unterschiedliche Farbwerte haben, können (im Gegensatz zur Polygonalisierung) nichtsdestotrotz zu einem Segment zusammengefasst werden.

Das Bild 2.18 wurde nach Texturen segmentiert. Auffälligster Unterschied im Gegensatz zur Farbsegmentierung ist, dass Texturen im Gegensatz zur Polygonalisierung (Bild 2.17) teilweise nur ein einziges Segment benutzen.

Textur-Segmentierung ist dafür deutlich aufwändiger, da nicht die direkten Pixelwerte für die Ermittelung des zuständigen Segments benutzt werden können, sondern stattdessen verschiedene Statistiken eines kleinen Bereiches im Umkreis eines Pixels seinen Wert festlegen. Hierfür gibt es eine Reihe von Verfahren, die im folgenden noch genauer vorgestellt werden.

Verfahren, die Kompression mittels Segmentierung vornehmen, gehen prinzipiell nach folgendem Prinzip vor:

- Zerlegung eines Bildes in Regionen mit „konstanten“ statistischen Eigenschaften, wie z.B. Texturen. (Segmentierung)
- Abspeichern der Regionen durch ein Mapping oder durch Kodieren der Polygoneckpunkte. (Polygon-Kodierung)
- Kodieren der einzelnen Regionen mittels eines statistischen Verfahrens, dessen Parameter speziell auf die jeweilige Textur angepasst sind. (Textur-Synthese/Analyse)

Die stark verlustbehafteten Verfahren dieses Typs nehmen hierbei keine Rücksicht auf das genaue Texturlayout im zu kodierenden Bild [80], sondern ersetzen ganze Flächen durch synthetisierte Texturen, wohingegen normal verlustbehaftete und verlustlose Verfahren [32] [77] die Textursegmente zwar

Abbildung 2.18: *Zerlegung eines Bildes in Segmente. Die gelben Linien stellen die Segmentbegrenzungen dar.*

analysieren, die Texturdaten dann aber nur zur Optimierung der Kodierung einsetzen.

Im folgenden werden nun die einzelnen Schritte der modellbasierten Kompression genauer erläutert.

Schritt 1: Segmentierung

Wie bereits angesprochen, hat die Segmentierung die Aufgabe, Bilder anhand von statistischen Eigenschaften in Regionen zu zerlegen. Normalerweise ist die Anwendung dieser Art von Bild-Segmentierung, Bildobjekte von ihrem Hintergrund zu trennen. [6]

Für zweidimensionale Bilder gibt es drei prinzipielle Verfahren: Bereichsverfahren, Kanten-basierte Verfahren, und iterative Verfahren.

Bereichsverfahren orientieren sich dabei an lokalen Pixelstatistiken (z.B. Farb oder Texton-Histogramme, siehe Kapitel 4.2.1), welche miteinander verglichen werden, und die an den Stellen, an denen Sie sich rapide verändern, eine Regiongrenze erzeugen. [71] [16] [96] Bereichsverfahren haben das Problem, dass sie dazu neigen, Kanten in Bildern zu „umrunden“, da Kanten je

nach verwendeter Statistik differenzierte Eigenschaften aufweisen [64].

Kantenbasierte Methoden benutzen nun wiederum genau diese Kanten: basierend auf Konturerkennung werden erst durch einen geeigneten Filter Pixel mit starken Gradienten (Kanten) gesucht, und diese dann in einer Art geeignet zusammengefasst, dass sich Objektumrundungspolygone ergeben. Dieses Verfahren hat allerdings Probleme, wenn Bereiche mit Texturen gefüllt sind, die ihrerseits Kanten (oder kantenähnliche Strukturen) aufweisen [63] [37] [35].

Iterative Verfahren benutzen meist sowohl Kantenerkennung als auch Bereichsstatistiken, um (meist auf verschiedenen Auflösungsstufen) zwischen den beiden Statistiken zu alternieren, und die temporäre Menge an Segmenten immer weiter zu verfeinern [64] [87]. So gut wie alle modernen Segmentierungsalgorithmen arbeiten nach diesem Prinzip.

Schritt 2: Polygon-Kodierung

Segmentierungsverfahren liefern als Ergebnis meist eine Mapping Bitmap, d.h. ein Bild, bei der die Zugehörigkeit jedes Pixels durch eine eindeutige Farbe angegeben ist, siehe Abbildung 2.19.

Um diese Bitmaps in Polygone zu zerlegen und zu komprimieren, bedient man sich dann Verfahren, wie sie in Kapitel 2.3.6 genannt wurden.

Schritt 3: Textur-Kompression

Es bleibt, die eigentlichen Texturen, sprich die „Füllung“ der einzelnen Polygone, möglichst effizient zu komprimieren.

Hierfür gibt es eine Reihe von Verfahren. Da die Texturen eines Bildes letztendlich wieder Bilder sind, kann prinzipiell jedes Kompressionsverfahren verwendet werden, was in der Lage ist, auf unförmigen (d.h. nicht rechteckigen) Bildregionen zu arbeiten.

Allerdings haben bei guter Segmentierung die einzelnen Bildteile sehr klare Muster bzw. Regelmäßigkeiten, und enthalten in der Regel keine lokale, zum Bildverständnis wichtige Information (wie z.B. Kanten, Objektgrenzen etc.) mehr, weshalb es sich lohnt, die Segmentkodierung über ein Texturorientiertes Verfahren (siehe Kapitel 2.3.5) durchzuführen.

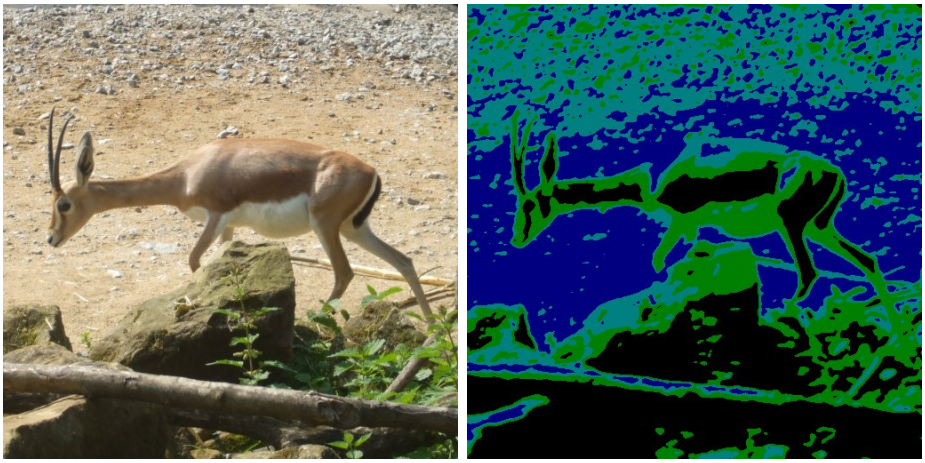


Abbildung 2.19: *Eine Mapping Bitmap (rechts) eines Bildes (links)- jeder Texturregion wird an jedem Pixel eine Nummer (bzw. hier: eine eigene Farbe) zugeordnet*

Probleme dieses Verfahrens

Wie bereits in der Kapiteleinführung angesprochen, sind momentan keine Verfahren bekannt, die das hier angesprochene **Segmentierung → Texturanalyse → Polygonalisierung → Kodierung** Datenkompressionsmodell implementieren.

Dies liegt u.A. an der Tatsache, dass jeder einzelne dieser Schritte sehr aufwändig und komplexer als so mancher „vollständige“ normale Kompressionsalgorithmus ist, als auch daran, dass bei Einzelbildern durch die Segmentierung nur relativ wenig Redundanz entfernt werden kann. Im wesentlichen läuft es nur auf eine Optimierung des statistischen Modells hinaus, die aber nicht ausgenutzt werden kann, wenn das Bild nicht in unterschiedlichen, nicht zusammenhängenden, Bereichen sehr ähnliche Texturen hat (was alleine schon wegen der perspektivischen Verzerrung unrealistisch ist).

Auch hier ändert sich aber die Situation, wenn das Verfahren von Einzelbildern auf Bildmengen erweitert wird — siehe Kapitel 3.5.

2.4 Zusammenfassung der besprochenen Algorithmen

Wie man diesem Kapitel entnehmen konnte, ist die heutige Bilddatenkompression ein relativ vielschichtiges Gebiet, in der eine große Anzahl von unterschiedlichen Algorithmen gegeneinander antreten.

Wohingegen die verlustlosen Algorithmen wie PNG (Kapitel 2.1.3) oder LOCO-I (Kapitel 2.1.7) im wesentlichen die klassische Datenkompression benutzen, um Bilder anhand ihrer (auf Byte-Ebene zu erfassenden) Redundanzen und Regelmäßigkeiten auf der „normalen Pixelebene“ zu komprimieren, arbeiten verlustbehaftete Algorithmen meist in einer anderen Beschreibungsform des Bildes (wie z.B. einer linearen Transformation (Kapitel 2.2.2), einer fraktalen Beschreibung (Kapitel 2.3.4), einer Zerlegung in Texturen (Kapitel 2.3.7) oder in lokale Filterzentren (Kapitel 2.3.3))

Bis auf wenige Ausnahmen (Kapitel 2.2.5) haben es diese Algorithmen gemeinsam, dass sie nur auf dem speziellen Fall der Einzelbildkompression vernünftig arbeiten, bzw. auf diesen optimiert sind.

Das Ziel der vorliegenden Arbeit ist es nach dieser Einführung nun, die Algorithmen aus den vorangegangenen Kapiteln insofern zu verändern und zu erweitern, dass sie es auch ermöglichen, (bessere) Kompressionserfolge auf Mehrbild-Sequenzen und Bilddatenbanken zu erzielen, sowie herauszufinden, wie man Bilder (oder Texturen) am besten so in Gruppen zusammenfasst, so dass sich die neuen Algorithmen optimal anwenden lassen.

Kapitel 3

Kompression von Bildmengen

Durch die Erweiterung von bereits etablierten Bildkompressions-Algorithmen (und auch Formaten) auf mehrere Bilder gleichzeitig lassen sich dabei die Redundanzen zwischen diesen Bildern ausnutzen.
Es gibt nun mehrere Möglichkeiten, welchen „Typ“ von Redundanzen man definiert — z.B. verwandte statistische Eigenschaften (wie Basisfunktionen), ähnliche Texturen oder vergleichbare räumliche Lokation von Objekten.
Dementsprechend gibt es (wie auch bei den Einzelbild-Algorithmen) vielfältige Ansatzpunkte für mögliche, auf Bildmengen erweiterte, Kompressionsalgorithmen.

Im Folgenden werden eine Reihe von erweiterten Algorithmen, die auf unterschiedlichen Ansätzen und Modellen basieren, vorgestellt.

Da es — unabhängig vom gewählten Algorithmus — zur Suche von optimalen Clustern wichtig ist, inwieweit sich zwei Bilder unter einem gegebenen Verfahren „ähnlich“ sind (sprich: wie gut sich zwei Bilder zusammen komprimieren lassen, im Vergleich zum Resultat was man erzielen würde, wenn man beide Bilder einzeln komprimiert), wird im folgenden auch auf entsprechende, auf dem jeweiligen Kompressionsverfahren aufbauende Ähnlichkeitsmetriken eingegangen.

Grundsätzlich lassen sich alle die Verfahren auf Bildmengen erweitern, die in irgendeiner Form großflächige Redundanzen und Ähnlichkeiten auch schon innerhalb von *Einzelbildern* erkennen und ausnutzen können.

Natürlich gibt es auch Verfahren, die nicht, oder nur durch erhebliche „Verfremdung“ des Algorithmus, erweiterbar sind. Diese arbeiten meist nur in

sehr lokalen Bereichen, und betrachten Bilder Pixel für Pixel oder Block für Block, ohne sich für die „Gesamtheit“ des Bildes zu interessieren.

Ein Beispiel für so einen (nicht erweiterbaren) Algorithmus ist die JPEG-Kompression — Wie bereits beschrieben, zerlegt diese jedes Bild in einzelne 8×8 Blöcke, von denen jeder unabhängig von den anderen transformiert und komprimiert wird. Die 8×8 Blöcke nehmen bei der Komprimierung keinerlei Einfluss aufeinander. (D.h. prinzipiell werden hier in Wirklichkeit für jedes „große“ Bild eine Reihe von unabhängigen „kleinen“ 8×8 Bildern komprimiert). Korrelation zwischen diesen 8×8 Blöcken werden nicht ausgenutzt — und deswegen auch keine Korrelationen zwischen Bildern in Bildmengen. Beim JPEG2000 Algorithmus verhält es sich ähnlich.

Auch die beispielsweise in BMP, PCX oder CCITT Bildern eingesetzte Lauflängenkodierung hat nur eine sehr begrenzte „Sichtweite“- hier werden nur die Korrelationen zwischen zwei direkt nebeneinanderliegenden Pixeln ausgenutzt, aber keine globale Redundanz, die sich auf Mehrbildkompression erweitern lassen würde.

Zweifellos gibt es Wege, Beispielverfahren wie diese in irgendeiner Form so zu „entfremden“ oder zu verbessern, so dass sie eventuell auch auf Bildmengen einsetzbar sind — beispielsweise gibt es in der Hyperspektralen Kompression Ansätze, aus mehreren Bildern Differenzbilder zu erstellen, und diese Differenzen mittels JPEG zu komprimieren [61].

Die in den nachfolgenden Kapiteln vorgestellten Algorithmen erheben deswegen (selbstverständlich) keinen Anspruch auf Vollständigkeit. Sie vermitteln aber einen guten Überblick über die Größe des Spektrums, aus der man wählen kann, wenn man Kompressionsalgorithmen auf Bildmengen implementieren will.

3.1 Fraktale Bildmengenkompression

Wie bereits in Kapitel 2.3.4 beschrieben, basiert das Grundprinzip der fraktalen Kompression darauf, dass für jedes Bild eine bestimmte Abbildung (dargestellt durch die Transformation einer Reihe von Bereichen des Bildes) gesucht wird, deren umgekehrte, rekursive Anwendung zurück zum ursprünglichen Bild führt.

Es gibt nun grundsätzlich zwei Möglichkeiten, dieses Verfahren auf Bildmengen zu erweitern: Das Einführen einer zusätzlichen Dimension, und damit die Erweiterung in den dreidimensionalen Raum, oder das Erlauben von Kreuzverweisen zwischen Bildern.

3.1.1 Erweiterung ins Dreidimensionale

Speichert man Bildmengen in einer 3D-Matrix, bei der die erste Dimension die Nummer des Bildes in der Gruppe und die übrigen beiden Dimensionen die Position innerhalb dieses Bildes beschreiben, so lässt sich auch der resultierende 3D Pixelwürfel mittels fraktaler Algorithmen komprimieren [60] [9]. Wie schon beschrieben bildet ein „Partioned Iterated Function System“ jeden Bildblock innerhalb eines Bildes durch eine lineare Transformation (sowie eine Helligkeits und Kontrastskalierung) auf einen anderen (gleich großen oder größeren) Bildblock in demselben Bild ab.
Analog zum PIFS zerlegt man das Bild bei der dreidimensionalen Version von diesem Algorithmus nun nicht mehr in 2D Rechtecke, sondern in kleine 3D-Quader, die durch eine Abbildung auf (verschobene, skalierte) 3D Quader beschrieben werden, siehe Abbildung 3.1.

Ein Nachteil dieses Ansatzes ist allerdings die im Vergleich zu 2D-Verfahren gesteigerte Komplexität von n^2 auf n^3, was die Suche nach geeigneten Bildtransformationen wieder in einen Bereich rückt, der für praktische Anwendungen zu aufwändig ist.
Des Weiteren müssen alle Bilder in der selben Größe/Auflösung vorliegen, was nicht immer gegeben ist. (Siehe Kapitel 3.2)

Eine sinnvollere Alternative, mehrere Bilder fraktal zu komprimieren, stellen Kreuzverweise zwischen Bildern dar, wie im nächsten Abschnitt beschrieben.

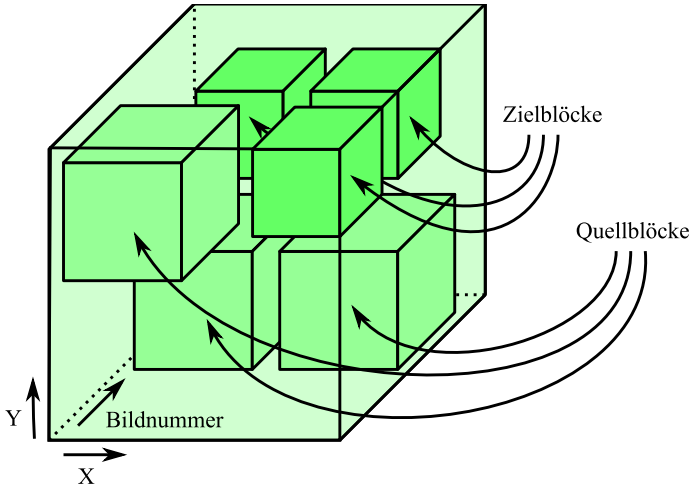


Abbildung 3.1: *Dreidimensionale fraktale Bildmengenkompression: Die Position eines Bildes in einer Bildsequenz wird als dritte Dimension betrachtet, und die fraktalen Funktionen sind Abbildungen zwischen dreidimensionalen Blöcken.*

3.1.2 Einführung von Kreuzverweisen zwischen Bildern

Kreuzverweise zwischen Bildern folgen der Idee, dass zwar Verweise eine dritte Komponente (die der Position des Bildes in der Bildsequenz) bekommen, die Blöcke, auf die bzw. von denen verwiesen wird, allerdings weiterhin nur zweidimensional sind, siehe Abbildung 3.2.
Die daraus resultierende fraktale Funktion für jedes Bild (bzw. jeden Bildblock) enthält als Folge dessen als Eingabewert bzw. -daten nicht mehr nur das momentan zu (de-)komprimierende Bild, sondern die Gesamtzahl aller Bilder der Bildmenge.

Die Dekompression erfolgt nun durch eine Art „verschränkte Rekursion“:
Sind beispielsweise I und J zwei zu komprimierende Bilder, und

$$f : R^{n\times m}, R^{n\times m} \rightarrow R^{n\times m}$$

und

$$g : R^{n\times m}, R^{n\times m} \rightarrow R^{n\times m}$$

die daraus errechneten Funktionen, so stellt die wiederholte Anwendung von

$$\begin{aligned} I_{n+1} &= f(I_n, J_n) \\ J_{n+1} &= g(I_n, J_n) \end{aligned}$$

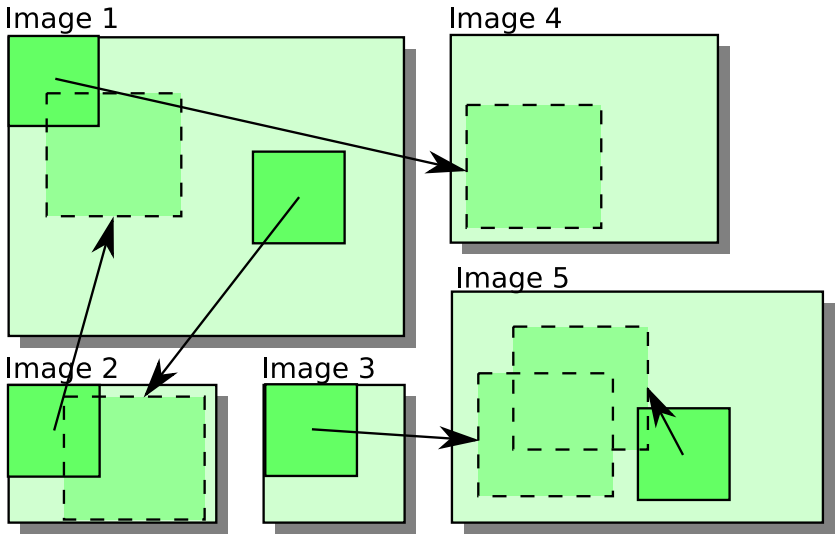


Abbildung 3.2: *Das Prinzip der fraktalen Kompression mit Kreuzverweisen: Es werden zweidimensionale Bereiche zwischen Bildern referenziert.*

das ursprüngliche Bild wieder her, siehe Abbildung 3.3. Die Anzahl von Bildern ist hierbei beliebig.

Allerdings steigt der Suchraum nur linear zur Anzahl der Bilder, die mittels eines verschränkten IFS oder PIFS komprimiert werden. Der daraus resultierende Aufwand ist entgegen dem Verfahren der Dimensionserweiterung aber noch überschaubar.

3.1.3 Algorithmus

Der im Rahmen dieser Arbeit implementierte fraktale Kompressionsalgorithmus auf Bildmengen komprimiert Bilder beliebiger Größe, und speichert sie in einer verschränkt rekursiven fraktalen Datenstruktur ab.

Bilder werden hierzu zuerst in Y/U/V Komponenten zerlegt, um auch Farbkodierung zu ermöglichen- hierbei werden die U und V Komponenten genauso behandelt wie ein eigenes Bild, und in derselben Gruppe mit dem Helligkeitsbild kodiert.

Die Bilder werden dann durch rekursive Vierteilung von Bildausschnitten in kleinere Blöcke zerlegt, siehe Abbildung 3.4.

Dies passiert solange, bis jeder Block durch eine fraktale Abbildung in adä-

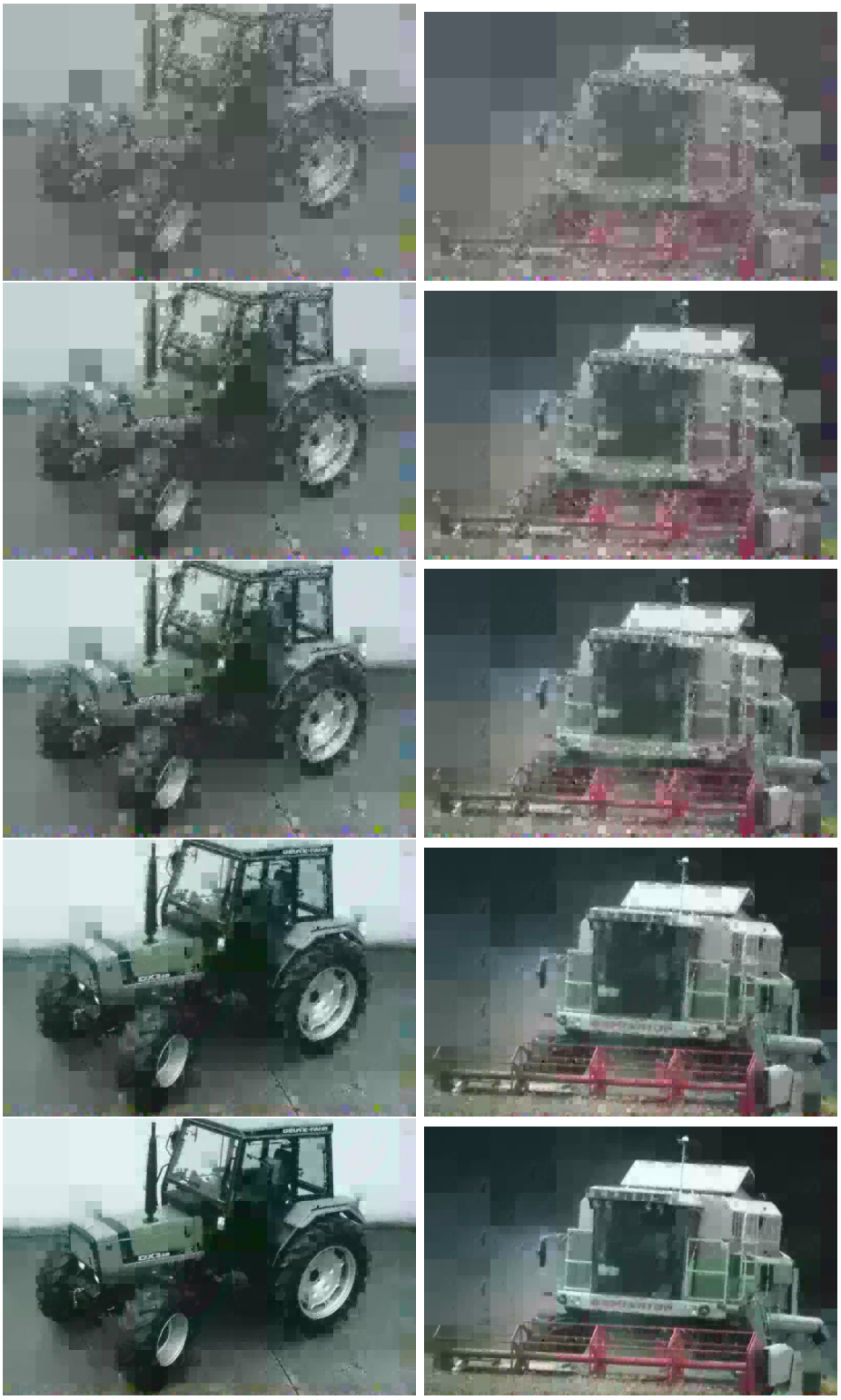


Abbildung 3.3: *Verschränkt rekursive Rückentwicklung von zwei fraktal miteinander komprimierten Bildern.*

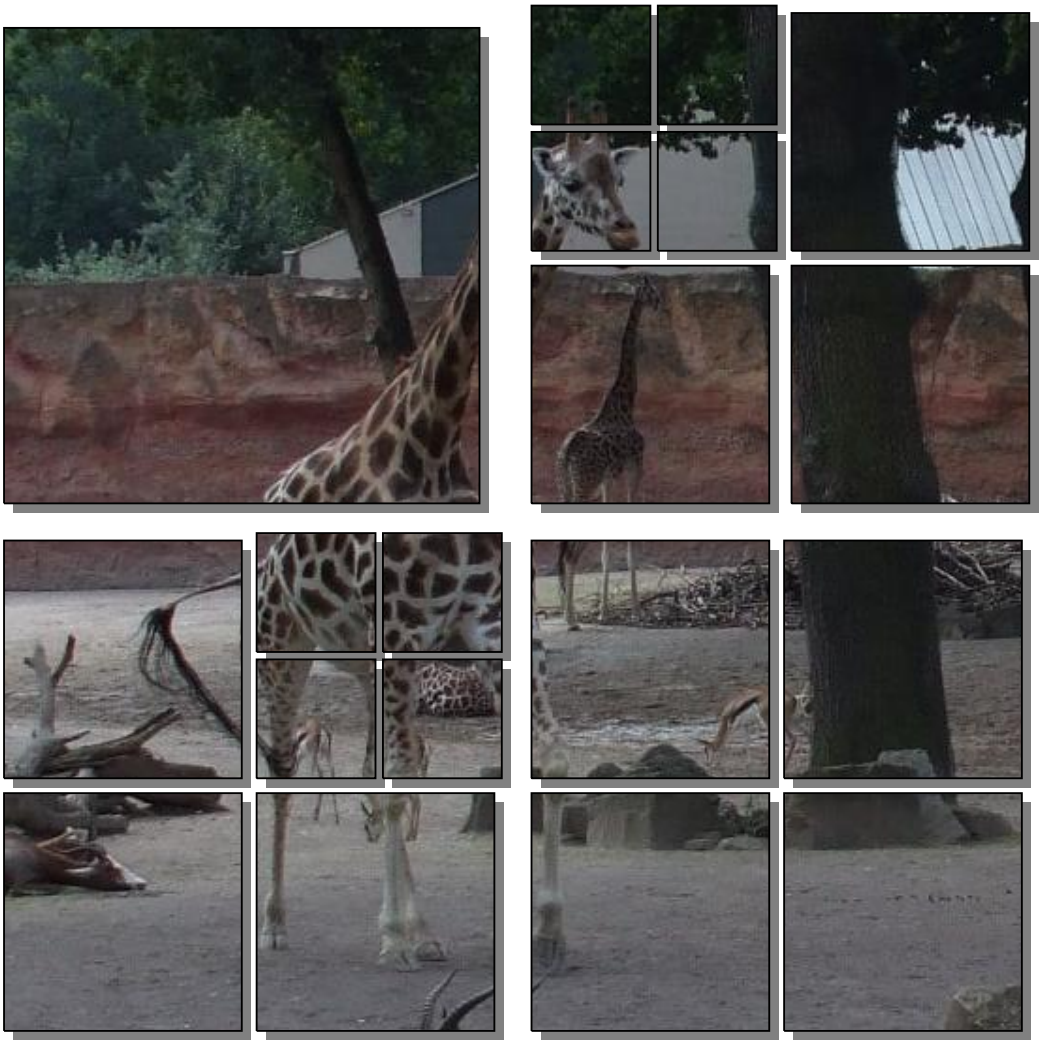


Abbildung 3.4: *Aufteilung eines Bildes mittels des Quadtree-Verfahrens: Die Bildblöcke werden entsprechend ihrer Komplexität in vier kleinere Bildblöcke zerlegt.*

quater Qualität darstellbar ist. Die gewünschte Kompressionsqualität des Bildes ist hierbei ein einstellbarer Parameter.

Da der Suchraum im Vergleich zur Einzelbildkompression bei der Mehrbildkompression von n Bildern immerhin n-mal so groß ist, ist es sinnvoll, zwei Optimierungen vorzunehmen:

- Nicht jede Pixelposition wird als mögliches Abbildungsziel behandelt, sondern nur alle (x, y)-Koordinaten mit x und y Vielfache von l/g, wobei l die Größe des abgebildeten Blocks ist, und g ein Granularisierungsparameter zwischen 1 und l, wobei $g = l$ die langsamste und $g = 1$ die schnellste Kompression bedeutet. Da der Parameter g an den Dekompressor übergeben wird, spiegelt sich eine langsamere Kompression aber nicht zwangsläufig in schlechteren Kompressionsresultaten wieder, da eine höhere Granularität (sprich weniger mögliche Koordinaten) auch weniger Bits Information bedeutet [82].
- Für alle Quellblöcke sowie das zu kodierende Bild wird die erweiterte Fouriertransformation sowie der Mittelwert und quadratische Mittelwert vorberechnet. Dadurch lassen sich Vergleiche zwischen Quell- und Zielblöcken unabhängig von der Blockgröße in konstanter Zeit durchführen [53].
- Die möglichen Quellblöcke für jedes Bild werden vorher in Klassen zerlegt, um eine leichtere Auffindung eines Quellblocks für eine bestimmte (bekannte) Klasse zu ermöglichen. Jeder mögliche Quellblock wird hierbei in vier Teile zerlegt, und diese nach Helligkeiten sortiert. Die Reihenfolge dieser Helligkeiten legt die Klasse fest, wodurch sich $4! = 24$ mögliche Unterteilungen ergeben [82].

Durch diese Optimierung lassen sich Bildmengen von bis zu ca. 30 Bildern auf heutigen Prozessoren mit erträglicher Rechenzeit komprimieren (Siehe Abbildung 3.5).

Für die Speicherung der komprimierten Bilder müssen für jeden einzelnen Block die folgenden Information abgelegt werden:

- Quad-Tree-Aufbau: Information, ob ein Block nochmals in vier Teile unterteilt wird, oder ob eine Kodierung folgt (1 Bit Information, falls der Block weiter unterteilt wird, wird nur dieses Feld gespeichert)

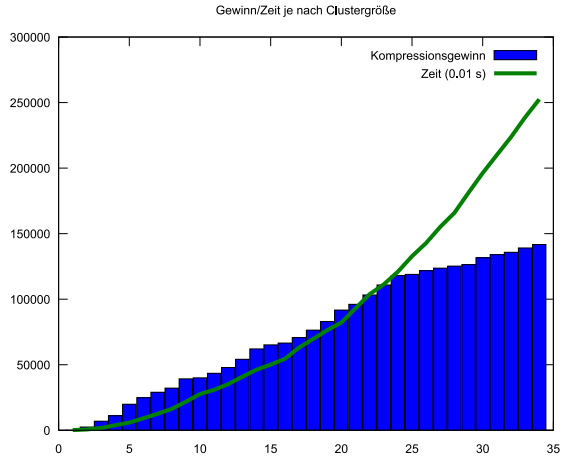


Abbildung 3.5: *Kompressionsgewinn sowie zur Komprimierung benötigte Zeit. (auf einem Pentium M mit 2.0 Ghz) Die zusammen in einer Bildmenge komprimierten Bilder hatten eine Größe zwischen 800x475 und 800x730 Pixeln. Bei diesen Bildgrößen liegt die optimale Clustergröße bei zwischen ca. 20 und 25 Bildern pro Cluster, da die Rechenzeit ja mit der Anzahl der Bilder quadratisch ansteigt, beim Kompressionsgewinn aber ab 20 Bildern langsam ein sichtbarer „Sättigungseffekt“ einsetzt.*

- Bildnummer i: Bildnummer des Quellblocks aus dem dieser Bildausschnitt generiert wird. ($\lceil log_2 n \rceil$ Bits Information, mit n die Anzahl der Bilder)
- Blockgröße l: Größe des Blocks, auf den verwiesen wird. ($\lceil log_2 s \rceil$ Bits Information, mit s die Anzahl der möglichen Blockgrößen)
- X/Y-Koordinate x, y: Falls Kodierung: Position des Blockes innerhalb des Zielbildes ($\lceil log_2 \lceil \frac{gw}{l} \rceil \lceil \frac{gh}{l} \rceil \rceil$) mit w, h die Höhe/Breite des Quellbildes i, g der Granularisierungsparameter und l die Blockgröße)
- Kontrast/Helligkeit: Falls Kodierung: Kontrast/Helligkeit, auf die der referenzierte Block angepasst wird. ($\lceil log_2(c_{max}) \rceil + \lceil log_2(l_{max}) \rceil$ Bits Information, wobei c_{max} und l_{max} jeweils den Maximalwert der zu kodierenden Helligkeit und Luminanz angeben. Mit größeren Werten in diesem Feld wird bessere Qualität erzielt, auf Kosten von schlechterer Kompressionsrate.

Die genaue Kodierung der einzelnen Felder ist frei, und kann z.B. über Huffman oder Arithmetische Kodierung optimiert werden.

Die Feld-Tupel werden dann rekursiv abgespeichert. Der Dekompressor benötigt zur Rekonstruktion des Kompressionsbaumes nur die genannten Informationen sowie die Werte c_{max}, l_{max}, g, w,h und s, die im Dateiheader abgespeichert werden.

Danach legt er (anfangs schwarze) Bilder der kodierten Größe an, und entwickelt dann durch rekursive Abbildung der affinen Transformationen jedes Bildes die ursprünglichen Bilder zurück. Das heißt natürlich auch, dass zur Dekompression eines Bildes aus einer Menge von zusammen komprimierten Bilder **alle** Bilder dekomprimiert werden müssen.

Die dekomprimierten Bilder haben natürlich aufgrund der Natur des Kompressionsalgorithmus (genauso wie die meisten anderen fraktalen Kompressoren) teilweise JPEG-ähnliche Block-Artefakte zwischen den einzelnen komprimierten Regionen. Diese sind allerdings nicht zwischen jeden 8×8 Blöcken, sondern nur zwischen den (unterschiedlich großen) Blättern des durch die rekursive Vierteilung gewonnenen Baumes.

3.1.4 Abschätzung der fraktalen Komprimierbarkeit

Da der fraktale Algorithmus relativ viele Rechenschritte durchführen muss, um eine Bildmenge zu komprimieren, bietet es sich an, auch Abschätzungen zu implementieren, die angeben, wir gut eine Bildmenge ungefähr komprimieren würde, ohne den (gesamten) eigentlichen Kompressionsalgorithmus laufen zu lassen.

Hierbei ist es nützlich, sich klar zu machen, dass zwei Bilder, die die zwischen ihnen herrschenden Redundanzen nutzen, relativ viele Kreuzverweise (d.h. nicht Verweise innerhalb eines Bildes, sondern von einem zum anderen Bild) haben, und weniger auf sich selber referenzieren. Siehe Abbildung 3.6.

Dies motiviert eine Abschätzung

$$s_{IJ} = \frac{\sum_{r \in I_r \cup J_r} b(r)}{|I_r| + |J_r|} \tag{3.1}$$

Mit I_r, J_r die Menge der jeweils von den Bildern I und J verwendeten Referenzen, und $b(r) = 1$ falls r eine Selbstreferenz (also ein Verweis innerhalb des gleichen Bildes) ist, sonst $b(r) = 0$.

Ein Nachteil dieser Statistik ist die Tatsache, dass sie keine mathematische Metrik ist.

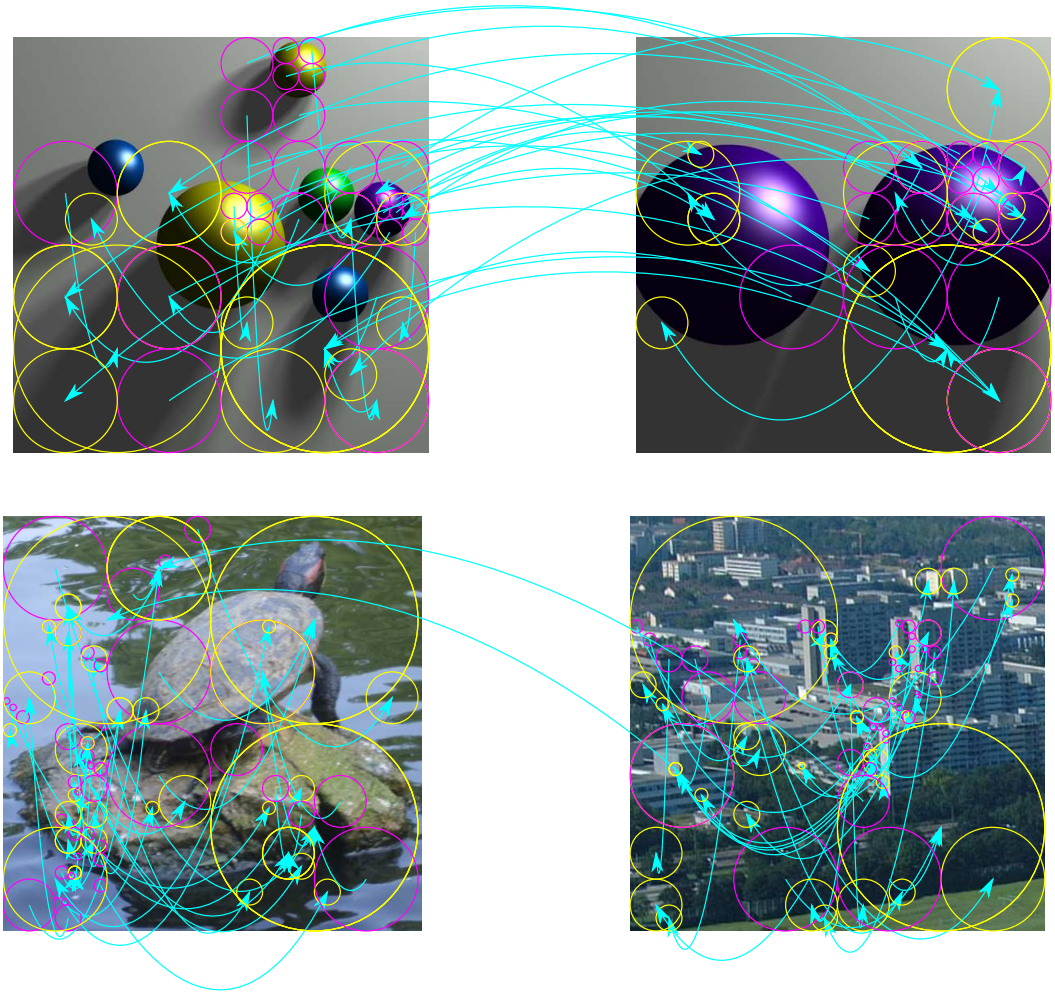


Abbildung 3.6: *Zwei Bildpaare, mit unterschiedlich guten Eigenschaften bzgl. gemeinsamer fraktaler Kompression. Die oberen beiden Bilder haben sehr gute gemeinsame Kompressionseigenschaften, wohingegen die unteren beiden zusammen nur geringfügig besser komprimieren als einzeln. Die Anzahl der Referenzen (blaue Pfeile, von Quellregion (lila Kreis) nach Zielregion (gelber Kreis)) zwischen den Bildern sind beim ersten Bildpaar auch deutlich höher. Bereiche mit zu niedrigem Kontrast (und deswegen ohne Bewegungsvektor) sind nicht abgebildet.*

Insbesondere gilt

$$\forall I : s_{II} \neq 0$$

(d.h. das Metrik-Axiom $d(x, x) = 0$ wird verletzt) da bei einem Bild, das mit sich selbst komprimiert wird, keine Fremdreferenzen existieren[1]. (Schließlich enthält das jeweils andere Bild keine Bildregionen, die „besser“ sind als die Regionen des eigenen Bildes)

Dies entspricht aber dem tatsächlichen Kompressionsverhalten: Im Gegensatz zu anderen Verfahren komprimieren beim fraktalen Algorithmus Bilder, die identisch oder auch nur „zu“ ähnlich sind, schlechter miteinander, da durch das Hinzufügen des zweiten Bildes keine zusätzliche für das erste Bild verwertbare Information erzeugt wird. (Siehe auch Abbildung 3.7)

Ein weiterer Nachteil ist allerdings, dass diese Statistik relativ aufwändig zu berechnen ist, da sie für jedes Bild voraussetzt, dass dieses mit allen anderen Bildern einen Kompressionslauf durchläuft, um die jeweils optimalen Block-Referenzen zu finden. D.h. letztendlich ist der Aufwand für die Distanzabschätzung ähnlich groß wie der Aufwand für die eigentliche Kompression.

Eine Optimierung besteht darin, nicht die Originalgröße der Bilder zur Distanzabschätzung zu verwenden, sondern eine herunterskalierte Version. Heruntergerechnete Bildpaare haben meist vergleichbare „Ähnlichkeitseigenschaften“, weshalb diese Statistik relativ genaue Approximationen liefern kann, siehe Abbildung 3.9.

Nichtsdestotrotz ist aber auch die fraktale Kompression von kleineren Bildern mit beträchtlichem Rechenaufwand verbunden, weshalb es sinnvoll scheint, diese Statistik durch alternative (generische) Bildstatistiken zu approximieren. Hierauf wird in Kapitel 4.2 eingegangen.

[1] Es gibt Versionen von fraktalen Algorithmen, die bei mehreren Bildern eine Abbildung zwischen Quell- und Zielblöcken der gleichen Größe erlauben [27].
Damit können Bildausschnitte zwischen Bildern kopiert werden (ähnlich wie bei Motion Estimation). Dies funktioniert allerdings nur in eine Richtung, sprich quasi als „Rückverweis“ in einer Bildkette, damit die Abbildung kontraktiv bleibt und konvergiert.

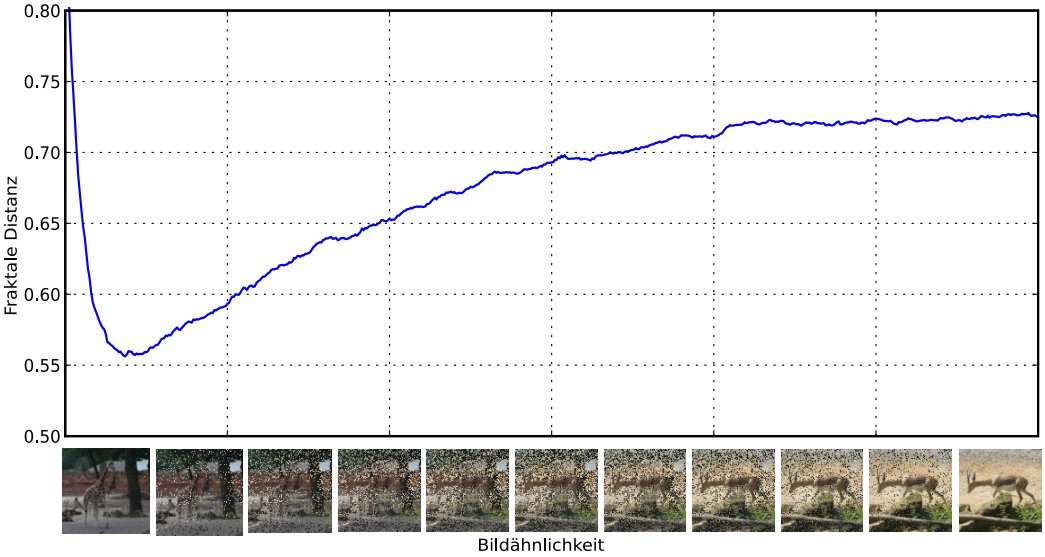


Abbildung 3.7: *Die fraktale Ähnlichkeits-Metrik (nach Formel 3.1) für einen Bildverlauf zwischen zwei Bildern. (oben) Die Distanz ist anfangs relativ groß (sprich auf Basis dieser Metrik ist das Bild zu sich selbst „unähnlich“, siehe auch Abbildung 3.8), und erst bei Veränderungen am Bild wird der fraktale Kompressionsgewinn größer als bei der Einzelbildkompression.*

Abbildung 3.8: *Ein Bild, das „mit sich selber“ komprimiert wurde. Da das jeweils andere Bild keine Information enthält, die nicht schon mittels einer Selbstabbildung aus dem eigenen Bild zu ermitteln ist, gibt es keine Fremdreferenzen in das andere Bild- dementsprechend ist die „Distanz“ zwischen den beiden Bildern sehr groß.*

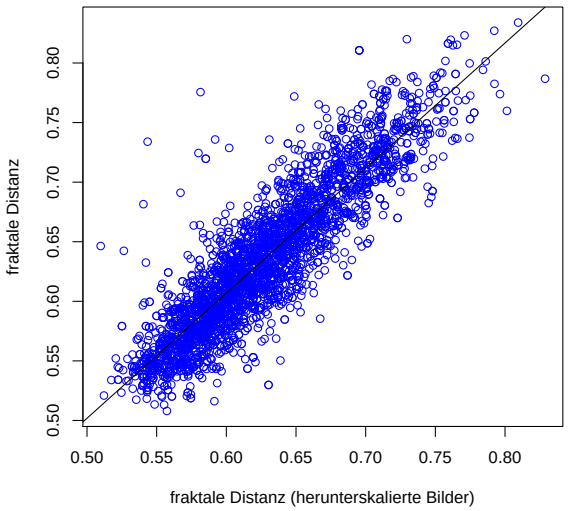


Abbildung 3.9: *Fraktale Distanz (nach Formel 3.1) zweier Bilder (vertikal) gegenüber Ähnlichkeit von zwei um den Faktor zwei herunterskalierter Bilder (horizontal). Jeder Punkt entspricht einem Bildpaar, das zufällig aus 128 Beispielbildern ausgewählt wurde.*

3.2 Videocodecs

Da das Feld der Videokompression seit langem etabliert ist und sich schon seit jeher mit der Kompression von korrelierten Bildern (Frames) beschäftigt, liegt es nahe zu versuchen, die dort verwendeten Methoden der Videokompression auch für die Kompression von Bilddatenbanken einzusetzen.

Videokompression ist allerdings letztlich nur ein Spezialfall der im vorigen Kapitel vorgestellten fraktalen Mehrbild-Kompression:
Das Grundprinzip (Abbildung von Bildblöcken zwischen Bildern) ist gleich, allerdings erlaubt die Videokompression im Gegensatz zur fraktalen Kompression nur Verweise in eine Richtung, und keine Größen- und Kontrastskalierung zwischen den Bildblöcken, siehe Abbildung 3.10. D.h. Videokompression ist deutlich weniger „mächtig“, dafür aber in der Regel schneller.

Da allerdings im vorigen Kapitel bereits die „Verallgemeinerung“ des Einsatzes von Videokompression für die Bilddatenbank-Kompression erörtert wurde, erscheint es wenig praktikabel, nochmals einen speziellen Videocodec zu entwerfen, da dieser kaum bessere Kompressionsresultate als die bereits erläuterte fraktale Kompression erzielen würde.

Eingegangen werden soll in diesem Kapitel deswegen nur auf einen Spezialfall: Die Verwendung von existierenden, bereits etablierten, Videoformaten, um Bildmengen zu komprimieren.

Hierbei sind allerdings zwei Probleme relativ offensichtlich: Erstens sind heutige Videocodecs in der Regel nicht darauf ausgelegt, Frames unterschiedlicher Größen bzw. Auflösungen zu komprimieren[2], zweitens gehen Videocodecs davon aus, dass ein Großteil der Frames stärker lokal korreliert ist (Beispiel Kameraschwenks, Bewegung einzelner Akteure im Bild) als in der Regel bei Bilddatenbanken der Fall.

Das Problem der Auflösungen wirkt allerdings auf den ersten Blick schwieriger als es tatsächlich ist. Bei einer klassischen Bilddatenbank wird normalerweise nur eine geringe Anzahl von Bildauflösungen verwendet — insbesondere da bei Datenbanken mit Foto- und Digitalisierungs-Bildbeständen meist Auflösungen vorliegen, die sich bei Digitalkameras und -scannern als

[2]Einige neuere Videocodecs (wie z.B. On2 TrueMotion VP6) ermöglichen die Umschaltung von Auflösungen in einem I-Frame. (Allerdings nicht in einem P- oder B-Frame) D.h. es ist zwar möglich, während des Videos die Auflösung zu ändern, allerdings können sich Frames unterschiedlicher Auflösung nicht referenzieren (da sie immer durch einen I-Frame getrennt sind)

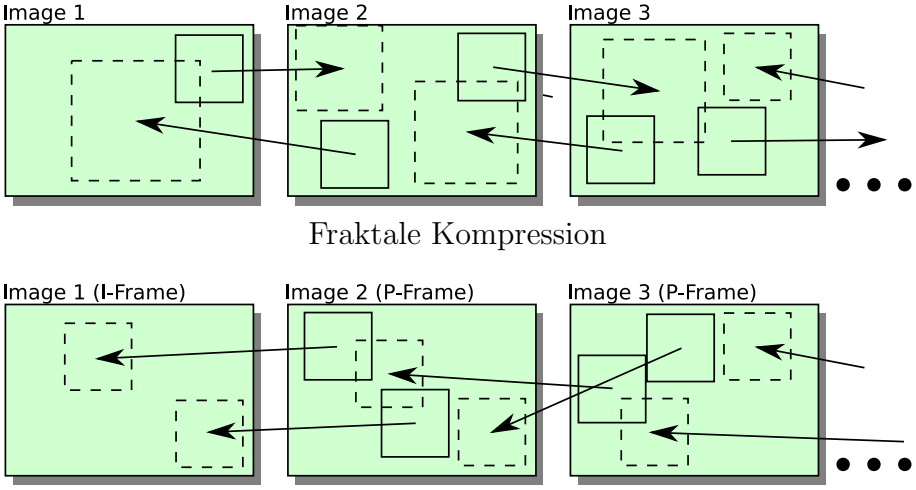


Abbildung 3.10: *Vergleich zwischen fraktaler Kompression und Videokompression: Bei Videos werden nur Verweise in einer Richtung verwendet, die zusätzlich auch nur auf Blöcke der gleichen Größe gerichtet sind. (Hinweis: Bei den in neueren Videocodecs zu findenden B-Frames werden Verweise auf* ***zwei*** *statt nur einem Frame zugelassen, die allerdings auch hier beide in der Kodierungsreihenfolge vorausgehen)*

Standardauflösungen etabliert haben (siehe Abbildung 3.11). D.h. es gibt in der Regel eine große Mengen an Bildern derselben Auflösung. Durch dieses vermehrte Auftreten von gängigen Auflösungen kann man eine Bilddatenbank mit geringem Aufwand derart aufteilen, so dass nur Bilder der gleichen Auflösung miteinander komprimiert werden.

Das andere angesprochene Problem ist allerdings schwerwiegender:
Klassische Implementierungen (z.B. [39]) schalten von den (Redundanzen zwischen Frames ausnutzenden) P-Frames und B-Frames zurück auf (nur die aktuelle Frame ohne Kontextinformationen modellierende) I-Frames, wenn die Veränderung zwischen zwei Frames zu groß ist.

Ein Video wird normalerweise mit einer zeitlichen Auflösung von 25 oder 29,97 Frames pro Sekunde aufgezeichnet. Demzufolge sind auch die Videocodecs darauf optimiert, Veränderungen zu beschreiben, die innerhalb von 0,03-0,04 Sekunden stattfinden. Ist die Veränderung zwischen zwei Bildern größer als typischerweise in so einem Zeitsegment möglich, so wird eine I-Frame Kompression verwendet, die bei den meisten Verfahren ähnlich wie JPEG oder JPEG2000 arbeitet, d.h. genau der Typ von Einzelbild-Verfahren

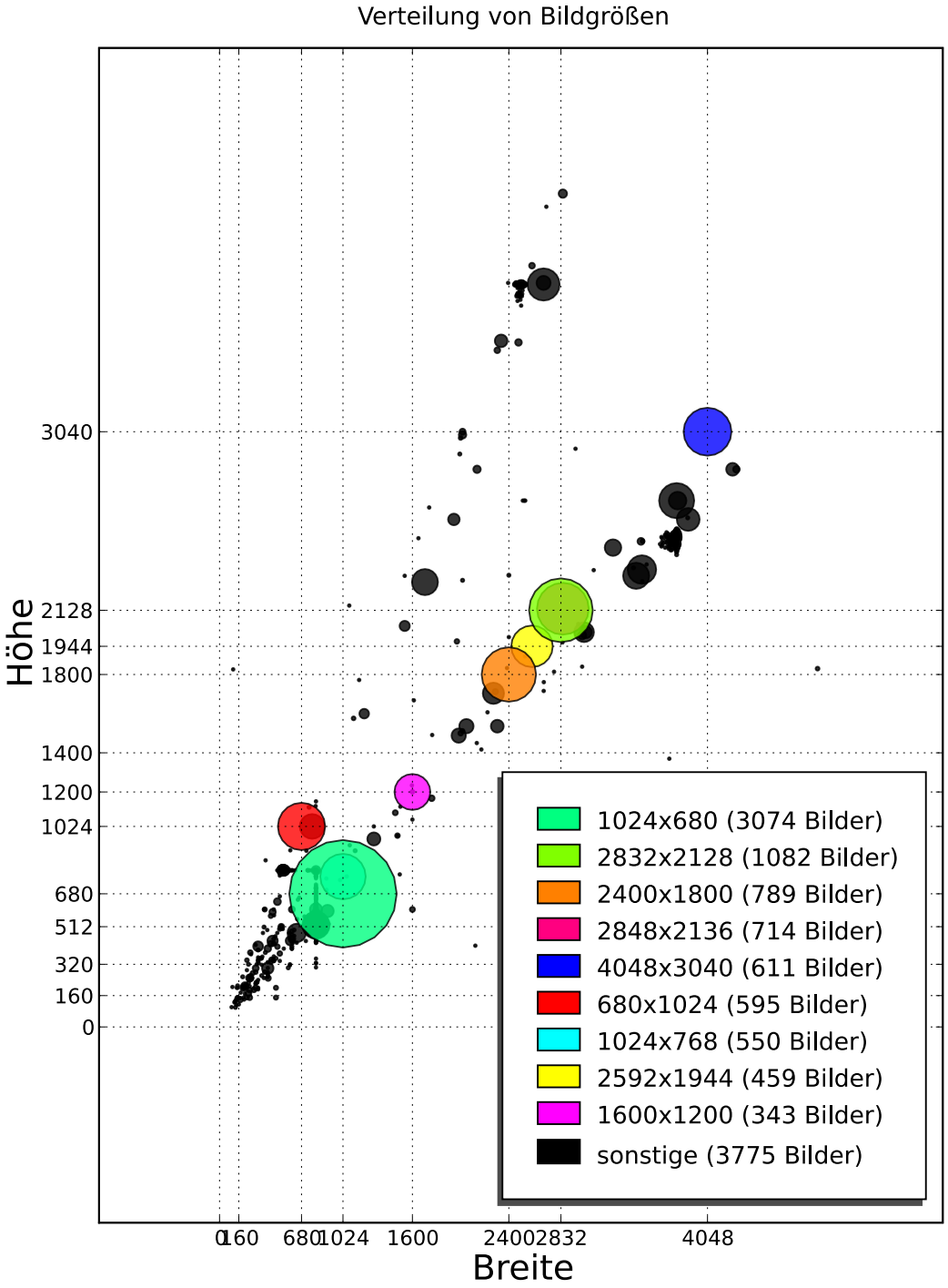


Abbildung 3.11: *Die verschiedenen Auflösungsstufen von Bildern am Beispiel der 19440 Bilder der mediaTUM-Bilddatenbank: Ein Großteil der Bilder sind in einem der Kamera- oder Bildschirm-Standardformate (wie z.B., in dieser Datenbank, 2832x2128 oder 1024x680) abgelegt.*

ist, von dem man sich bei der Bildmengen-Kompression eigentlich lösen will.

Tabelle 3.1 zeigt, inwieweit (visuell ähnliche) Bildpaare (Abbildung 3.12) ähnlich genug waren, um in einem Standard-Videocodec (h.263) eine „echte“ Videokompression (Kodierung eines der Bilder als P-Frame) zu erzeugen.

Ein weiteres Problem ist die Tatsache, dass die meisten Videocodecs nur auf sehr geringen Auflösungen arbeiten (selbst HDTV Codecs arbeiten „nur“ mit einer Auflösung von 1920x1152 [88]). Neuere Digitalfotos oder Scans sind aber mit einer um ein Vielfaches höheren Auflösung aufgenommen, so dass auch in einer typischen Bilddatenbank insbesondere die „lohnenswerten“ (sprich großen) Bilder jenseits von dem sind, was man mit normalen Videocodecs komprimieren kann.
Es ist zwar natürlich möglich, die Bilder einer Bilddatenbank so zu „zerschneiden“, so dass die Teilstücke im Bereich dessen liegen, was Videocodecs komprimieren können. Dies ist allerdings ineffizienter als eine Vollbild-Kompression.

Insgesamt scheint die Verwendung von Videocodecs zur Kompression von Bilddatenbanken aufgrund der genannten Nachteile (und der Tatsache, dass Videokompression in diesem Anwendungsgebiet aufgrund der Bauweise immer im Schatten der fraktalen Kompression stehen wird) eher eine Sackgasse, weshalb wir diese Art der Mehrbild-Kompression im folgenden nicht weiter vertiefen werden.

Bildpaar 1

Bildpaar 2

Bildpaar 3

Bildpaar 4

Bildpaar 5

Abbildung 3.12: *Eine Reihe von manuell ausgewählten visuell ähnlichen Bildpaaren aus einem Bildarchiv*

Bildpaar	Frametyp	Kompressionsgewinn
1	P-Frame	6272
2	I-Frame	0
3	I-Frame	0
4	P-Frame	4546
5	I-Frame	0

Tabelle 3.1: *Frametyp des zweiten Bildes sowie Kompressionsgewinn bei Zusammenfassung von zwei Bildern in einen h.263 Videostream*

3.3 Karhunen-Loeve auf Bildmengen

Wie in Kapitel 2.3.1 erläutert, errechnet der Einzelbild-KLT-Algorithmus für jedes Bild eine eigene Transformationsmatrix, die speziell für die Eigenschaften dieses Bildes optimiert wurde.

Das grundlegende KLT-Verfahren hat die folgende Form:

1. Extrahiere alle $n \times n$ Blöcke $b_1, b_2, \ldots \in \mathcal{R}^{n^2}$ aus einem Bild I
2. Berechne die Kovarianzmatrix $C \in \mathcal{R}^{n^2}$ der Block-Koeffizienten
3. Berechne die optimale Transformation $K \in \mathcal{R}^{n^2}$ aus C durch Lösen von $KCK^T = \Lambda$
4. Kodiere K sowie $K \cdot b_1, K \cdot b_2, \ldots$

Nachteile bei der Einzelbildkompression ist insbesondere die Größe dieser Transformationsmatrix (n^4, mit n die Kantenlänge der Bildblöcke), weshalb sich dieses Verfahren bisher nicht bewährt hat.

Bei der Kompression von Bildmengen ist die Tragweite des letztgenannten Problems allerdings deutlich kleiner:
Da es möglich ist, Statistiken von mehreren Bildern zusammenzufassen, so dass man zwar diese Bilder optimiert komprimieren kann, aber nur **eine** Transformationsmatrix abgespeichert werden muss, spart man sich den „Overhead“ der mehrfachen Speicherung dieser Transformationsmatrix:
Letztere wird dann aus den Bildblöcken von mehreren Bildern gleichzeitig errechnet, und für jede Bildmenge nur einmal abgespeichert, und fällt deswegen weniger in's Gewicht.

Die auf Bildmengen (aus jeweils m Bildern, mit m variabel pro Gruppe) erweiterte Form des oben angegebenen Verfahrens wird dann folgendermaßen beschrieben:

1. Extrahiere alle $n \times n$ Blöcke $b_{j,1}, b_{j,2}, \ldots \in \mathcal{R}^{n^2}, j = 1 \ldots m$ aus **allen** Bildern $I_i, i = 1 \ldots m$ einer Bildmenge
2. Berechne die Kovarianzmatrix $C \in \mathcal{R}^{n^2}$ der Block-Koeffizienten
3. Berechne die optimale Transformation $K \in \mathcal{R}^{n^2}$ aus C durch Lösen von $KCK^T = \Lambda$
4. Kodiere K, Λ sowie $K \cdot b_{11}, K \cdot b_{12} \ldots, K \cdot b_{21}, K \cdot b_{22} \ldots$ [3]

Die Größe der Matrix K hängt nur von der Blockgröße, nicht aber von m ab, und muss nur einmal pro Bildmenge gespeichert werden. Aus diesem Grund ist es natürlich wünschenswert, möglichst viele Bilder in einem zu komprimierenden Cluster abzulegen, um das Maximum an Kompression zu erreichen.

Auf der anderen Seite wird, je nachdem, was für Bilddaten man in den Ursprungsbildern hat, die aus den zusammengefassten Bildern generierte KLT-Matrix mit der Zeit gegen eine generalisierte Transformation konvergieren, die die speziellen Eigenheiten der Einzelbilder nicht mehr so stark berücksichtigt (Vergleiche die beiden konvergierten KLT-Matrizen in Abbildung 3.13 und Abbildung 3.14)
Man kann sogar beweisen [36], dass, wenn man genug statistische Daten zur Herleitung einer KLT Matrix verwendet, die resultierende Matrix gegen die DCT Matrix konvergiert.

Mit anderen Worten: Bei zu vielen (insbesondere vom Typ her verschiedenen) Bildern in einem Cluster stellt sich eine Art „Sättigung“ ein, die sich negativ auf das Kompressionsresultat auswirkt.

Im KLT-Cluster-Kompressionsverfahren spiegeln sich Korrelationen zwischen Bildern nur in der im Cluster gespeicherten Kovarianzmatrix bzw. KLT-Matrix, aber nicht in den tatsächlichen Daten. Das heißt insbesondere, dass

[3] Wie sich noch zeigen wird, ist es sinnvoll für jede Bildmenge nicht nur die KLT-Matrix, sondern auch die darauf basierende Kovarianzmatrix abzuspeichern. Letztere kann effizient abgelegt werden, indem man sie aus den Eigenwerten der KLT-Zerlegung berechnet: $\Lambda = KCK^T$

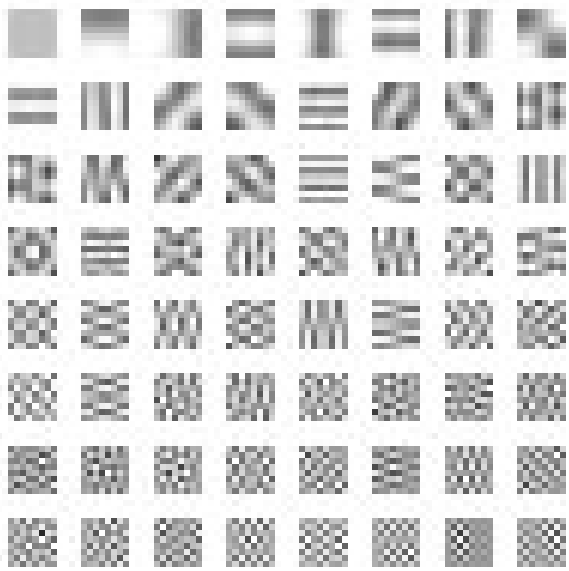


Abbildung 3.13: *Die Transformationsmatrix (Basisbilder) einer KLT von 4000 Fotos. Die Pixelwerte bei einem „Basisbild“ (a.k.a. “Eigenimage“) stammen aus jeweils einer Spalte der Matrix K.*

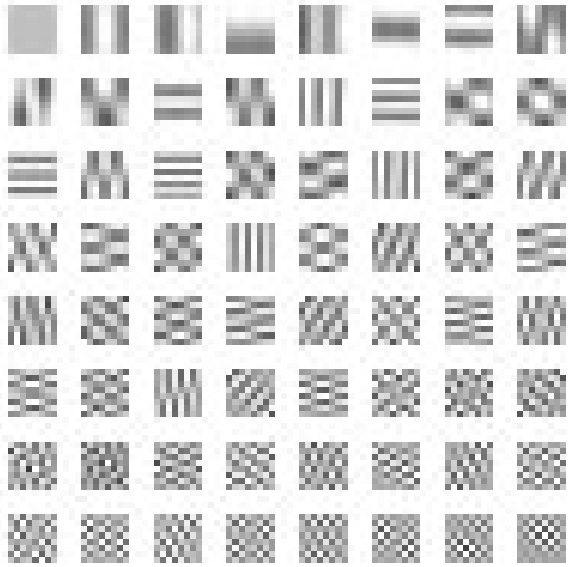


Abbildung 3.14: *Die Transformationsmatrix (Basisbilder) einer KLT von 4000 Textseiten*

einzelne Bilder jederzeit aus einem Cluster gelöscht oder in diesen hinzugefügt werden können ohne dass eine Neukomprimierung der restlichen Bilder notwendig ist.

3.3.1 Kompressionsdistanz von KLT

Zwei Bilder

Die klassische Methode, um herauszufinden wie gut sich zwei Bilder in einem Cluster zusammen komprimieren lassen, ist es, mit dem Verfahren aus 3.3 die Einzelbilder zu komprimieren, und danach beide Bilder in einem Cluster zusammenzufassen. (Sprich eine Probekompression durchzuführen) Dieses Verfahren ist gangbar, allerdings relativ rechenzeitaufwändig.

Optimierungsmöglichkeiten ergeben sich mittels Verfahren aus der Videokompression, in denen keine tatsächliche Kompression vorgenommen wird, sondern die vermutliche Kompressionseffizienz aus Statistiken der aus der Transformation hergeleiteten Koeffizienten (jedes Bildblocks) angenähert wird.

In der Videokompression wird dies verwendet, um einen optimalen Motion-Block zu finden, sprich einen Bildbereich, dessen Differenzkodierung zum gerade zu kodierenden Bildblock möglichst wenig Bits beansprucht.

Wenngleich das hier besprochene Einsatzziel unterschiedlich ist (Abschätzung der Bitanzahl der KLT-Komprimierung eines kompletten Bildes), so ist das Verfahren doch dasselbe: In beiden Fällen will man eine Approximation einer Koeffizientenkodierung ausrechnen.

Der Videokompressor MEncoder [39] benutzt hierbei unter anderem die folgenden Verfahren:

- **SATD**: „Sums of absolute total differences“- Berechnung der absoluten Summe $\sum_1^n |c_i|$ aller Koeffizienten $c_1, c_2, \ldots, c_n$
- **coeffmax**: Ermitteln des größten Koeffizienten $c = max(c_2, c_3, \ldots, c_n)$ (der „DC“ Koeffizient c_1 wird i.d.R. separat kodiert, und deswegen vorher abgeschnitten)
- **bitsum**: Summierung der Bitanzahl, d.h. Zusammenzählung der zum Abspeichern der Koeffizienten benötigten Bits (höchstes gesetztes Bit jedes Koeffizienten in der Dualdarstellung).

Diese Optimierungen können den Rechenaufwand einer Kompressionsgüte-Abschätzung deutlich reduzieren, allerdings müssen auch sie trotzdem zuerst die komplette KLT-Transformation durchführen, und durch alle Bildblöcke (sprich Koeffizienten) iterieren.

Deswegen soll hier nun noch ein Verfahren vorgestellt werden, mit dem die Kompressionsgüte direkt aus den Kovarianzmatrizen sowie der Transformationsmatrix der beiden Bilder berechnet wird:

Die Idee hierbei ist, dass eine optimale Transformationsmatrix daran erkennbar ist, dass die Kovarianzmatrix KCK^T der (transformierten) Blöcke (mit ursprünglicher Kovarianzmatrix C) möglichst nahe an der Einheitsmatrix ist, sprich die einzelnen Koeffizienten optimal dekorreliert sind.

Da für zwei Bilder I_1, I_2 sowohl K_1, K_2 als auch C_1, C_2 bekannt sind, können für eine neue Transformation K (Errechnet aus $K(C_1 + C_2)K^T = \Lambda$) die entsprechende Kovarianzmatrizen $C_{trans,1} = KC_1K^T$ und $C_{trans,2} = KC_2K^T$ auch durch einfache Matrixmultiplikation berechnet werden.

Das Verhältnis der absoluten Einträge der Diagonal und der Nebendiagonalelemente dieser beiden Matrizen gibt dann eine Abschätzung darüber, wie gut die Transformationsmatrix K für diese beiden Bilder arbeitet, siehe Abbildung 3.15.

Drei und mehr Bilder

Die Generalisierung auf drei oder mehr Bilder erfolgt, indem man die Methoden des vorangegangenen Abschnitts jetzt nicht mehr auf die Kovarianzmatrizen der einzelnen Bilder, sondern auf die Kovarianzmatrizen von Bildmengen anwendet.

Bei der Implementierung des Diagonal/Nebendiagonal-Abschätzung werden dann die folgenden Statistiken eingesetzt:

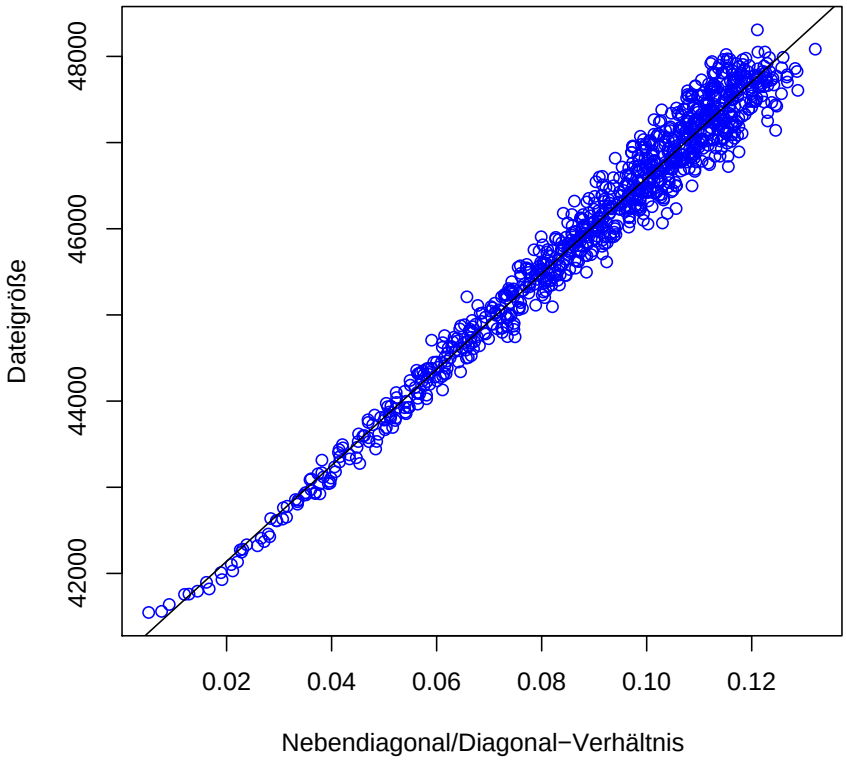


Abbildung 3.15: *Abschätzung der Kompressionsrate durch die Kovarianzmatrix: Für ein gegebenes Bild wurde die Kovarianzmatrix der Bildblöcke immer stärker mit Zufall angereichert, und danach zur Kompression verwendet. Sowohl die eigentliche Kompressionsrate (i.e., die Dateigröße von beiden Bildern) als auch das Verhältnis von Nebendiagonale gegenüber Diagonale wurde ermittelt und geplottet.*

A	B	Kompressionsmetrik	Matrizenmetrik
Bild	Bild	Komprimiere A, B mit gemeinsamen K_{AB} (Berechnet aus $C_A + C_B$)	Diagonal/Nebendiag. Verhältnis von $K_{AB}C_AK_{AB}^T$ + $K_{AB}C_BK_{AB}^T$
Bild	Bildmenge	Komprimiere A mit existierendem K_B der Bildmenge	Diagonal/Nebendiag. Verh. von $K_BC_AK_B^T$
Bildmenge	Bildmenge	Komprimiere alle Bilder (in A und B) mit K_{AB} (Berechnet aus $C_A + C_B$)	Diagonal/Nebendiag. Verhältnis von $K_{AB}C_AK_{AB}^T$ + $K_{AB}C_BK_{AB}^T$

Um eine Abschätzung der Kompressionseffizienz einer Bildmenge, zusammengefasst mit einer anderen Bildmenge, herzustellen, muss die gemeinsame Kovarianzmatrix C zusammen mit dem Cluster gespeichert werden. Dies benötigt allerdings nur $O(n)$ Speicherplatz, da letztendlich nur $\Lambda = KCK^T$, sprich die in der Diagonale von Λ liegenden Eigenwerte von C, gespeichert werden müssen.

3.4 PNG auf Bildmengen

Wie bereits in Kapitel 2.1.3 vorgestellt, teilt der PNG Algorithmus ein Bild vertikal in Zeilen auf, die er einzeln betrachtet, und mit einer von 5 Filterfunktionen bearbeitet. Die gefilterten Bilddaten werden dann durch den standardisierten LZ77 Algorithmus [22] verlustfrei komprimiert.

Dieses Verfahren basiert im wesentlichen auf zwei Prinzipien:

- Durch die Filterung werden Gradienten und homogene Flächen dekorreliert, und damit für die Kompression vorbereitet. Die Filter wirken auf die aktuelle Zeile, unter Einbeziehung der vorangegangenen Zeile.
- Durch die eigentliche Kompression werden sowohl Ähnlichkeiten zwischen Bildzeilen, als auch homogene gefilterte Bereiche (z.B. Gradienten, Gebiete mit wenig Helligkeitsschwankungen) reduziert. Die LZ77 basierte Kompression arbeitet hier üblicherweise mit einem Fenster von 32768 Bytes, d.h. bei einer Tiefe von 32 Bit (d.h. 4 Bytes pro Pixel) werden $\frac{32768}{4w} = \frac{8192}{w}$ horizontale Bildzeilen oberhalb der momentanen betrachtet, wobei w die Breite des Bildes darstellt.

Die hier vorzustellende Erweiterung dieses Verfahrens basiert auf der Tatsache, dass der LZ77 Codec durch das 32 Kilobyte große Fenster immer eine Reihe von vorhergehenden Zeilen zur Kompression der aktuellen Zeile „mitbenützen“ kann. (Das funktioniert natürlich am besten bei schmalen Bildern, sprich bei Bildern bei denen möglichst viele vorhergehende Zeilen in diesen Puffer passen)

Hierbei – und das ist die Idee – muss die vorhergehende Zeile aber nicht immer notwendigerweise aus dem aktuellen Bild sein. D.h. man kann den Algorithmus auf mehrere Bilder erweitern, indem man alle Bildzeilen aus allen Bildern einer Bildmenge in einer Art und Weise so übereinandergelegt, dass möglichst viele ähnliche Zeilen aus den verschiedenen Bildern übereinander zum liegen kommen (siehe Abbildung 3.16), und das resultierende Bild dann in gewohnter Weise PNG-komprimiert. Zusätzlich wird noch für jede Bildzeile abgespeichert, aus welchem Bild sie stammt.

Der Vorteil dieses Verfahrens ist, dass das zu komprimierende, aus mehreren Teilbildern bestehende, Bild wieder ein normales PNG ist, bei dem nur die Zeilen neu zu sortieren sind (das sieht man dem resultierenden Bild auch „an“, siehe Abbildung 3.17). D.h. dieser Algorithmus ist insbesondere

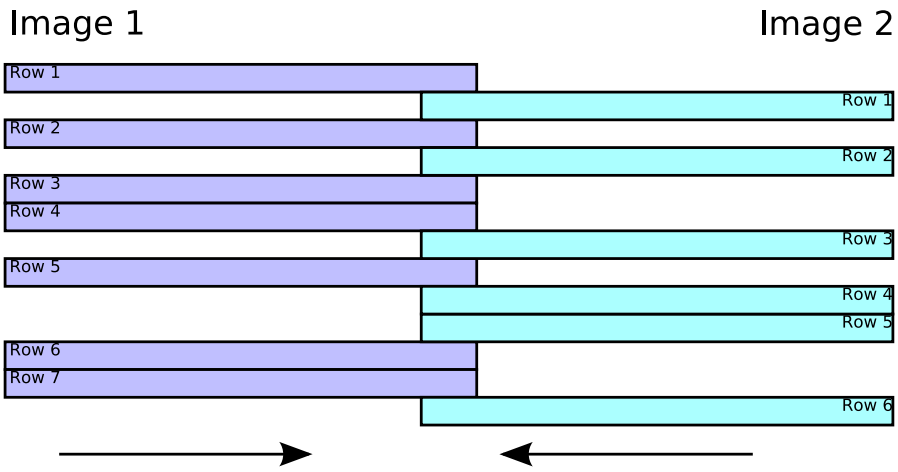


Abbildung 3.16: *Veranschaulichung des Mehrbild PNG Algorithmus: Durch Alternieren der Zeilen von zwei verschiedenen Bildern entsteht ein neues Bild*

für längerfristige Speicherung der Bilder interessant, also wenn sichergestellt werden muss, dass man auch zu einem Zeitpunkt in der Zukunft wieder in der Lage ist, die Bilder zu dekodieren — es werden ja keine „neuen“ Algorithmen eingeführt, sondern nur der bestehende PNG Algorithmus in transparenter Art und Weise erweitert.

Der Kern des Verfahrens liegt in der Auswahl der übereinander zu platzierenden Scanlines. Hierbei wird optimalerweise die durch die einzelnen Bilder definierte lineare Ordnung der Zeilen beibehalten, d.h. es werden zwar die Bildzeilen untereinander vermischt, aber nicht gegeneinander permutiert (siehe Abbildung 3.16). Dies hat vor allem zwei Gründe:

Berechnungszeit: Eine Permutation der Zeilen würde zu einer Explosion des Suchraumes führen, da es bereits bei einem einzelnen Bild der Höhe h ja insgesamt $h!$ mögliche Permutationen gibt.

Archivierungsaspekte: Wie bereits angesprochen, ist das Verfahren insofern „transparent“, da die Bilder mit einem Standardcodec dekomprimiert werden können, und man den dekomprimierten Bildern visuell ansieht, in welcher Art und Weise sie aus den verschiedenen Bildern zusammengestellt wurden, und es dementsprechend auch ohne Vorhandensein der Original-Spezifikation möglich ist, einen Dekompressions-

Abbildung 3.17: *Zwei Bilder (links), die durch den „Merge Scanlines“ Algorithmus zu einem größeren, besser komprimierbaren, Bild zusammengefasst wurden (rechts).*

Algorithmus zu formulieren. Dieser Vorteil würde durch eine (visuell nicht ohne weiteres zu erschließende) Permutation zunichte gemacht.

D.h. der resultierende Algorithmus fügt unter Beibehaltung der linearen Ordnung der Bildzeilen jedes Bildes bei zwei Bildern die h_2 Scanlines des zweiten Bildes (I_2) zwischen die h_1 Scanlines des ersten Bildes (I_1). Hierfür gibt es $k = \frac{(h_1+h_2)!}{h_1!h_2!}$ Möglichkeiten, oder allgemein bei n Bildern (Multinomialkoeffizient)

$$k = \begin{pmatrix} \sum_{i=1}^{n} h_i \\ h_1, h_2, \ldots, h_n \end{pmatrix} = \frac{(\sum_{i=1}^{n} h_i)!}{\Lambda_{i=1}^{n} h_i!}$$

Möglichkeiten.

Dieser Suchraum ist immer noch verhältnismäßig groß, weshalb eine weitere Vereinfachung sinnvoll erscheint: Anstatt alle möglichen Kombinationen zu betrachten, arbeitet man sich anhand eines Greedy-Algorithmus von einem lokalen Minimum $(y_1, \ldots, y_n)$, das die optimale Komprimierung für jedes Bild bestehend aus jeweils $y_k, k = 1 \ldots n$ Zeilen der Quellbilder darstellt, durch alle Bilder durch.

3.4.1 Implementierung

Das im vorigen Kapitel vorgestellte Verfahren wurde im Rahmen dieser Arbeit für den generischen Fall von n Bildern implementiert.

Hierbei wächst bei gleich großen Bildern der Kompressionsaufwand um den Faktor n^2, da beim resultierenden n mal größeren (höheren) Bild für jede Zeile n andere Bilder zu betrachten sind.

Der implementierte Algorithmus hat folgendes Aussehen:

Algorithm 1 MULTI-IMAGE-PNG($I_1, I_2, \ldots, I_n$)

```
final_width = max(width_1, width_2,..., width_n)
final_height = max(height_1, height_2,..., height_n)
for y ∈ 1, ..., final_height do
    for i ∈ 1, 2, ..., n do
        if y_i < height_i then
            z_i = best filter for current scanline y_i of image I_i
            l_i = compressed scanline (using z_i)
        end if
    end for
    i = argmin(len(l_i), i = 1 ... n)
    write l_i into output
    y_i = y_i + 1
end for
```

Dieser Algorithmus erzielte für ähnliche Bilder eines Testdatenbestands Kompressionsraten von 13.34 Bits pro Pixel, im Gegensatz zu den 13.37 Bits pro Pixel einer Einzelbildkompression. Die Kompression ist hierbei verlustfrei, dementsprechend ist diese Kompressionsrate nicht so imposant wie z.B. bei der erweiterten fraktalen Kompression.

Es wäre denkbar, den erweiterten PNG-Algorithmus z.B. durch die Einführung neuer Filterfunktionen (Unterschied zwischen Filterung gegen die vorhergehende Zeile des „eigenen“ Bildes gegenüber der zuletzt kodierten Zeile, etc.) weiter zu verfeinern und so sicher noch einiges an Kompressionsrate herauszuholen (das Byte, in dem die Filterfunktion abgespeichert wird, kann noch 250 bisher unbelegte Zustände annehmen) — allerdings ist der resultierende Algorithmus dann natürlich kein PNG mehr, was die bisher erwähnten Vorteile bzgl. Transparenz des Verfahrens zunichte machen würde.

Erwähnenswert ist ansonsten noch, dass die Bilder, die mit diesem PNG Algorithmus gewinnbringend komprimiert werden sollen, dafür einen gewissen Mindestgrad an Ähnlichkeit aufweisen müssen, wie in Abbildung 3.20 aufgezeigt wird.

3.4.2 Distanz-Relation zwischen Scanlines

Bei der Abschätzung der gemeinsamen Komprimierbarkeit der erweiterten PNG-Kompression greifen ähnliche Argumente wie bei den fraktalen Algorithmen: Wenn zwei Bilder bis zu einem bestimmten Maß gut zusammen komprimieren, so gilt das auch für die herunterskalierten Versionen, und umgekehrt.

Eine noch bessere Abschätzung erhält man allerdings, wenn man die Bilder nicht (wie bei der Abschätzung für die fraktalen Kompression) in beide Richtungen, sondern nur in die X-Richtung herunterskaliert. Hierbei ist die Überlegung, dass das Hauptkriterium bei der PNG-Komprimierbarkeit die Y-Position von ähnlichen (sprich: durch Y-Filter auf geringe Entropie abbildbaren) Bildzeilen ist. Zwei Bildzeilen sind aber auch dann noch ähnlich, wenn sie in X-Richtung zusammen geschoben wurden — und ihre Y-Position bleibt davon unbeeinflusst. Dieses Vorgehen ist in Abbildung 3.19 illustriert.

Möglicherweise kann man diese Metrik auch noch weiter optimieren, indem man verschiedene Statistiken (am öftesten auftretende Farben, Varianz, etc.) aus den Bildzeilen extrahiert und diese gegeneinander in's Verhältnis setzt. Allerdings generiert auch schon die auf horizontaler Herunterskalierung basierende Metrik zufriedenstellende Abschätzungen, siehe Abbildung 3.20.

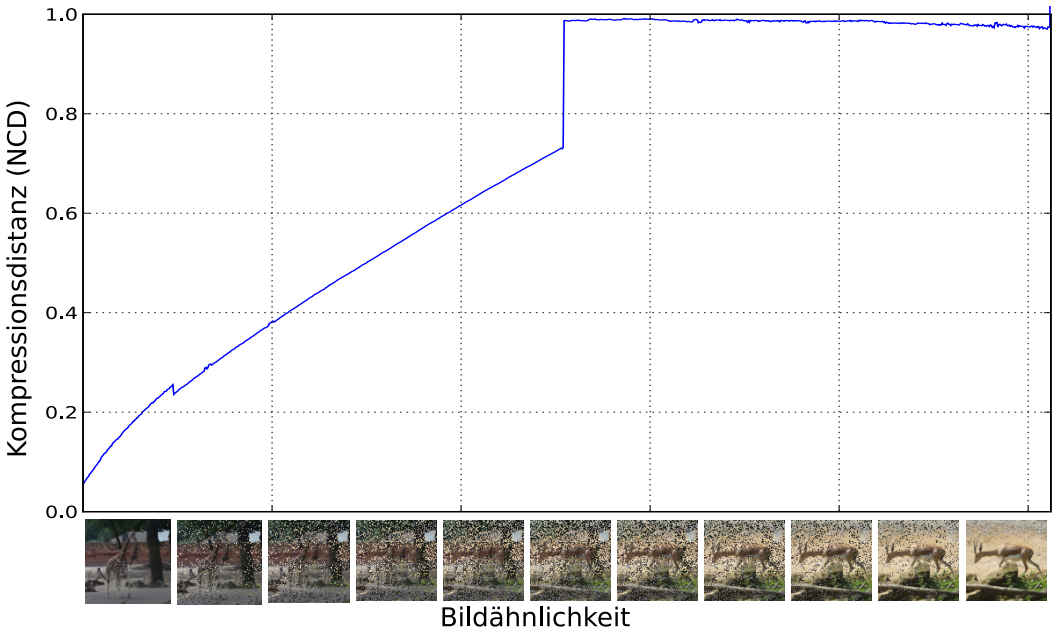


Abbildung 3.18: *Abhängigkeit der PNG-Komprimierbarkeit von Bildpaaren von der Bildähnlichkeit. Horizontal: Ähnlichkeitsmaß der Bilder (ganz links: zwei identische Bilder, ganz rechts: zwei unterschiedliche Bilder, dazwischen: Mischbild). Vertikal: Kompressionsdistanz der beiden Bilder. Interessant ist hier der Sprung bei ca. 50% Ähnlichkeit: Sind Bilder so unterschiedlich, dass die noch vorhandenen Ähnlichkeiten nicht mehr in den 32 Kilobyte Puffer von zlib passen, so springt die Filegröße plötzlich nach oben.*

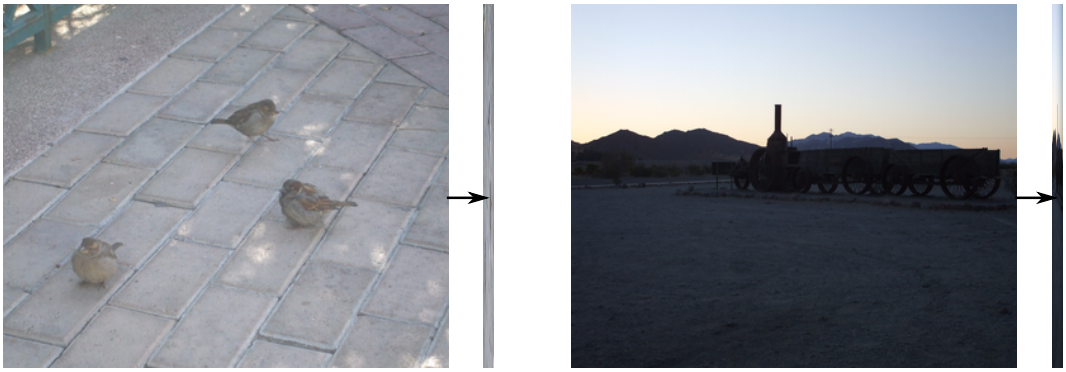


Abbildung 3.19: *Zwei Beispiele für die Generierung eines horizontal heruntergerechneten Bildes (jeweils rechts des Originalbildes) für die Abschätzung der gemeinsamen PNG Komprimierbarkeit.*

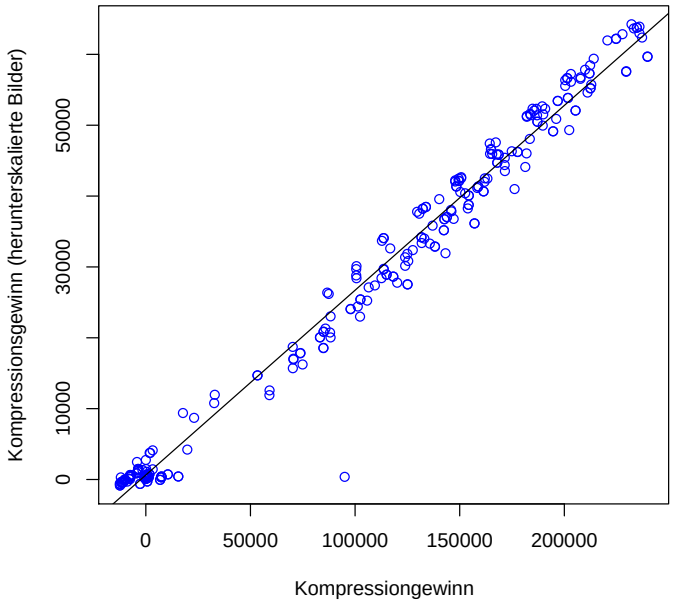


Abbildung 3.20: *Korrelation zwischen Kompressionseffizienz von Bildern normaler Auflösung (horizontal) gegenüber Bildern, die in der Breite skaliert wurden (vertikal). Jeder Punkt entspricht einem aus 60 Beispielbildern ausgewählten Bildpaar.*

3.5 Kompression von Bildmengen mittels Textursynthese und Segmentierung

Wie bereits in Kapitel 2.3.7 erläutert, kann man Bilder durch Segmentierung und anschließende Textur-Analyse komprimieren. Die hier existierenden Verfahren sind allerdings eher experimenteller Natur, und erreichen nur mäßige Kompressionsraten. Insbesondere durch perspektivische Einflüsse (siehe Bild 3.21) sind die Texturen innerhalb **eines** Bildes allerdings oft zu unähnlich, um aufeinander abgebildet zu werden bzw. um bei der Speicherung Informationen reduzieren zu können.

Allerdings ist, gerade bei Bildarchiven mit vielen ähnlichen Bildern ein und desselben Objekts, oftmals eine starke Redundanz bezüglich der verwendeten Texturen zwischen **mehreren** Bildern feststellbar.

Mit anderen Worten: Texturen in ähnlichen Bildern neigen dazu, durcheinander ersetzbar zu sein. Aufschlussreich sind beispielsweise die beiden Bilder in Abbildung 3.22, bei denen eine große Anzahl an Texturen in beiden Bildern vorkommen.

Deshalb liegt es nahe, texturbasierte Kompressionsverfahren auf Bildmengen auszuweiten. Ein derartiges Verfahren arbeitet dann vom Prinzip her folgendermaßen:

- Segmentiere alle zu komprimierenden Bilder eines Clusters in Texturen
- Fasse ähnliche Texturen zwischen den Bildern zusammen, und speichere sie in einer zentralen Texturdatenbank
- Speichere für jedes Bild Segmentierungsinformationen, Referenzen in die Texturdatenbank und Nicht-Textur-Bereiche.

Des Weiteren benötigt man noch eine Distanzfunktion zwischen Texturen (um in der Texturdatenbank in der Lage zu sein, Texturen zusammenzufassen), sowie eine Implementation einer effizienten Kodierung sowohl der Segmentierungsinformationen als auch der Texturdaten.

Im folgenden wird nun für jeden dieser Bereiche der hier eingesetzte Algorithmus motiviert sowie die Implementierung erläutert.

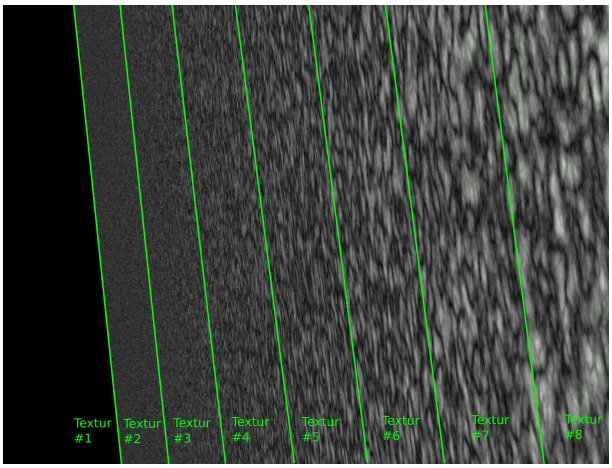


Abbildung 3.21: *Ein (künstliches) Bild mit nur einer Textur, die allerdings durch perspektivische Einflüsse in verschiedenen Versionen gespeichert werden muss*

Abbildung 3.22: *Zwei Bilder, mit einer Reihe ähnlicher Texturen: Das Gras, der Himmel, die unterschiedlichen Hauswände sowie der Wald im Hintergrund haben fast identische Textureigenschaften.*

3.5.1 Texturanalyse und -synthese

Um Texturen allgemeingültig zwischen Bildern zu beschreiben, bieten sich prinzipiell alle Familien von Texturanalyse an (siehe Kapitel 2.3.5). Allerdings ist es insbesondere für das Zusammenführen von Texturen sowie die nachfolgende, bildbereichsabhängige, Synthese von Texturen am günstigsten, Algorithmen auszuwählen, die die Textur in einer „geschlossenen“ Form betrachten. D.h. letztere sollte nicht anhand eines Beispielbildes, sondern anhand von Parametern beschrieben werden. Nur in dieser parametrisierten geschlossenen Form ist es möglich, auch „Mittelpunkte“ zwischen Texturen (sprich Texturen, die eine Mischung von zwei Texturbereichen zweier verschiedener Bilder sind) anzugeben.

Um Texturen in parametrisierter Form zu beschreiben, ist momentan das Verfahren von Portilla & Simoncelli [73] führend, das die folgenden Parameter verwendet, um eine Textur zu klassifizieren:

- Korrelation benachbarter Pixel auf verschiedenen Auflösungsstufen
- Kreuz-Korrelationen zwischen Auflösungsstufen unter Einfluss verschiedener Richtungsfilter
- Schräge und Kurtosis des Bildes in verschiedenen Auflösungen unter Einfluss verschiedener Richtungsfilter
- Varianz und Erwartungswert der ungefilterten Bilder sowie der Hochfrequenzanteile (Filter-Residuum).
- Pixelminima und -maxima des Bildes in der Originalauflösung.

Das genannte Modell ist sehr mächtig, allerdings (insbesondere bei kleinen Texturen) nicht sonderlich robust, und zudem auch ziemlich aufwändig zu programmieren.

Da es — je nachdem in wie kleine Bereiche die Segmentierungsfunktion das Bild zerlegt — nicht immer notwendig ist, allzu komplizierte Texturen zu beschreiben, ist es ohne nennenswerte Qualitätseinbußen möglich, nur einen Teil der oben genannten Parameter zu verwenden [55]. Insbesondere wurden beim vorgestellten Modell die Varianz- und Kreuzkorrelations-Statistiken durch eine Fourier-Statistik ersetzt. Neu hinzugekommen sind dafür zwei Parameter, die die Grundfarbe der Textur beschreiben:

- Radien der komplexen Fourierkoeffizienten der Textur
- Varianz, Schräge und Kurtosis (statistisches Moment zweiter, dritter und vierter Ordnung) sowie Pixelminima und -maxima des Bildes in der Originalauflösung.
- U und V Mittelwert einer YUV-Farbraum-Version des Bildes.

Eine Texturanalyse nach diesem Schema ist bereits in der Lage, relativ komplexe Texturen zu beschreiben bzw. synthetisieren — In Abbildung 3.23 sind ein paar Syntheseresultate von Texturen aus der (in diesem Forschungszweig relevanten) VisTex-Datenbank aufgeführt.

Die Synthese einer Textur funktioniert iterativ — zuerst werden die Radien der Fourierkoeffizienten angepasst, danach die verschiedenen Statistiken. Diese Prozedur wird solange wiederholt, bis die Textur konvergiert ist [73].

Die derart durchgeführte Textursynthese hat die angenehme Eigenschaft, dass ursprünglich nicht kachelbare Texturen (sprich Texturen, bei denen eine „Fuge“ sichtbar ist, wenn man die Kacheln seitlich aneinander legt) nach der Synthese kachelbar sind, also einen fließenden Übergang aufweisen.

Die Ursprungsdaten für die Texturanalyse stammen aus rechteckigen Bereichen, die aus den (in Bezug auf Texturen als homogen angesehenen) Bildsegmenten extrahiert wurden, siehe Abbildung 3.24.

Mit dem verwendeten Modell kann man also Texturen durch Kodieren von wenigen Parametern beschreiben, gleichzeitig mehrere Texturen zu einer „Mischtextur“ zusammenfassen, sowie mit kleinen Ursprungstexturen große (kontinuierliche) Flächen füllen.

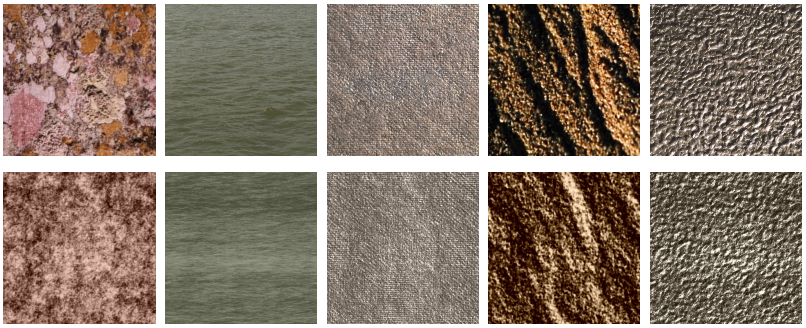


Abbildung 3.23: *Eine Anzahl von Texturen (oben), die durch den in dieser Arbeit vorgestellten Algorithmus neu synthetisiert wurden (unten).*

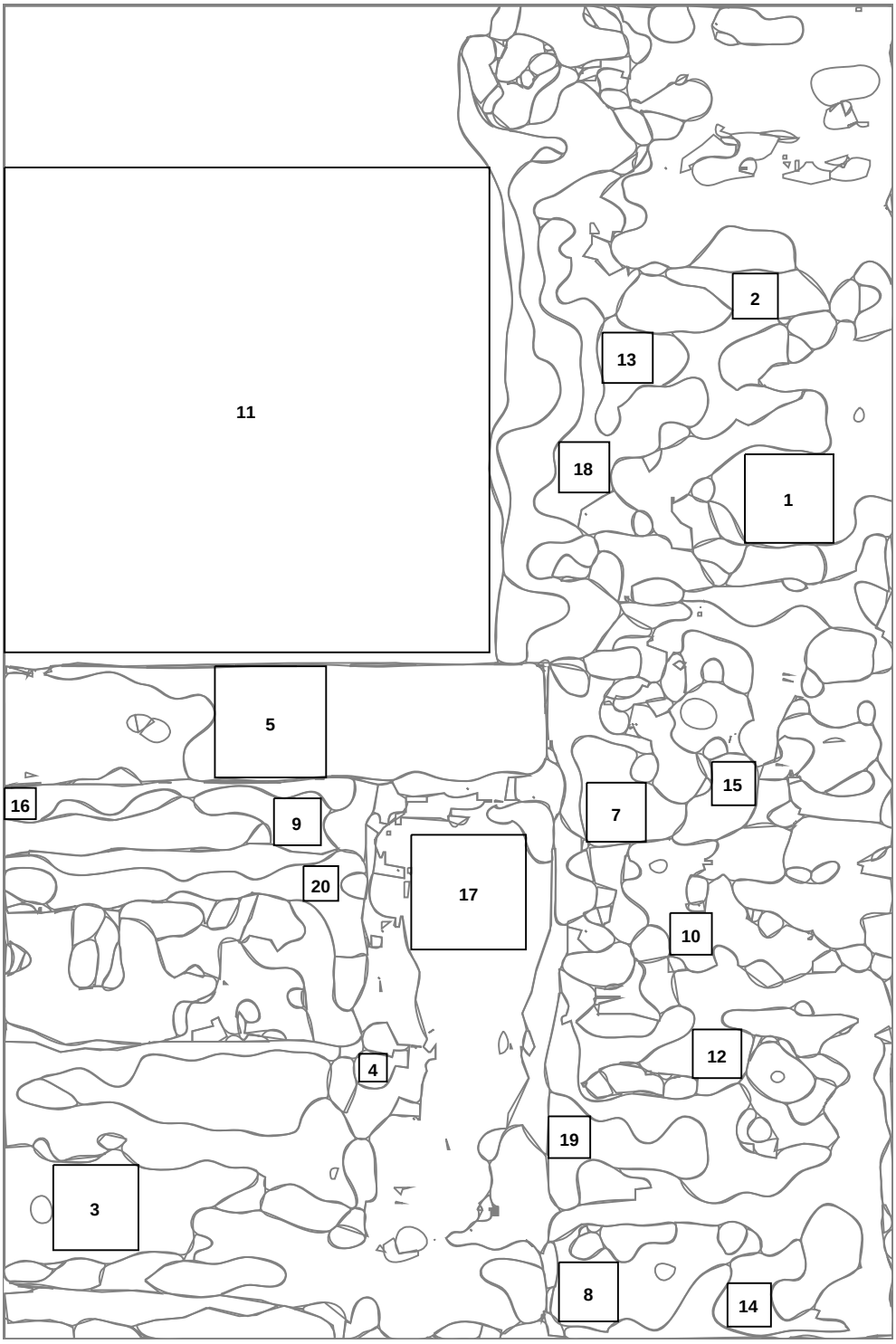


Abbildung 3.24: *Extraktion von Texturen aus Bildregionen durch Einpassen von Quadraten maximaler Größe*

3.5.2 Segmentierung

Da die Segmentierung der einzelnen Bilder nicht (wie Bildsegmentierung sonst) visuell motiviert ist, sondern nur möglichst gute Resultate in der nachfolgenden Kompression erzielen soll, macht es Sinn, nicht die üblichen Segmentierungs-Kriterien (wie z.B. durch die V1 Zellen der Augen in der Neurophysiologie motivierte Statistiken, siehe [111]) zu betrachten, sondern stattdessen ein abstrakteres Kriterium, was möglichst nah an die bereits eingeführten Definition der Texturparameter angenähert ist.

Wie bereits im vorhergehenden Kapitel besprochen, ist die hier verwendete Texturdefinition durch die Radien der komplexen Fouriertransformation sowie diverse Pixelstatistiken (Minimum, Maximum, statistische Momente zweiter, dritter und vierter Ordnung) definiert. D.h. eine optimale Segmentierung fasst demzufolge Bereiche zusammen, die in Bezug auf diese Statistiken möglichst homogen sind.

Da die genannten Statistiken sich auf Bildbereiche beziehen, und nicht auf einzelne Pixel, man allerdings für die Segmentierung eine pixelgenaue Positionierung benötigt, berechnet man diese Statistiken klassischerweise auf kleinen Bildbereichen.

Ein Großteil von klassischen Bildsegmentierungs-Algorithmen erzielt dies durch exponentiell abfallende Gewichtung der Nachbarpixel rund um den zu analysierenden Pixel. D.h. um die an einem einzelnen Pixel anliegenden Frequenzen (sprich die Fourier-Radien) zu messen, benutzen diese Verfahren Gabor-Filter mit den entsprechenden Frequenzanteilen. (a.k.a. „Fensterfouriertransformation“) Eine Abschätzung der Stärke F_{xy} der in der lokalen Umgebung U eines Pixels p_{xy} vorherrschenden Frequenzen f_x und f_y ist damit dann gegeben durch die Gabor-Faltung

$$\hat{p}(x, y, f_x, f_y) = \left| \int_{(u,v)\in U} exp(-\delta(u^2 + v^2) + 2\pi i(f_x u + f_y v))p_{x+u,y+v} \right| .$$

Der Vorteil dieser Funktion ist, dass ihre Primärkoeffizienten auch im Fourier-Spektrum auf einen lokalen Bereich konzentriert sind, was u.A. nützlich ist, die Berechnung zu beschleunigen [29].

Allerdings hat die exponentielle Gewichtung den Nachteil, dass umliegende Pixel auf unbestimmte Art und Weise in die Analyse des zentralen Pixels eingehen. Dies ist insbesondere bei Kanten innerhalb des Bildes ein Problem,

an denen sich verschiedene Texturen „treffen“, und deswegen dann „Fremdtexturen“ Eingang in die Statistik-Berechnung des lokalen Pixels finden[4].

Gerade bei der Textursegmentierung ist es deshalb sinnvoller, die Bereiche, aus denen Texturdaten extrahiert werden, durch eine einfachere Funktion zu beschreiben. Im vorliegenden Fall wird dies durch eine einfache Stufenfunktion erzielt, sprich die Textur an einem Punkt wird durch Extraktion von Parametern aus einem rechteckigen umliegenden Bereich (mit homogener Gewichtung) vorgenommen. Die Parameter, die aus diesen rechteckigen Bereichen extrahiert werden, orientieren sich am vorliegenden Texturmodell.

Die eigentliche Zuweisung eines Pixels zu einer Textur funktioniert dann durch Ermitteln aller Bereiche, von denen er ein Teil ist, und die entsprechende Zuweisung der Textur, die in selbigen am häufigsten vorkommt (siehe Abbildung 3.25).

Dieses Verfahren funktioniert überraschend gut, und erkennt sogar selbständig Kanten zwischen Texturen, ohne Bedarf an einer expliziten Kantenanalyse des Bildes zu haben [54].

Die Berechnung der verschiedenen Pixelstatistiken (Minimum, Maximum, Varianz etc.) für die unterschiedlichen Bildbereiche kann relativ schnell anhand der „Integral-Image“ Datenstruktur vorgenommen werden. [99] Für die Berechnung des Mittelwertes eines Parameters an einer bestimmten Stelle werden dann nur 2 Additionen und eine Subtraktion benötigt. Hierbei wird für jeden Parameter ein „Integral Image“ angelegt.
Die Berechnung der lokalen Fourierkoeffizienten (sprich die Faltung mit lokalisierten Frequenzfunktionen) kann effizient durch Vorberechnung einer Fourier-Transformation der Filter, punktweiser Multiplikation selbiger mit einer (ebenfalls vorberechneten) Fourier-Transformation des Bildes sowie einer inversen Fourier-Transformation berechnet werden.

Die eigentliche Implementierung des Algorithmus gestaltet sich relativ einfach. Allerdings ist die genaue Gewichtung der verschiedenen Texturparameter gegeneinander noch ein offenes Problem, das im vorliegenden Fall von Hand optimiert wurde. Hier gibt es vielversprechende Ansätze die solche Parameter anhand von genetischer Programmierung optimieren, unter Zuhilfenahme von handsegmentierten Bildern als „Lerndatenbestand“ [2].

Ein letztes Problem stellt noch die Anzahl der Texturen dar, in die man

[4] U.A. gibt es deswegen auch Segmentierungsverfahren, die versuchen, bei der Segmentierung Kanten explizit „herauszuschneiden“ [64].

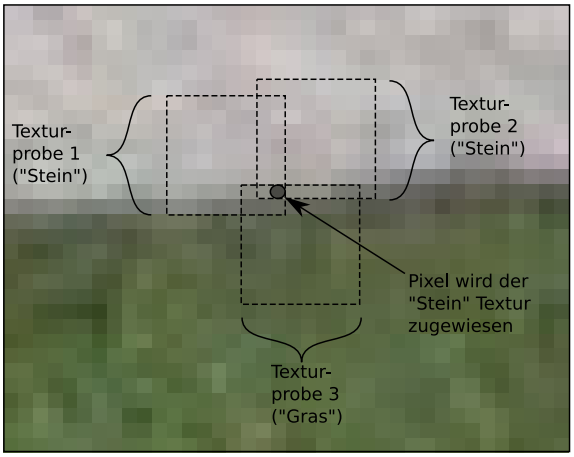


Abbildung 3.25: *Zuweisung von Pixeln zu Texturen.*

das Bild zerlegen möchte — dies ist ein Parameter, den der Segmentierungs- (bzw. Clustering-) Algorithmus von Anfang an benötigt. Klassische Segmentierungsverfahren verwenden hier relativ viel Zeit auf die Ermittlung einer entsprechenden Anzahl [14] [109]. Allerdings ist der hier vorgestellte Algorithmus gegenüber der Texturflächenzahl relativ robust, weshalb es in dem hier besprochenen Fall auch ausreicht, den Parameter anhand der Bildgröße fest zu definieren und die resultierenden Flächen gegebenenfalls noch einmal zusammenzufassen.

3.5.3 Distanz-Relation zwischen Texturen

Im Gegensatz zu den bisher besprochenen Cluster-Kompressionsalgorithmen werden beim hier besprochenen Kompressionsverfahren nicht die Bilder als solches in Ähnlichkeitsgruppen zusammengefügt, sondern nur die Texturen eines Bildes.

Das heißt, auch die Distanzfunktionen sind zwischen den Texturen, nicht zwischen den Bildern, definiert.

Hierbei kann man sich die Tatsache zunutze machen, dass die Texturen bereits in Parameter zerlegt sind, von denen bekannt ist dass sie visuell wichtig sind. D.h. eine Distanzfunktion zwischen Texturen wird durch eine Summe der (gewichteten) Abstände der einzelnen Parameter definiert.

Experimente mit der Bildqualität scheinen hier darauf schließen zu lassen

dass manche Parameter, insbesondere Helligkeit sowie Farbgebung hier stärker zu gewichten sind als beispielsweise Frequenzanteile und Statistiken 3. oder 4. Ordnung (Schräge, Kurtosis).

Hat man die Distanzen zwischen Texturen auf diese Art und Weise definiert, so lassen sich Texturen zwischen verschiedenen Bildern nun zusammenfassen, womit (verlustbehaftete) Kompression erreicht wird, siehe Abbildung 3.26.

Dieses Verfahren scheint insbesondere gut bei Landschaftsbildern zu funktionieren, die viele „sanfte“ Übergänge zwischen Texturen haben, und in denen durch die Textur als solches wenig Kontext vermittelt wird.

3.5.4 Exkurs: Nachkompression von JPEG Bildern mittels Texturen

Wie bereits in den vorhergegangenen Kapiteln erörtert, ist das hier besprochene Verfahren zwar sehr leistungsfähig, aber auch hochgradig verlustbehaftet: Komprimierte Bilder sehen möglicherweise komplett anders als ihre „realen“ Gegenstücke aus. D.h. Texturkompression ist primär aus Forschungssicht interessant, hängt bzgl. realem Nutzen aber anderen Kompressionsverfahren hinterher.

Interessanter ist deswegen u.A. der Punkt der „Nachkompression“. Hat man bereits eine Reihe von Bildern, die mittels eines Standardverfahrens (wie z.B. JPEG) komprimiert sind, so lassen sich theoretisch mittels Texturdaten die Kompression dieser Bilder verbessern, ohne die Originaldaten zu verändern. D.h. die **zusätzliche** Kompression ist verlustfrei. (Die ursprünglichen Daten liegen dabei dann bereits in einer verlustbehaftet komprimierten Variante vor, die von nachfolgenden Codecs nicht verändert wird) Um sich klar zu machen, warum dies möglich ist, ist es notwendig, nochmals auf die Fouriereigenschaften einer Textur einzugehen. Wie bereits in den vorhergegangenen Kapiteln angesprochen, sind bei bestimmten Texturen manche Gruppen von Fourierkoeffizienten stärker als andere. (Der hier vorliegende Algorithmus betrachtet ja den quadratischen Mittelwert bestimmter Fourier-Koeffizientengruppen als einen Modellparameter)

Gleichzeitig ermittelt aber die JPEG Kompression auch ihre Koeffizienten aus einer Variante der Fouriertransformation, der sogenannten DCT (Discrete Cosine Transformation). Diesen Koeffizienten liegt zwar ein anderes Berechnungsschema als denen der Textur zugrunde (die JPEG Koeffizienten

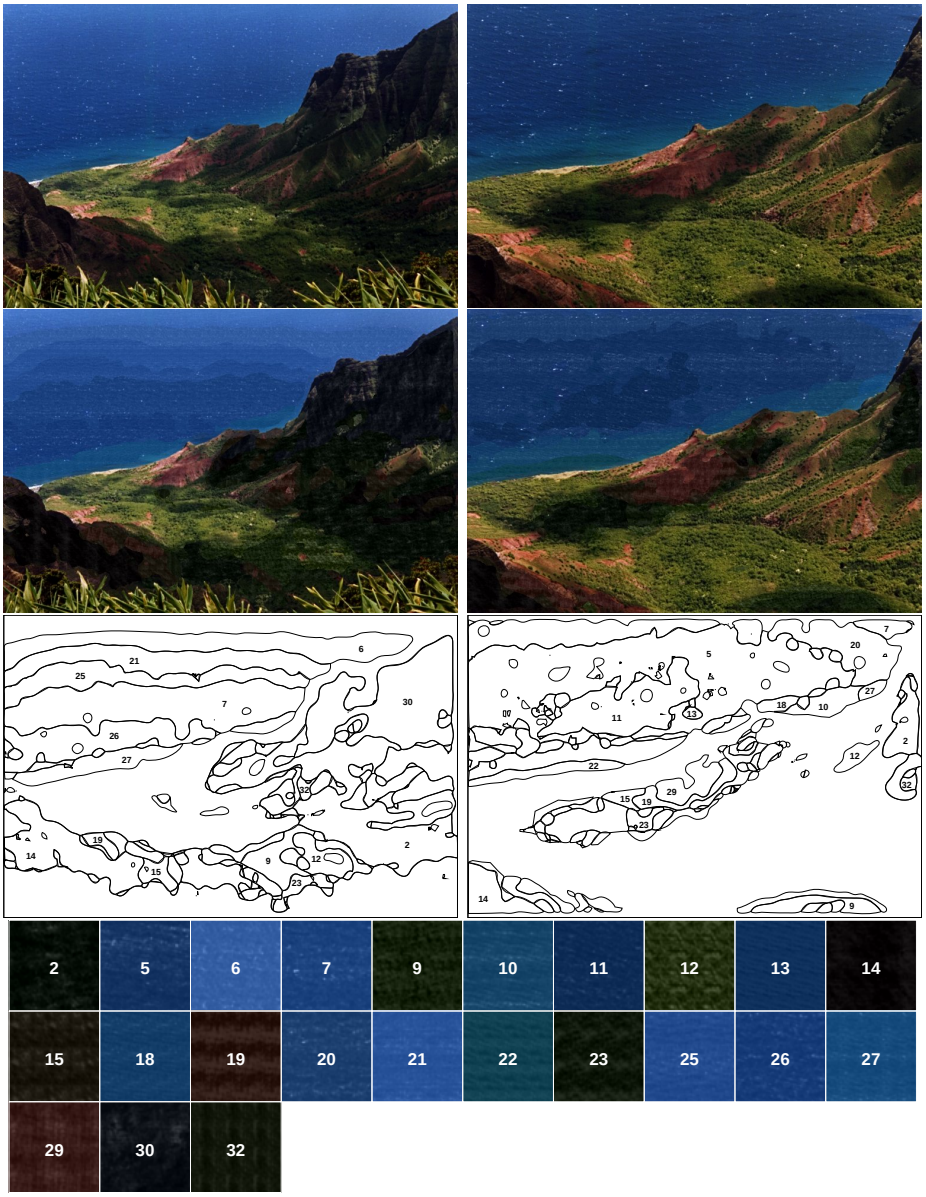


Abbildung 3.26: *Beispiel von zwei Bildern, die eine Reihe von Texturen teilen. Erste Zeile: Originalbilder. Zweite Zeile: "Komprimierte" Bilder (mit ersetzten Texturen), dritte Reihe: Zuweisung der Texturen zu den Bildregionen (Bildregionen ohne Zahl wurden nicht durch eine Textur ersetzt). Vierte Zeile: Die in beiden Bildern benutzten Texturen.*

beziehen sich nur auf einen relativ kleinen 8×8 Block, und sind auch nicht komplex sondern nur real), nichtsdestotrotz erzeugt die Textur, in der einer dieser JPEG Blöcke komprimiert ist, einen Wahrscheinlichkeitsraum für jeden Parameter, siehe Abbildung 3.27. D.h. bestimmte JPEG Parameter sind je nach Textur wahrscheinlicher als andere- d.h. wenn die Textur bekannt ist, in der man sich gerade befindet, so lässt sich das statistische Modell für die JPEG-Parameter optimieren. Allerdings muss dazu für jeden 8x8 Block die verwendete Textur gespeichert werden — was u.U. mit einigen Bits zu Buche schlägt und deswegen den eigentlich Kompressionsgewinn zunichte machen kann.

Inwieweit diese Nachkompression von JPEG-Bildern möglich ist, ist deswegen noch eine offene Frage. Nichtsdestotrotz handelt es sich aber nach Meinung des Autors hier um ein sehr vielversprechendes Forschungsgebiet.

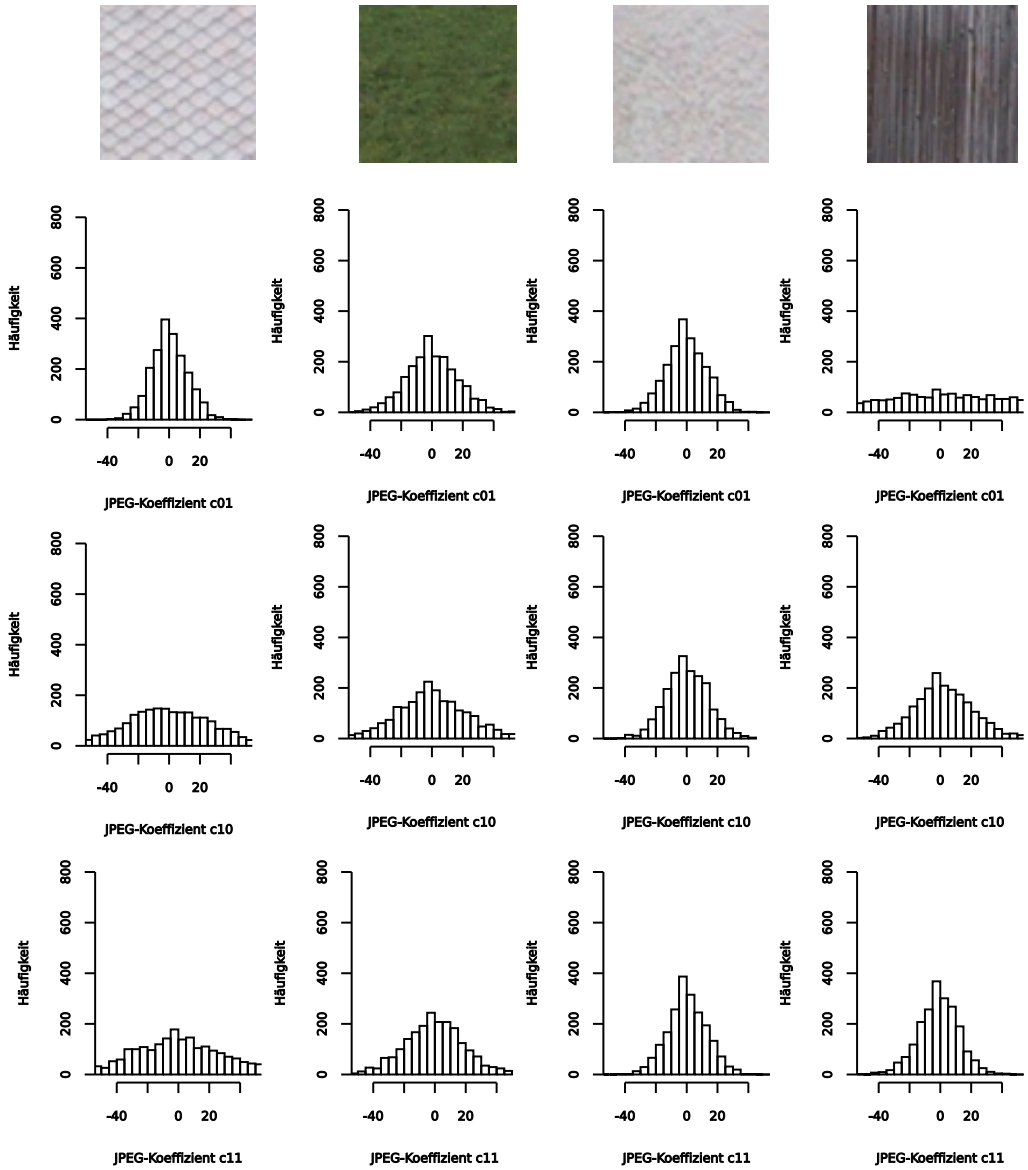


Abbildung 3.27: *Die Auftretenshäufigkeit von bestimmten Werten in den JPEG-Koeffizienten, die aus mehreren synthetisierten Texturen (oben) der Größe 362x362 gewonnen wurden. Jede Spalte entspricht einer Textur, jede Zeile einem Parameter. Je nachdem, welche Textur komprimiert wurde, haben die verschiedenen JPEG-Parameter unterschiedliche Auftrittswahrscheinlichkeiten.*

3.6 Eigenschaften der neuen Kompressionsalgorithmen

In diesem Kapitel wurde versucht, eine möglichst vielschichtige Auswahl von Algorithmen vorzustellen, die verschiedene Arten von Redundanz zwischen Bildern ausnützen, und unterschiedliche Eigenschaften und Zielsetzungen haben. Auch hat jedes Verfahren seine eigenen Vorteile und Nachteile.

So ist der Implementierungsaufwand für einen auf der erweiterten PNG-Kompression basierenden Dekompressor sehr gering, und das Verfahren ist, da es auf einem Standardverfahren basiert, auch zukunftssicher und ohne die Nachteile eines „proprietären“ Verfahrens (und gleichzeitig auch verlustlos), hat aber das Problem dass man bei der Dekompression immer **alle** zusammen komprimierten Bilder wieder dekomprimieren muss, auch wenn man nur Zugriff auf eines dieser Bilder benötigt, und bietet nur marginal bessere Bitraten als die normale PNG-Kompression.

Die verlustbehafteten Verfahren der Karhunen-Loeve sowie fraktalen Kompression erzielen im Gegensatz dazu deutlich höhere Kompressionsraten als ihre Einzelbild-Verwandten, allerdings benötigt man für das fraktale Verfahren sehr viel Rechenzeit (und muss auch wie bei der PNG-Kompression immer mehrere Bilder dekomprimieren, um Zugriff auf eines davon zu erhalten). Zudem eignet sich Karhunen-Loeve (bei handhabbaren Blockgrößen) nur für Bildmengen aus wenigen (kleinen) Bildern, da sonst ein Sättigungseffekt der Eigenbildmatrix K eintritt.

Die texturbasierte Kompression hat keinen der bisher genannten Nachteile, ist allerdings in einer Art und Weise verlustbehaftet, die bisher in der normalen Bildverarbeitung noch nicht genau genug analysiert wurde, was es schwierig macht, in automatischer Art und Weise einen Bildqualitäts-Parameter festzulegen.

Aufgrund der genannten Unterschiede sollte beim Einsatz der vorgestellten Algorithmen auf einer realen Bilddatenbank genau abgewogen werden, welcher Algorithmus bei welchen Bildmengen am meisten Sinn macht, weshalb in der in Kapitel 5.1 vorgestellten Anwendung in einem Bildverwaltungssystem mehrere der Algorithmen verwendet wurden.

Kapitel 4

Kompressionsoptimierung durch Bildgruppensuche

Es ist von Bildmenge zu Bildmenge unterschiedlich, wieviel Ähnlichkeiten, sprich Redundanz, **zwischen** Bildern existiert.
So gibt es für jedes Kompressionsverfahren Bildgruppen, die sich stark unterscheiden, und deswegen mit den Verfahren aus dem letzten Kapitel kaum gewinnbringend(er) komprimiert werden können, oder solche, die sehr ähnlich sind, und deswegen sehr gut miteinander komprimieren.

Gleichzeitig gibt es für die meisten der vorgestellten Verfahren eine gewisse Optimalanzahl von Bildern, für die der Algorithmus am besten angewandt werden sollte— diese orientiert sich am Kompressionsalgorithmus, der eingesetzten Hardware und der gewünschten Kompressionsrate. Diese Größe ist typischerweise in der Größenordnung von zwischen ca. 10 und 50 Bildern. Eine typische Bilddatenbank besteht aber üblicherweise aus tausenden von Bildern, weshalb letztere möglichst gut in günstige Gruppen aufgebrochen werden müssen, um mit den besprochenen Verfahren komprimiert zu werden.

Dieses und das folgende Kapitel beschäftigt sich nun deswegen mit eben dieser Suche von Bildgruppen (Clustern) von handhabbarer Größe, die in Bezug auf ein bestimmtes Kompressionsverfahren möglichst gut zueinander passen, sprich möglichst viele durch ein gegebenes Verfahren ausnutzbare Redundanzen enthalten.

Dazu wird zuerst im nachfolgenden Unterkapitel darauf eingegangen, inwie-

weit man zwei Bilder (zum Zwecke der Bildung von Clustern) möglichst gut miteinander „vergleicht“, im darauf folgenden Kapitel werden dann konkrete Clustering-Algorithmen vorgestellt.

4.1 Kompressions-Distanzmetriken

Die bisher vorgestellten auf Bildmengen erweiterten Kompressionsalgorithmen berechnen für eine Menge von Bildern eine komprimierte Datei mit einer bestimmten Größe. Je größer diese Datei (im Verhältnis zu der Größe der Originalbilder) ist, desto schlechter konnte der entsprechende Algorithmus die Bildmenge komprimieren. D.h. die Größe der entstandenen Datei ist gleichzeitig ein Maß dafür, wie gut zwei oder mehr Bilder „zusammen komprimieren“.

Natürlich muss hier aber berücksichtigt werden, wie gut die beiden Bilder alleine komprimieren. D.h. die „Güte“ einer Bildmenge hängt sowohl von der Dateigröße der komprimierten Bildmenge als auch von den Dateigrößen der (einzeln) komprimierten Einzelbilder ab.

Schreibt man die komprimierte Dateigröße eines Einzelbildes $I \in \mathcal{I}$ als C_I, und die Dateigröße einer komprimierten Bildmenge $I_1, I_2, \ldots, I_n \subset \mathcal{I}$ als $C_{I_1,I_2,\ldots,I_n}$, so kann man eine Formel für die Qualität eines aus zwei Bildern bestehenden Clusters z.B. folgendermaßen konstruieren:

$$d(I_1, I_2) = \frac{C_{I_1,I_2}}{C_{I_1} + C_{I_2}}$$

Je kleiner der Wert von $d(I_1, I_2)$, desto kleiner ist die Dateigröße der zusammen komprimierten Bilder I_1 und I_2, d.h. desto „ähnlicher“ sind die Bilder.

Eine derartige Funktion $d(I_1, I_2)$, die zwei Objekte vergleicht, nennt man auch „Metrik“— mathematisch hat eine Metrik (angewandt auf die hier verwendeten Daten, sprich Bilder) die folgenden Eigenschaften:

1. $d(x, x) = 0$ (Identische Bilder haben den Abstand 0)
2. $d(x, y) = d(y, x)$ (Symmetrie)
3. $d(x, z) \leq d(x, y) + d(y, z)$ (Dreiecksungleichung)

Die oben konstruierte Funktion erfüllt nun die Anforderung der Symmetrie, allerdings nicht die des Abstandes 0 zwischen identischen Bildern.

Es gibt allerdings eine Variante namens NCD („normalized compression distance“), die alle obigen Anforderung erfüllt [18]:

$$d(I_1, I_2) = NCD_{I_1,I_2} = \frac{C_{I_1,I_2} - min(C_{I_1}, C_{I_2})}{max(C_{I_1}, C_{I_2})} \tag{4.1}$$

mit $\frac{0}{0} := 0$.

Die NCD hat die Eigenschaft, für identische Bilder den Abstand 0 zurückzuliefern, und liefert des Weiteren für maximal unterschiedliche Bilder (sprich Bilder bei denen gilt: $C_{I_1,I_2} = C_{I_1} + C_{I_2}$) genau Abstand 1.

Im folgenden bezeichnen wir eine derartige Metrik als „Metrik auf gewichteten Graphen“, da man jeweils zwei Bilder I_1, I_2 als Knoten in einem Graph auffassen kann, und dementsprechend die Distanz $d(I_1, I_2)$ als Gewicht der Kante zwischen I_1 und I_2. Es gibt unter Berücksichtigung der Symmetrie bei n Bildern dann $\frac{n\cdot(n-1)}{2}$ solche Distanzen $d(I_i, I_j)$ mit $i \in 1 \ldots n, j \in 1 \ldots n$

Manche Clusteringalgorithmen interpretieren eine Kante mit hohem Gewicht so, dass zwei Bilder sehr ähnlich sind, anstatt (wie bei der NCD) sehr unähnlich. Deswegen ist es teilweise notwendig, die Metrik zu invertieren- Bei der NCD kann man dies recht einfach als

$$\hat{d}(I_1, I_2) = 1 - d(I_1, I_2)$$

formulieren. Hierbei ist $\hat{d}$ dann natürlich **keine** mathematische Metrik mehr.

Erweitert man das Modell des „gewichteten Graphen“ derart, dass nicht nur zwischen jeweils zwei Bildern I_1, I_2, sondern zwischen beliebigen Bildmengen $I_1, I_2, \ldots, I_n \in \mathcal{I}$ eine (Hyper-)Kante verläuft, d.h. geht man davon aus, dass man auf gewichteten Hypergraphen arbeitet[1], so lässt sich eine Metrik unter Verallgemeinerung von Gleichung (4.1) ausdrücken durch

$$d(I_1, I_2, \ldots, I_n) = NCD(I_1, I_2, \ldots, I_n) := \frac{C_{I_1,I_2,\ldots,I_n} - min(C_{I_1}, C_{I_2}, \ldots, C_{I_n})}{\sum_{i=1\ldots n} C_{I_i} - min(C_{I_1}, C_{I_2}, \ldots, C_{I_n})}$$

[1]Ein Spezialfall dieses Modells, $d(I_1, I_2, \ldots, I_n) \in \{0, 1\}$ ist insbesondere in der Elektrotechnik zur Optimierung von elektronischen Schaltungsplänen wichtig, und wurde deswegen sehr vielschichtig untersucht [50; 51; 3; 48] — für die Anwendung auf Kompressionsmetriken ist allerdings der verallgemeinerte Fall $d(I_1, I_2, \ldots, I_n) \in \mathcal{R}$. wichtig.

Je nach Kompressionsalgorithmus ist die Berechnung einer Metrik oft eine sehr aufwändige Operation, aber natürlich enthält eine auf beliebigen Knotenmengen definierte (Hyper-)Metrik mehr Information als eine „normale“ Metrik eines gewichteten Graphen: Ein Hypergraph aus n Knoten hat $2^n - 1$ Hyperkantengewichte.

Zusätzlich zu einem Distanzwert „zwischen“ Bildern $I_1, I_2, \ldots, I_n$ kann man auch eine Distanz zwischen zwei Bildmengen $I_1, I_2, \ldots \subset \mathcal{I}$ und $J_1, J_2, \ldots \subset \mathcal{I}$ angeben:

$$d_2(\{I_1, I_2, \ldots\}, \{J_1, J_2, \ldots\}) = d(I_1, I_2, \ldots, J_1, J_2 \ldots) - d(I_1, I_2, \ldots) - d(J_1, J_2, \ldots)$$

Diese Operation entspricht gewissermaßen einem Kantengewicht in einem bipartiten Hypergraphen, und gibt an, wie gut oder schlecht die beiden Cluster $I_1, I_2, \ldots$ und $J_1, J_2, \ldots$ zusammenpassen. Dies ist wichtig für manche Clusteralgorithmen, wenn Entscheidungen zu treffen sind, ob zwei Cluster zu einem größeren Cluster zusammengefasst werden sollten.

Schlussendlich sei noch eine weitere — im Gegensatz zur NCD etwas „anschaulichere“ — Statistik angegeben, die der gewonnenen Bytes:

$$b^+(I_1, I_2, \ldots, I_n) = \sum_{I_j \in I_1, I_2, \ldots, I_n} C_{I_j} - C_{I_1, I_2, \ldots, I_n}$$

Die Funktion b^+ gibt an, wie viele Bytes gespart wurden dadurch, dass die Bilder $I_1, I_2, \ldots, I_n$ nicht als Einzelbilder sondern als Gruppe komprimiert wurden. Auch diese Funktion lässt sich zwischen Bildmengen anwenden:

$$b^+(\{I_1, I_2, \ldots\}, \{J_1, J_2, \ldots\}) = b^+(I_1, I_2, \ldots, J_1, J_2 \ldots) - b^+(I_1, I_2, \ldots) - b^+(J_1, J_2, \ldots)$$

Letzteres gibt an, wieviel Bytes man spart, wenn man die Cluster $I_1, I_2, \ldots$ und $J_1, J_2, \ldots$ zu einem größeren Cluster zusammenfasst, eine Metrik, die insbesondere bei iterativen Hypergraph-Clusteringverfahren (wie z.B. SAHN oder Greedy, siehe Kapitel 4.3.2) nützlich ist. Dieser Typ von Funktion (auf normalen Graphen und Hypergraphen) ist in Abbildung 4.1 gegenübergestellt.

Die b^+ Funktion kann man des Weiteren auch benutzen, um ein bezüglich Kompressionseffizienz optimales Clustering in geschlossener Form anzugeben — siehe dazu Kapitel 4.3.9.

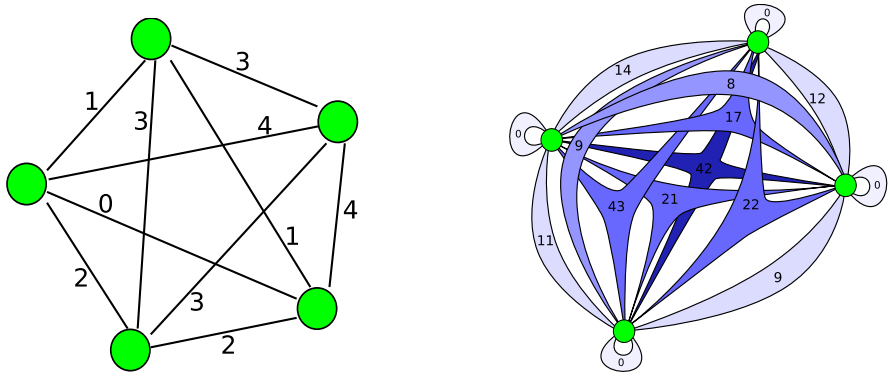


Abbildung 4.1: *Die b^+-Statistik auf Graphen (links) und Hypergraphen (rechts). In beiden Fällen ist als Kantengewicht angetragen, wie viele Bytes man sparen würde, wenn man die entsprechenden Knoten bzw. Bilder zusammen komprimiert.*

4.2 Klassische Distanz-Metriken zwischen Bildern

Teilweise ist die Ermittelung der Distanzrelation zwischen zwei oder mehr Bildern durch Durchführung einer (evtl. auch nur approximativen) Probekompression zeitlich zu aufwändig. Insbesondere wenn man bedenkt, wie viele Kompressionsläufe man benötigt, um eine komplette $1:1$ Distanzmetrik in einer Bilddatenbank mit n Bildern zu gewinnen: Man müsste dann den entsprechenden Algorithmus (unter Annahme der Symmetrie) $\frac{n\cdot(n-1)}{2} + n$ mal anwenden, und bei einer Distanzmetrik auf Hypergraphen sogar $2^n - 1$ mal.

Deswegen kann es effizienter sein, diese „Probekompressionsläufe“ durch ähnliche, weniger aufwändige, Verfahren zu ersetzen:
Generell ist das Suchen von Redundanzen zwischen Bildern ja analog dem Problem, Bilder zu finden, die ähnliche Features aufweisen. Letztendlich sind zwei Bilder ja genau dann ähnlich (oder redundant), wenn sie durch eine ähnliche „Klasse von Wahrscheinlichkeitsfunktionen“ erzeugt wurden.

D.h. es bietet sich an, auch ein paar, der im Bereich der Bildähnlichkeitsanalyse existierenden, weniger rechenzeitintensiven, Verfahren genauer zu beleuchten, um eventuell die komplexen Kompressionsmetriken durch einfachere Metriken ersetzen zu können.

Im folgenden beschäftigen wir uns deswegen mit allgemeinen, aus dem Bereich der Bildsuche stammenden Bildähnlichkeitsanalyse-Verfahren. Diese

sind auch als Distanzmetrik zwischen zwei Bildern definiert, und sagen aus, inwieweit zwei Bilder sich visuell ähnlich sind, sprich ob sie von menschlichen Beobachtern als ähnlich wahrgenommen werden. Diese Thematik findet sich in der Literatur klassischerweise als Bildsuche („image retrieval“)— ein Überblick ist z.B. in [66] zu finden.

4.2.1 Featureextraktion

Eine gängige Methode, Bilder auf Ähnlichkeiten hin zu untersuchen ist die Extraktion von bestimmten Features (z.B. Helligkeits- und Farbhistogramme, Kantendichte und -richtungen, Kovarianzen zwischen Auflösungsstufen etc.), um diese bei den Bildern gegenüber zu stellen.

Diese Features können sowohl global für das Gesamtbild generiert werden, als auch für aus dem Bild extrahierte Bildblöcke. Letzteres ist insbesondere eine elegante Methode, um beispielsweise Bilder mit ähnlichem Layout (z.B.: blauer Himmel oben, grüne Wiese unten) zu suchen.

Diese Features müssen pro Bild nur einmal extrahiert werden — danach werden Sie dann in einer Datenstruktur gespeichert, mittels derer dann sehr schnell die Features von zwei Bildern miteinander verglichen werden können.

Hierbei können insbesondere sogar Datenstrukturen aus dem Bereich der Volltextsuche, wie „Inverted Files“ [107], benutzt werden, um diese Operationen weiter zu beschleunigen. D.h. manche Features kommen nur in bestimmten Bildern vor, und man kann über diese Datenstrukturen schnell bestimmen, in welchen Bildern welches Feature vorkommt (und dementsprechend schnell Schnittmengen zwischen Bildern auf Basis bestimmter Features berechnen). [69]

Im Folgenden wird eine kleine Anzahl von möglichen Features vorgestellt, die für den Aufbau einer solchen Ähnlichkeitsdatenbank verwendet werden kann.

Farbhistogramme

Jedes Bild in der Computergrafik besteht aus einer endlichen Anzahl von Pixeln. Beispielsweise wird ein RGB-Bild durch eine Funktion

$$f : N \times N -> [0,1] \times [0,1] \times [0,1]$$

beschrieben. Die Umkehrung dieser Funktion

$$f^{-1} : [0,1] \times [0,1] \times [0,1] \rightarrow \mathcal{P}(N \times N)$$

bildet jeden der (diskreten) RGB Wert wiederum auf die Pixel ab, in denen er vorkommt. Betrachtet man nun die Funktion g der Pixelzahl pro Farbe

$$g : [0,1] \times [0,1] \times [0,1] \rightarrow N$$

$$g(r,g,b) = |f^{-1}(r,g,b)|$$

so erhält man für jede Farbe die Häufigkeit innerhalb eines bestimmten Bildes.

Als Beispiel dafür, inwieweit verschiedene RGB Histogramme Bilder differenzieren, sind in Abbildung 4.2 und 4.3 zwei Bilder nebst entsprechendem RGB Farbwürfel abgebildet.

Statt RGB sind andere Farbräume möglich, sinnvoll sind hier insbesondere Farbmodelle, die Helligkeit und Farbton voneinander trennen, wie z.B. HLS, YUV oder L*A*B. [95]

Üblicherweise quantisiert man nun diese Funktion auf eine Handvoll Farbwerte. Z.B. das in [90] beschriebene System benutzt 18 Farbtöne, 3 Farbsaturierungen, 3 Farbhelligkeiten und 4 Grauwerte.
Zwei Beispiele für, durch Quantisierung dieser Funktion, erzeugte Histogramme sind in Abbildung 4.4 abgebildet.

Die Farbhistogramme von Bildern liefern normalerweise eine relativ gute Klassifizierung von den meisten Bildern, sofern genug Diversität in der Farbkomponente der Objekte vorhanden ist.

Ist das nicht der Fall, oder will man auch mit Schwarzweißbildern arbeiten können, so bietet es sich an, auch auf Pixelhelligkeiten basierende Schattierungs- und Texturinformationen in die Klassifikationsroutinen mit aufzunehmen.

Auch die Präsenz, und Richtung, von auffälligen Kanten kann wesentliche Informationen über die visuelle Charakteristik eines Bildes geben.

Kantendichte und -richtung

Eine einfache, und sehr effektive Methode, Kantendichte und -richtung zu ermitteln, ist die Anwendung eines komplexen (d.h. aus zwei Komponenten

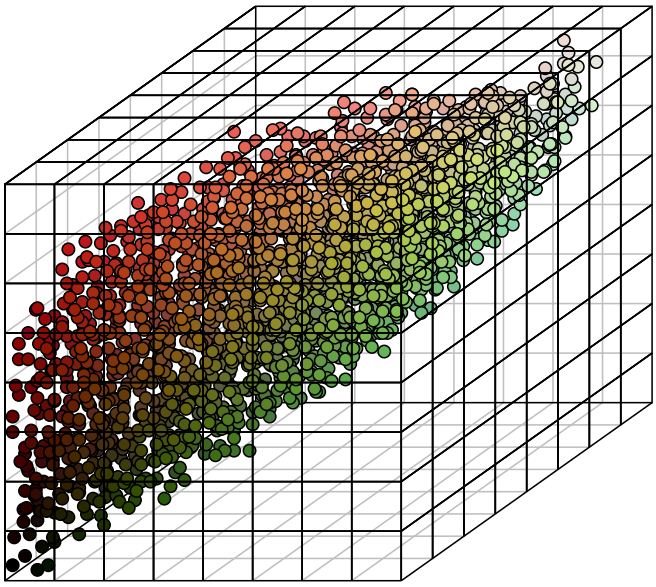


Abbildung 4.2: *Ein Bild, mit entsprechender RGB-Farbwürfelbelegung. Die Art der in ein Bildern verwendeter Farben kann Aufschluss über die Ähnlichkeit zwischen zwei Bildern geben.*

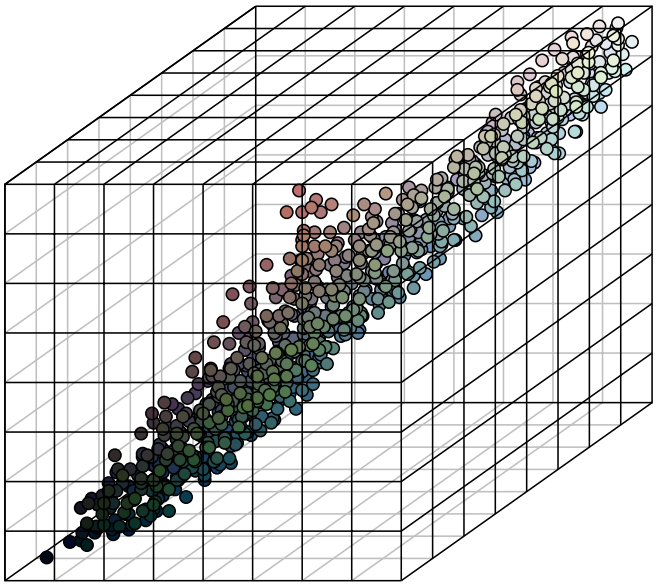


Abbildung 4.3: *Ein zweites, blaulastigeres Bild mit weniger Kontrast (stärkere Konzentration der RGB-Werte um die Diagonale) sowie weniger stark gestreuten Farbanteilen.*

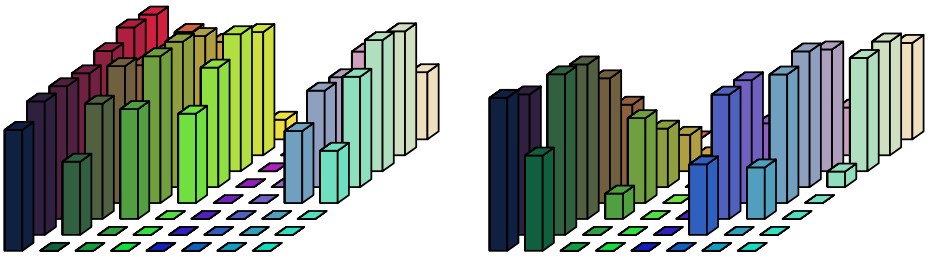


Abbildung 4.4: *Quantisierte Histogramme für die beiden Bilder aus Abbildung 4.2 (links) und 4.3 (rechts). Histogramme lassen sich aufgrund der niedrigen Anzahl von Werten gut vergleichen.*

bestehenden) Filters auf das Bild. Der Vorteil ist hierbei, dass bei einem komplexen Filter gleichzeitig auch noch die Kantenrichtung ermittelt wird.

In der Praxis haben sich Gabor Funktionen [52] oder „Difference of Gaussians“ (DOG) bewährt [28].

Die komplexe Form eines DOG-Filters hat folgende Form:

$$DOG(x,y) = \frac{x}{\pi\sigma^2} e^{(\frac{x^2+y^2}{\sigma^2})} + i\frac{y}{\pi\sigma^2} e^{(\frac{x^2+y^2}{\sigma^2})}$$

Hierbei ist σ die Größe des Filters, und damit auch die Dicke von Kanten, auf die dieser anspricht.

Aus dem Ergebnis eines mit diesem Filter gefilterten Bild $I_{DOG} = I \star DOG$ sind zwei Dinge abzulesen:

$|I_{DOG}(x,y)|_2$ gibt die Stärke der Kante an der Position (x,y) an.

$atan2(Re(I_{DOG}(x,y)), Im(I_{DOG}(x,y)))$ gibt die primäre Richtung der Kante (den Gradienten) an der Position (x,y) an.

Bild 4.5 zeigt ein Bild, aus denen mittels eines DOG Filters die wesentlichen Kantenrichtungen herausgefiltert wurden.

Filterhistogramme

Analog zu den Farbhistogrammen kann man auch Histogramme von Filtern, wie beispielsweise Kantenfilter, ableiten. Allerdings sind hier nicht nur

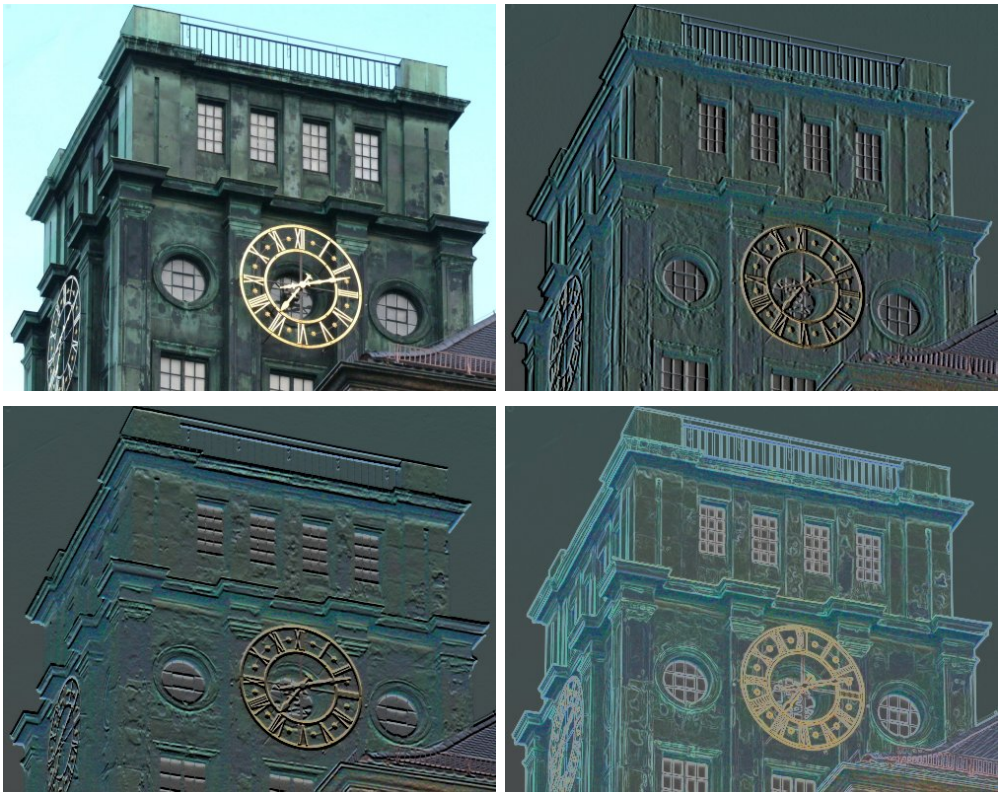


Abbildung 4.5: *Ein „Difference of Gaussians“ Kantenfilter. Der Filter wurde auf die Helligkeitskomponente der Bilder angewendet. Die Farbkomponente wurde in die gefilterten Bilder übertragen. Oben links: Das Originalbild. Oben rechts: Reelle Filterkomponente in X-Richtung. Unten links: Imaginäre Filterkomponente in Y-Richtung. Unten rechts: Radius bzw. Absolutbetrag* $|I_{DOG}(x, y)|_2$ *der komplexen Polardarstellung.*

Kantenartefakte ausschlaggebend, Filter sind auch ein mächtiges Mittel, um beispielsweise Bilder mit ähnlichen Texturen auszumachen, siehe Kapitel 3.5.

Zusätzlich zu den Kantenfiltern gibt es noch eine Reihe von anderen Filtern, wie Hochpass-/Tiefpass-/Bandpassfilter etc.
Durch eine Faltung von jedem Bild mit einer Reihe von Filtern (für ein Beispiel für eine größere Filterbank siehe Abbildung 2.11) ergibt sich an jedem Pixel der Bitmap ein Array von Filterantworten (ein Wert pro Filter).

Die durchschnittliche quadratische Antwort jedes Filters liefert dadurch einen Histogramm-Eintrag für das Bild. Je nachdem, welche Filter wie stark auf welches Bild ansprechen, enthält man z.B. Aussagen über Glattheit, Rauhheit, Kantenstärke und Kantendichte.

4.2.2 Vergleich von Features

Hat man in der in den vorigen Unterkapiteln beschriebenen Art und Weise eine Reihe von Histogramme für jedes Bild extrahiert, so lässt sich dadurch für jeweils zwei Bilder ein Ähnlichkeitswert ermitteln.

Um zwei Histogramme miteinander vergleichen zu können, gibt es eine Reihe von Verfahren, die u.A. in [74] näher erläutert werden.

Eine Möglichkeit ist z.B. die χ^2 Statistik, die folgendermaßen definiert ist [64]:

$$\chi^2(h_A, h_B) = \frac{1}{2} \sum_{i=1}^{N} \frac{[h_A(i) - h_B(i)]^2}{h_A(i) + h_B(i)}$$

Eine Alternative ist die Überlagerungsdichte [70] [91]:

$$\lambda(h_A, h_B) = \sum_{i=1}^{N} \frac{2 \cdot min(h_A(i), h_B(i))}{h_A(i) + h_B(i)}$$

Wenn die Histogramme sehr unterschiedlich sind, liefert $\chi^2(h_A, h_B)$ sehr große Werte, und $\lambda(h_A, h_B)$ sehr kleine.

Einen Vergleich von mehreren Histogrammen (z.B. sowohl Farb- als auch Filterhistogramme) erzielt man durch Addition der einzelnen χ^2 oder λ Sta-

tistiken. Der so ermittelte Wert gibt dann an, inwieweit zwei Bilder unter den verwendeten Histogrammen ähnlich sind.

4.2.3 Approximation von Kompressionsdistanz mittels Bildähnlichkeitsfeatures

Die vorgestellten Distanzmetriken wurden speziell dafür konzipiert, Aussagen zu treffen inwieweit Bilder (für den menschlichen Beobachter) „ähnlich aussehen“. Allerdings schlägt sich eine Ähnlichkeit von Bildern auf dem visuellen Level in der Regel durchaus in den „Bits und Bytes“ der entsprechenden Bilddatei nieder, weshalb Kompressionsalgorithmen auf (unter den eben vorgestellten Distanzmetriken) „ähnlichen“ Bildern meist bessere Kompressionsraten aufweisen als „unterschiedliche“ Bilder, siehe Abbildung 4.6.

Natürlich spielen in diese Korrelation zwischen visueller Ähnlichkeit und Kompressionsähnlichkeit auch Sekundärfaktoren mit ein, wie z.B., dass zwei **sehr** ähnliche Bilder von ein und demselben Objekt vermutlich zur selben Zeit vom selben Fotografen (und mit derselben Kamera) gemacht wurden, und sich deswegen sowohl Farbraum (d.h. Pixelstatistiken) als auch (bei JPEG Bildern) die genaue Implementierung des für die initial angewandte Einzelbildkompression verwendeten Algorithmus sowie eventuelle vorhandene Texturen ähneln.

In jedem Fall stellen Bildähnlichkeitsfeatures ein wichtiges Hilfsmittel dar, um Bilder zum Zwecke des Clusterings schnell vergleichen zu können, da es um Größenordnungen schneller ist, die entsprechenden Statistiken anzuwenden, als Probekompressionsläufe zur Ermittlung der „wahren“ Distanz durchzuführen.

Im nächsten Kapitel werden nun die Clustering-Algorithmen, die auf Basis einer Ähnlichkeitsfunktion verwendet werden können um Bilder in Gruppen zusammenzufassen, vorgestellt. Diese lassen sich natürlich sowohl auf die vollwertigen Kompressionsdistanzmetriken als auch auf Bildähnlichkeitsstatistiken anwenden.

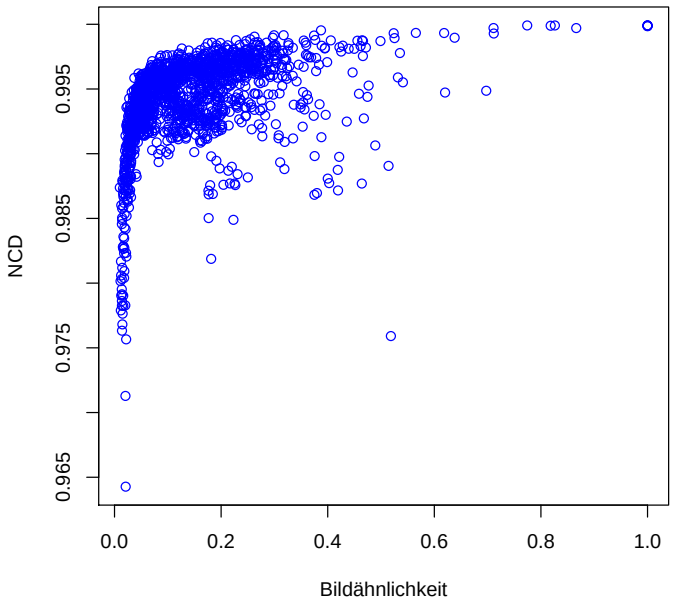


Abbildung 4.6: *Korrelation zwischen Kompressionsähnlichkeit (hier: fraktale Kompressionsdistanz) sowie über Features ermittelte Bildähnlichkeit. Die Abbildung zeigt beide Werte gegeneinander, für 1600 zufällig generierte Bildpaare der Größe* 512×512 *Je kleiner die über Bildähnlichkeit ermittelte Distanz zwischen den Bildern ist (d.h. je ähnlicher die beiden Bild sind), desto mehr Kompressionseffizienz (gemessen mittels der NCD-Metrik) erzielt man in den meisten Fällen, wenn man beide zusammen komprimiert.*

4.3 Clustering Algorithmen

Zu jedem Clusteringverfahren unter Verwendung eines bestimmten Kompressionsalgorithmus kann für eine Menge von Bildern eine klare Statistik ermittelt werden, wie gut die gefundenen Cluster tatsächlich sind- die Güte eines Clustering wird hier einfach durch die resultierende Dateigröße bestimmt.

Es gibt aber noch andere Kriterien für die Qualität eines Algorithmus: Insbesondere die typische Laufzeit (diese hängt — insbesondere bei Kompressionsmetriken — hauptsächlich von der Anzahl von Bildvergleichen ab, die durchgeführt werden müssen) sowie die Ausgewogenheit der resultierenden Bildmengen, sprich ob der Algorithmus dazu neigt, teilweise sehr große oder sehr kleine Cluster anzulegen.

Von den hier angesprochenen Algorithmen sind nCut und MST auf gewichteten Graphen definiert. SAHN und Greedy sowie genetische Verfahren arbeiten auf Hypergraphen, und k-Means sowie MeanShift auf durch „Multidimensional Scaling“ aus den gewichteten Graphen ermittelten Vektorfeldern.

Im folgenden folgt eine genauere Beschreibung der einzelnen Algorithmen. Es sei angemerkt, dass die hier angesprochenen Algorithmen nur eine Auswahl dessen repräsentieren, was in der Informatik an Clustering-Verfahren zu finden ist. Es wurde bei der Auswahl darauf geachtet, möglichst Algorithmen zu untersuchen, die sich für das hier verwendete Einsatzgebiet (Clustering auf Basis von Kompressionsmetriken) eignen.

4.3.1 Kleinster Spannbaum (MST)

Der MST-Algorithmus operiert auf gewichteten Graphen, und ist aus graphentheoretischer Sicht optimal: Er generiert Cluster derart, dass für einen beliebigen Cluster $P \in \mathcal{I}$ gilt [108]:

$$\forall A \subset P, B \subset P, A \cap B = \emptyset, A \cup B = P : \forall x \in A : \left(\begin{array}{c} \text{argmin} \\ y \in \mathcal{I} \backslash A \end{array} d(x, y) \right) \in B$$

Mit anderen Worten, jeder Cluster ist dadurch definiert, dass die kürzeste Verbindung von jedem Knoten (oder jeder Menge von Knoten) wieder zu einem anderen Knoten im selben Cluster führt.

Die eigentliche Berechnung der Cluster erfolgt durch Ermitteln des optimalen Spannbaumes (Minimal Spanning Tree, MST), und darauffolgendes Entfernen einer Reihe von Kanten dieses Baumes, solange bis die gewünschte Clusteranzahl erreicht ist, siehe Abbildung 4.7.

In seiner ursprünglichen Formulierung [108], bei der die kleinsten Kanten vom Baum abgeschnitten werden, erzeugt dieser Algorithmus nur einen Cluster der aus mehr als einer Node besteht, sowie eine Reihe von einzelnen Nodes[2], siehe Abbildung 4.8, was natürlich für praktische Anwendungen eine Konfiguration mit sehr begrenztem Nutzen ist.

Interessanter sind Modifikationen von MST, die nicht die schwächsten Kanten des Spannbaumes durchtrennen, sondern stattdessen die Kanten, die den Baum möglichst optimal in gleichgroße Teile aufteilen [5], um ausgewogenere Cluster zu erhalten. (Allerdings auf Kosten der graphentheoretischen Optimalität)

Eine der in dieser Arbeit verwendeten Testimplementierungen durchtrennt den Spannbaum rekursiv immer anhand der Kante, bei denen das Verhältnis aus den beiden entstehenden Baumteilen am ausgewogensten ist — der entsprechende Algorithmus wird im nachfolgenden mit „MST2“ bezeichnet.

4.3.2 SAHN

Die Familie der SAHN (Sequential Agglomerative Hierarchical Nonoverlapping) Algorithmen basiert auf dem Prinzip, dass bei jedem Schritt zwei Cluster zusammengefasst werden und einen neuen Cluster bilden, solange bis die gewünschte Clusteranzahl erreicht ist, siehe Abbildung 4.9. Gestartet wird der Algorithmus, indem für jede Node ein eigener Cluster angelegt wird.

Der SAHN Algorithmus erzeugt in der Regel ausgewogenere Cluster als der MST Algorithmus, benötigt dafür aber eine auf bipartiten Hypergraphen definierte Distanzfunktion (Siehe Kapitel 4.1). Er evaluiert allerdings nur einen Teil der darin definierten Gewichte.

[2]Man kann anhand des Spannbaum-Algorithmus von Prim leicht nachvollziehen, dass die kleinste Kante im Baum auch die Kante ist, die zuletzt zum Baum hinzugefügt wurde. Deswegen ist sie auch in jedem Fall eine Kante, für die für einen ihrer Knoten k gilt: $deg(k) = 1$. Des Weiteren hat der Graph, der entsteht, wenn der Knoten k vom ursprünglichen Graph entfernt wird, denselben Spannbaum (ohne k), und auch hier verbindet die schwächste Kante einen einzelnen Knoten mit dem Restgraph. Diese Argumentation kann man rekursiv für die restlichen Kanten/Cluster anwenden.

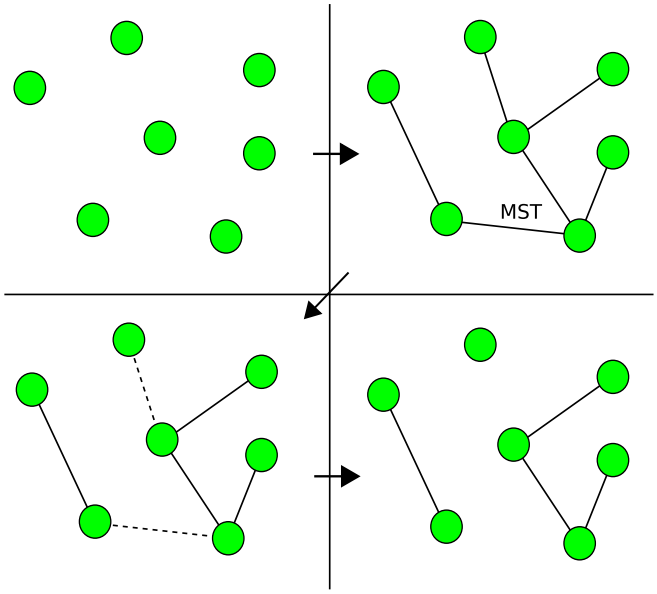


Abbildung 4.7: *Der MST Algorithmus: Es wird zuerst ein minimaler Spannbaum errechnet, und dieser dann durch Entfernen von Kanten in eine Reihe von kleineren Bäumen/Cluster zerlegt.*

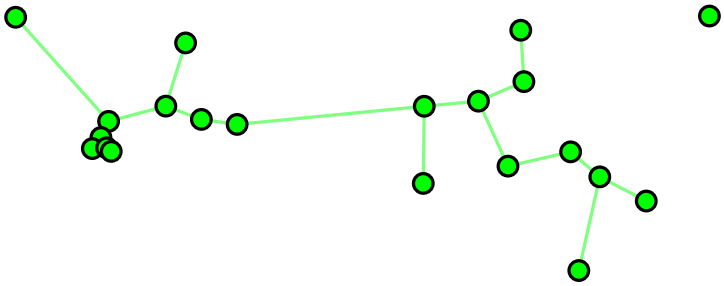


Abbildung 4.8: *Entartetes Ergebnis des „klassischen“ MST Algorithmus: Vom Spannbaum wurde die längste Kante (die zum Punkt rechts oben) entfernt, woraus sich zwei Cluster ergeben: Ein kleiner aus einem Punkt bestehender, sowie ein großer, der aus sämtlichen restlichen Punkten besteht — wenngleich man vermutlich eher erwarten würde, dass ein Clustering-Algorithmus den linken Punkthaufen vom rechten getrennt hätte. Klassisches MST Clustering ist aber aufgrund seiner Natur gezwungen, die einzelne Node rechts abzuspalten.*

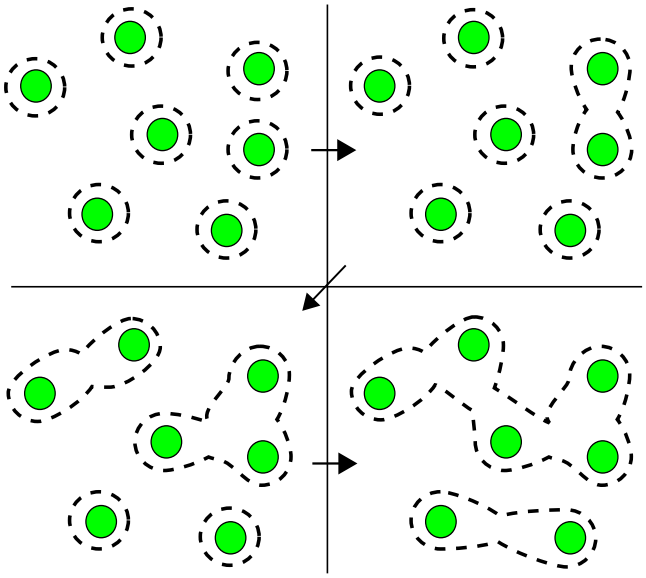


Abbildung 4.9: *Der SAHN Algorithmus*

Da in jedem Rechenschritt durch Zusammenfassung von zwei Clustern die Clusteranzahl um eins reduziert wird, benötigt man $n-k$ Rechenschritte. Des Weiteren muss in jedem Rechenschritt noch die Distanzfunktionen zwischen dem im letzten Schritt neu hinzugekommenen Cluster und den restlichen Clustern berechnen werden, was in Schritt i immer $n-i$ Vergleiche kostet. D.h. insgesamt finden

$$\sum_{i=1}^{n-k} n - i = \frac{1}{2}(n-k) \cdot (n+k-1)$$

Berechnungen von Hypergraph-Gewichten statt.

Man kann, ähnlich wie bei MST, den SAHN Algorithmus verändern, um ausgewogenere Cluster zu erhalten, z.B. indem man bei der Kombination von Clustern nur die betrachtet, die momentan minimale Größe haben. Dies ist auch eine Verfeinerung, die im Rahmen dieser Evaluation vorgenommen wurde, unter dem Namen „SAHNb“.

4.3.3 Greedy

Der Greedy Algorithmus [54] arbeitet ähnlich dem gerade vorgestellten SAHN Algorithmus auf bipartiten Hypergraph-Gewichten, muss aber im Vergleich zu SAHN nur durchschnittlich $n \cdot k$ Gewichte evaluieren.

Der Algorithmus heißt Greedy, weil er jedes Bild nur genau einmal bearbeiten muss: In jedem Schritt wird ein neues Bild den bereits vorhandenen k Clustern hinzugefügt, wobei nur die Distanzen zwischen den momentanen Clustern sowie dem Bild k benötigt werden.
Insbesondere muss der Algorithmus deswegen (im Gegensatz zu den anderen bisher hier vorgestellten Algorithmen) auch nicht im Vorhinein wissen, wie viele Bilder es gibt, sondern Bilder können auch im Nachhinein zur Datenbank hinzugefügt werden.

Am Anfang der Bearbeitung werden k Cluster aus einer Menge von zufälligen Nodes erzeugt (die Anfangs-Clustergröße m ist hierbei ein einstellbarer Parameter), und danach werden alle Nodes der Reihe nach jeweils dem Cluster zugewiesen, zu dem sie die kleinste (Hyperkanten-)Distanz haben, siehe Abbildung 4.10.

Das heißt in jedem Schritt werden k Distanzrelationen benötigt, und der Algorithmus terminiert in $n - km$ Schritten, was $k \cdot (n - km) \simeq k \cdot n$ Gewichtsevaluationen entspricht.

4.3.4 nCut

nCut („normalized cuts“) ist das „top down“ Gegenstück zum „bottom up“ SAHN Algorithmus. SAHN basiert ja darauf, den Graph durch Zusammenfassung von Nodes aufzubauen — nCut geht vom Gesamtgraph aus, und zerspaltet diesen solange in Einzelteile, bis die gewünschte Anzahl von Nodes erreicht ist, d.h. jeder (Sub-)Graph der Relation wird in jedem Schritt in zwei Teile zerlegt. (Siehe Bild 4.11)

Die Art und Weise, in der die Graphen in zwei Teile zerlegt werden, basiert auf der Optimierung (Maximierung) des „nCut-Kriteriums“ [86]:

$$nCut(A, B) = \frac{cut(A, B)}{assoc(A, B)} + \frac{cut(B, A)}{assoc(B, A)}$$

wobei $assoc(A, B) = \sum_{i \in A, k \in A \cup B} d_{ik}$ die Summe aller Verbindungsgewichte

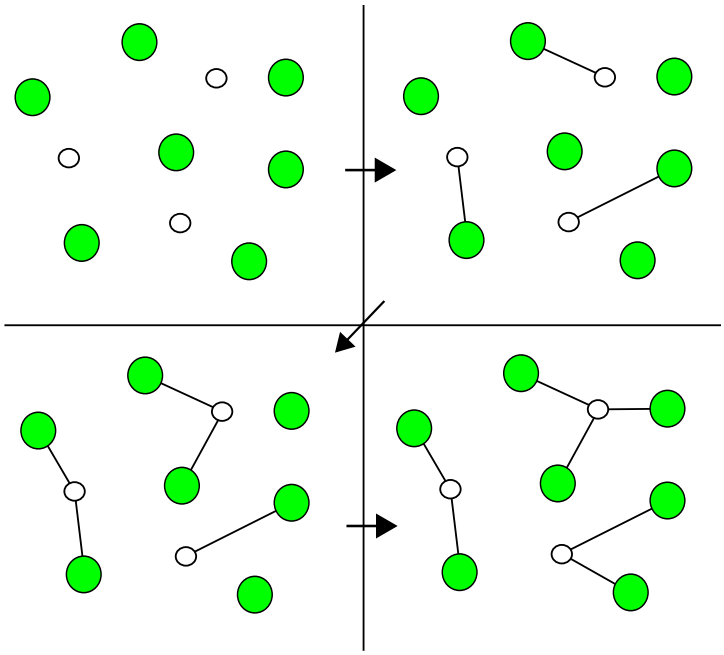


Abbildung 4.10: *Der Greedy Algorithmus— hier werden zuerst $k = 3$ (leere) Cluster initialisiert (in der Abbildung als kleine weiß gefüllte Kreise gekennzeichnet), und danach werden nach und nach die Nodes den jeweils besten Clustern zugewiesen*

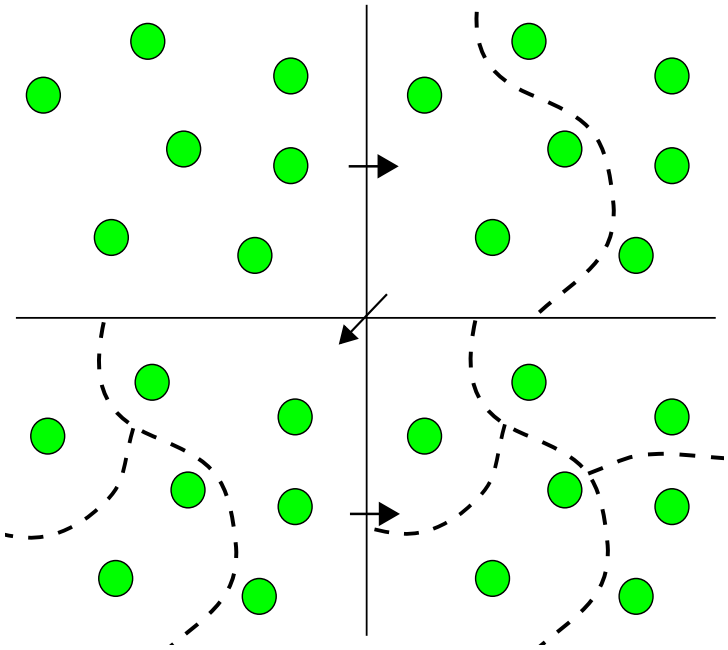


Abbildung 4.11: *Der nCut Algorithmus*

Algorithm 2 SUBDIVIDE_GRAPH(G, NUM_CLUSTERS)

```
current_clusters = {G}
while num(current_clusters) < num_clusters do
    C = find_biggest_cluster(current_clusters)
    C1,C2 = nCut(C)
    Add C1,C2 to current_clusters
    Remove C from current_clusters
end while
return current_clusters
```

von A in den Rest des Graphen $(A \cup B)$ ist, und $cut(A, B) = \sum_{i \in A, j \in B} d_{ij}$ die Summe aller Verbindungsgewichte der Kanten zwischen A und B.

Minimiert wird der Wert von $nCut$ in Abhängigkeit von den Knotenmengen A und B durch Lösen des Eigenwertproblems

$$(D - W)x = \lambda Dx$$

mit D Diagonalmatrix, $d_{ii} = \sum_{j=0}^{N} w_{ij}$ und W die Kantengewichtungsmatrix. Der zum zweitkleinsten Eigenwert gehörende Eigenvektor enthält draufhin für den ersten Cluster positive und für den zweiten Cluster negative Werte [86].

Dieser Algorithmus benötigt $O(n^3)$ Rechenschritte zum Berechnen der Eigenvektoren, und damit $O(kn^3)$ Schritte, um einen Graphen aus n Knoten in Cluster zu zerlegen. Allerdings benötigt er nur Gewichte auf normalen Graphen (nicht auf Hypergraphen), weshalb der etwas erhöhte Aufwand zur Bestimmung der Cluster (auf Basis der Gewichte) erträglich ist. In praktischen Anwendungen ist der Aufwand zur Berechnung der Eigenvektoren meist vernachlässigbar, und es spielt nur der Zeitaufwand zur Kantengewichtsberechnung eine Rolle.

Es gibt auch eine Variante von nCut, die auf gewichteten Hypergraphen arbeitet (d.h. Bildmengen vergleicht) [110], allerdings benötigt diese zur Berechnung des Indikatorvektors **alle** $2^n - 1$ Hypergraph-Gewichte, und ist deswegen für praktische Anwendungen ungeeignet.

4.3.5 Multidimensional Scaling

Nicht immer ist der Ansatz, die Ähnlichkeit zwischen zwei Nodes auf Basis eines gewichteten (Hyper-)Graphen zu modellieren, zuträglich für die Performance eines Algorithmus. Deswegen gibt es Algorithmen, die in den nachfolgenden Kapiteln vorgestellt werden, welche stattdessen auf Vektoren im m-dimensionalen Raum arbeiten, und i.d.R. im Vergleich zu Graphalgorithmen sehr schnell sind, da sie einige geometrische Optimierungen vornehmen können, die bei Graphen nicht möglich wären.

Das heißt diese Algorithmen gehen davon aus, dass jede Node x_k ein Punkt $x_k \in \mathcal{R}^n$ in einem n-dimensionalen Raum ist, und deswegen u.A. die Möglichkeit besteht, Mittelpunkte zwischen Nodes zu berechnen, und auch für diese „neuen“ Nodes mittels einer gängigen n-dimensionalen Metrik (z.B. Mahalanobis-Distanz [21]) Distanzfunktionen zu anderen Nodes anzugeben.

Um einen derartigen Algorithmus auf Daten anwenden zu können, die nicht als Vektoren sondern als Graph vorliegen, ist es notwendig, diese Daten zuerst zu konvertieren. Diese Konvertierung ist in beide Richtungen möglich:
Als Vektorfelder definierte Daten lassen sich in Metriken konvertieren (indem man eine geeignete geometrische Distanzfunktion auf dem Vektorfeld definiert), gleichzeitig ist es aber auch möglich, mittels „Multidimensional Scaling“ aus Metriken [19] [13] oder gewichteten Hypergraphen [30] Vektorfelder zu gewinnen.

Die Technik des „Multidimensional Scaling“ (MDS) stammt ursprünglich

aus der statistischen Analyse. Es werden aus bekannten Distanzen zwischen Punkten, also einer Distanzmatrix D (mit $d_{i,j}$ die Distanz zwischen Punkt i und Punkt j) eine Menge von m-dimensionalen Vektoren $x_i, i = 1 \dots n$ (das heißt eine Matrix $X \in \mathcal{R}^{n \times m}$) berechnet, bei denen die „neue“ eulersche Distanz $\hat{D}$ mit $\hat{d}_{i,j} = |x_i - x_j|_2, i, j \in 1 \dots n$ möglichst gut die vorgegebene Distanz $d_{i,j}$ wiedergibt. Der Parameter m (d.h. die Dimension der resultierenden Vektoren) ist hierbei meist vom Anwender anzugeben. Generell gilt: Je höher die Dimension m, desto genauer wird die ursprüngliche Distanzfunktion approximiert.

In der bekanntesten Form des MDS, dem „Classical Scaling“ [13] ermittelt man die Vektoren, indem man zuerst die quadratischen Distanzen $\hat{D}^2$ in Matrixform schreibt:

$$\hat{D}^2 = \overrightarrow{tr}(XX^T)i^T + i\overrightarrow{tr}(XX^T) - 2XX^T$$

(mit $\overrightarrow{tr}(X)$ der Vektor der Diagonalelemente von X, und $i := \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$)

Da $i^T J = J i^T = 0$, ergibt sich durch Multiplikation mit der Matrix $J = \frac{1}{\sqrt{2}}(E - \frac{1}{n} i i^T)$ von beiden Seiten die Vereinfachung [19]:

$$\begin{aligned} J\hat{D}^2 J = & \quad J(\overrightarrow{tr}(XX^T)i^T + i\overrightarrow{tr}(XX^T) - 2XX^T)J \\ = & \quad J\overrightarrow{tr}(XX^T)i^T J + Ji\overrightarrow{tr}(XX^T)J - J2XX^T J \\ = & \quad JXX^T J \end{aligned}$$

Geht man bei den zu berechnen Koordinaten X davon aus, dass $JXX^T J = XX^T$ (d.h. man verlangt von der Lösung, dass X spaltenzentriert ist [13]), so lässt sich X aus JD^2J durch Eigenwertzerlegung lösen:

$$X = Q\Lambda^{\frac{1}{2}}$$

mit

$$JD^2J = Q\Lambda Q^T$$

und Λ Diagonalmatrix.

Durch diese Berechnung wird der Fehler zwischen den vorgegebenen quadratischen Distanzen und den Distanzen zwischen den zu berechnenden Punkten minimiert [34]. Man kann zeigen, dass die für die durch die MDS berechneten Distanzen bestimmte Fehlerfunktion

$$\epsilon := |D^2 - \hat{D}^2|_2 = \sqrt{\sum_{i,j \in 1 \ldots n} (d_{i,j}^2 - (|x_i - x_j|_2)^2)^2}$$

den kleinsten Wert annimmt.

Die durch MDS ermittelten Koordinaten kann man sich auch in der Datenanalyse nutzbar machen, indem man z.B. die durch eine Distanzmatrix spezifizierte Relation zweidimensional darstellt. Im folgenden werden wir sie allerdings benutzen, um spezielle, auf Vektorfeldern arbeitenden, Clusteringalgorithmen benutzen zu können.

4.3.6 k-Means

Der k-Means Algorithmus [75] ist einer der Algorithmen, die im Gegensatz zu den bisher vorgestellten Verfahren nicht auf Graphen bzw. Hypergraphen arbeiten, sondern auf Vektorfeldern.

D.h. dieser Algorithmus geht davon aus, dass jede Node $I_k \in \mathcal{I}$ ein Punkt $I_k \in \mathcal{R}^n$ in einem n-dimensionalen Raum ist, der entweder durch MDS ermittelt wurde, oder bereits ursprünglich so vorlag. In so einem Datenmodell ist es möglich, Mittelpunkte zwischen Nodes zu berechnen, und auch für diese „neuen“ Nodes mittels einer gängigen n-dimensionalen Metrik (z.B. euklidische Distanz, Manhattan-Norm, Mahalanobis-Distanz [21] etc.) Distanzfunktionen zu anderen Nodes anzugeben.

Wie die meisten anderen Algorithmen, die intern auf diesem Datenmodell arbeiten, ist k-Means sehr schnell, mit einer Laufzeit von $O(kn)$.

Das Prinzip von k-Means auf Vektoren der Dimension m ist das folgende: Zuerst werden zufällig k Clusterzentren bestimmt (z.B. durch zufällige Auswahl von k Punkten, oder durch Generieren von k m-dimensionalen Zufallsvektoren), danach folgt abwechselnd auf Basis einer im m-dimensionalen bestimmten Distanzmetrik eine Zuordnung der verschiedenen Nodes zu den Clusterzentren, sowie die Berechnung der Clusterzentren, durch Mitteln (z.B. arithmetisches Mittel) der zu ihnen gehörenden Punkte.

Durch diese Weise stoßen sich die Clusterzentren gegenseitig ab, und konvergieren gegen Punkte allgemeiner Konzentration, siehe Abbildung 4.12.

4.3.7 MeanShift

Auch der MeanShift Algorithmus arbeitet auf Vektorfeldern, und benötigt deswegen, wenn die Daten (wie hier) als Distanzrelation vorliegen, den MST-Algorithmus aus dem vorgehenden Kapitel.

MeanShift arbeitet prinzipiell ähnlich wie k-Means — eine Reihe von Clusterzentren wird vorinitialisiert, zufällig im Raum platziert, und dann gegen die jeweiligen Konzentrationsbereiche konvertiert.

Der maßgebliche Unterschied zu k-Means ist es, dass sich Clusterzentren nicht gegenseitig abstoßen, und es deswegen sein kann, dass mehrere Zentren in dasselbe lokale Minimum konvergieren.

Die Bewegung jedes Clusterzentrums erfolgt aufgrund seiner lokalen Nachbarschaft im Featureraum — es bewegt sich in die Richtung, die die stärkste Konzentration an Nodes in der momentanen Umgebung aufweist, siehe Abbildung 4.13.

MeanShift wird in der Literatur teilweise auch als „Neuronales Gas“ [67] bezeichnet, da die Clusterzentren von Verhalten her diffundierenden Gaspartikeln ähneln.

Mathematisch ist MeanShift motiviert durch ein Gradientenverfahren, welches versucht, den „kernel density parameter“ zu optimieren [20]:

$$f(x) = \sum_{i=1}^{n} k(|x - p_i|^2)$$

(mit $k(x)$ eine m-dimensionale Distanzfunktion, wie z.B. die euklidische Distanz, und $p_i, i = 1 \ldots n$ die Punkte im Featureraum)

Bildet man den Gradienten der obigen Gleichung, so ist die Richtung des stärksten Anstiegs gegeben durch:

$$\nabla f(x) = \sum_{i=1}^{n} \nabla k(|x - p_i|^2) = 2 \sum_{i=1}^{n} (x - p_i) \delta k(|x - p_i|^2)$$

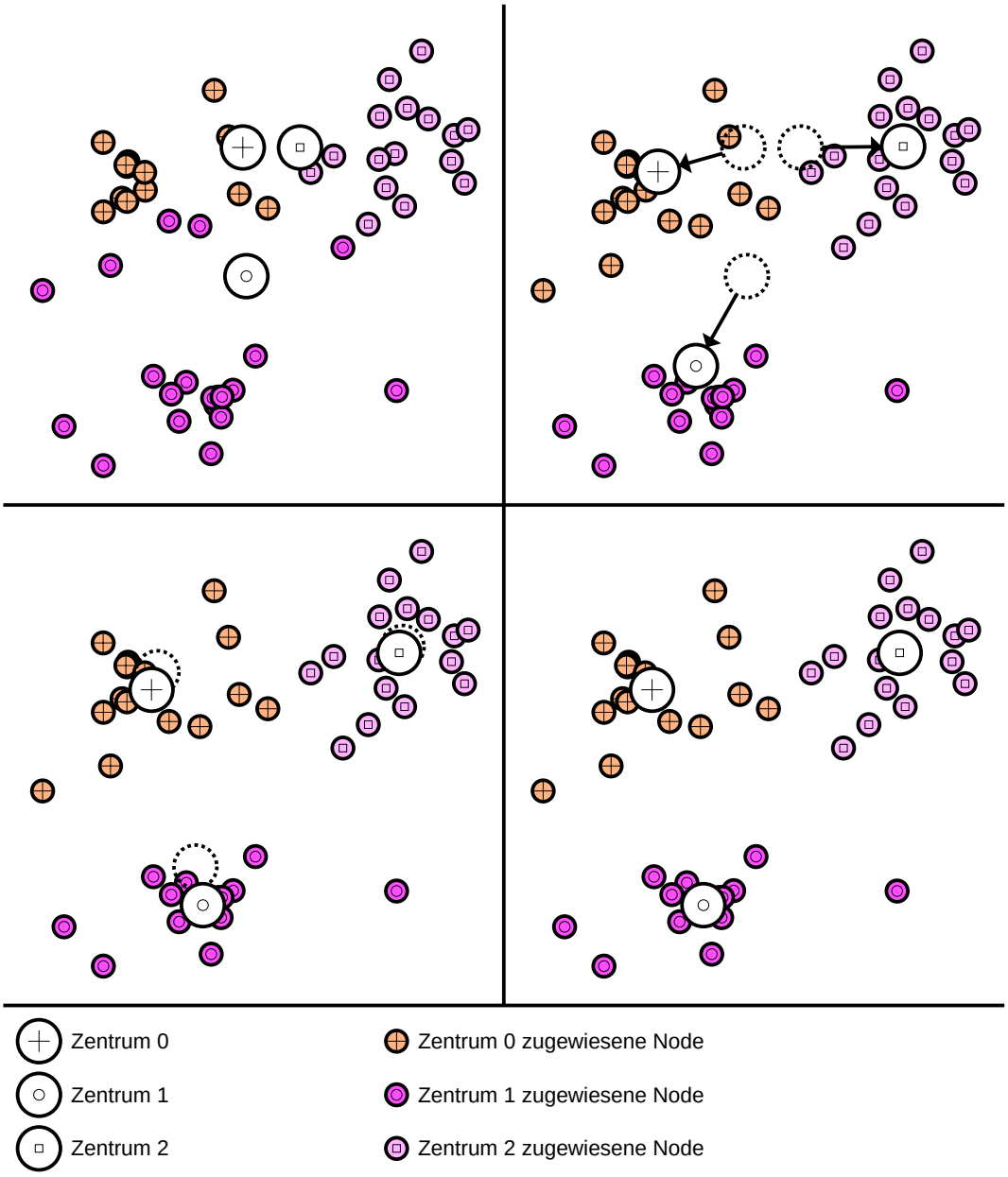


Abbildung 4.12: *Der k-Means Algorithmus benötigt beim dargestellten (zweidimensionalen) Datenmaterial vier Schritte, um Cluster zu bestimmen. Abwechselnd werden die (hier etwas größer dargestellten) Clusterzentren zum Mittelpunkt „ihrer“ Punkte bewegt, und danach alle Punkte neu den Clusterzentren zugewiesen, mittels (hier) euklidischer Distanz.*

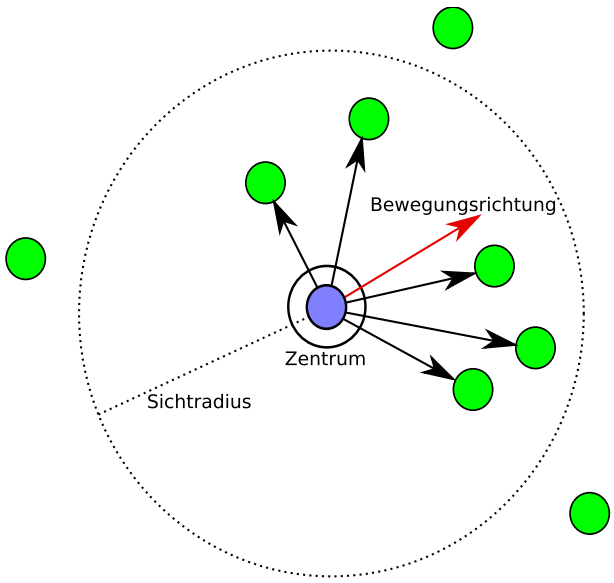


Abbildung 4.13: *Der MeanShift Algorithmus*

Definiert man nun die Distanzfunktion derart, dass sie in einem bestimmten Radius r konstant ist, und außerhalb 0, d.h.

$$k(x) = \begin{cases} x \text{ für } x < r \\ r \text{ sonst} \end{cases}$$

so ist

$$\delta k(x) = \begin{cases} -1 \text{ für } x < r \\ 0 \text{ sonst} \end{cases}$$

und der Gradient reduziert sich auf [20]

$$\nabla f(x) = \sum_{i=1\ldots n, |x-p_i|^2<r} p_i - x$$

Diese Funktion ist auch bei großen Punktdichten schnell zu berechnen, weshalb MeanShift teilweise schneller arbeitet als k-Means. Insbesondere muss im Gegensatz zu k-Means nicht nach jedem Rechenschritt die neue Zugehörigkeit der Punkte bestimmt werden. Punkte werden erst am Ende des MeanShift-Algorithmus den Clusterzentren zugeordnet.
Allerdings muss i.d.R. eine größere Clusteranzahl berechnet werden, als im Endergebnis benötigt wird, da öfter Clusterzentren gegen die gleichen Punkte konvergieren.

4.3.8 Genetische Algorithmen

Ein Verfahren, das nicht explizit für das Clustering von Daten konstruiert wurde, sich aber dafür verwenden lässt, ist die Familie der genetischen Algorithmen.

Genetische Algorithmen eignen sich für eine sehr große Anzahl von Optimierungsproblemen, insbesondere für Probleme mit vielen unterschiedlichen Nebenbedingungen, für die nicht leicht eine explizite algorithmische Lösung gefunden werden kann.

Das Prinzip eines genetischen Algorithmus beruht auf einem **Gen-Pool**, in dem eine Reihe von **Individuen** miteinander konkurrieren, und in dem durch **Mutation** und **Kreuzung** immer neue, bessere, Generationen entstehen.

Hierbei werden einzelne Individuen, die die bisher besten Lösungen zu einem gegebenen Optimierungsproblem darstellen, durch Abwandlung (Mutation) oder durch Vermischung mit anderen Individuen (Kreuzung) gezielt verändert und im Gen-Pool weitergeführt. Das bisher beste Individuum nach einer Reihe von Generationen ist dann die genetisch optimierte Lösung des Problems [33].

Um genetische Algorithmen nun auch zum Clustering von Daten einzusetzen, sind eine Reihe von Ansätzen möglich, je nachdem, welche Ähnlichkeitsrelation zwischen den Nodes angegeben ist. Die Gene bzw. Chromosomen im genetischen Pool sind dann entweder z.B. Clusterzentren [68], Prototypen [57], oder Indikatorfunktionen [56].

Im hier eingesetzten Fall werden Gene als Cluster-Zugehörigkeiten eines bestimmten Bildes und dementsprechend Individuen als Aufteilung einer Bilddatenbank in Cluster behandelt.
Die Qualität („fitness“) eines Individuums definiert man dementsprechend durch die Summe aller Dateigrößen der komprimierten Cluster, sprich durch den Plattenplatz, der durch ein gegebenes Clustering gespart wird.

Die Kreuzung von zwei Individuen A und B findet für jeweils zwei Cluster einzeln statt, indem durch einen Maximum-Matching-Algorithmus zuerst eine Zuordnung der Cluster in A zu den Clustern in B vorgenommen wird, und danach die Mengen der „Doppelcluster“ geeignet miteinander vermischt.

Durch das maximale bipartite Matching wird (global) die Clusterzuordnung ermittelt, bei der die gegenüber zu stellenden Cluster möglichst viele gemein-

same Elemente haben.

Das heißt, das Maximum Matching optimiert die Funktion

$$f(M) = \sum_{j=1}^{n_A} |A_j \cap B_{M(j)}|$$

mit M die Funktion, die einen Cluster in A auf den entsprechenden Cluster auf B abbildet, n_A die Anzahl der Cluster in A, und A_j, B_j der j-te Cluster des Individuums A bzw. B.

Ist eine solche Zuordnung gefunden (was in polynomieller Zeit bewerkstelligt werden kann) [81] so werden die Individuen nach folgendem Verfahren neu gemischt:

Jeder Cluster in A wird mit seinem Gegenstück in B zu einem Doppelcluster zusammengefasst. Jede Node gehört ab diesem Zeitpunkt zu einem oder zwei Doppelclustern. Jede Node, die zu zwei Doppelclustern gehört, wird aus einem ihrer beiden Cluster entfernt. D.h. jede Node, die in zwei (miteinander durch das Matching verbundenen) Clustern vorkommt, wird in den neuen Cluster überführt, die anderen Nodes werden zufällig verteilt.
Abbildung 4.14 zeigt die prinzipielle Vorgehensweise der Kreuzung.

Das Ergebnis ist ein neues Clustering, das möglichst viele Eigenschaften der beiden ursprünglichen Clusterzusammenstellungen behält.

Die zweite Möglichkeit, neue Individuen zu schaffen, die Mutation, ist einfacher — sie passiert dadurch, dass bei einem Individuum zufällig Nodes zwischen Clustern vertauscht werden.

Die Individuen werden durch den Algorithmus nun eine definierte Zeit lang mutiert bzw. gekreuzt, danach terminiert der Algorithmus, und gibt das bisher beste Individuum, d.h. die Aufteilung der Bilder die am wenigsten Plattenplatz beansprucht, zurück.

Vorteilhaft am genetischen Algorithmus ist nun insbesondere, dass sich die Qualität des Ergebnisses auch dadurch beeinflussen lässt, wie lange man bereit ist, den Algorithmus rechnen zu lassen. Dies ist ein Vorteil zu den anderen bisher hier besprochenen Verfahren, die es nicht ohne Weiteres ermöglichen, ein bereits errechnetes Ergebnis noch weiter zu „verfeinern“.

Eine weitere Besonderheit ist es, dass man Ergebnisse von anderen Clustering-Verfahren (insbesondere schnell zu berechnende, wie MST, k-Means oder nCut) als Eingabewerte für den genetischen Algorithmus benutzen kann, um

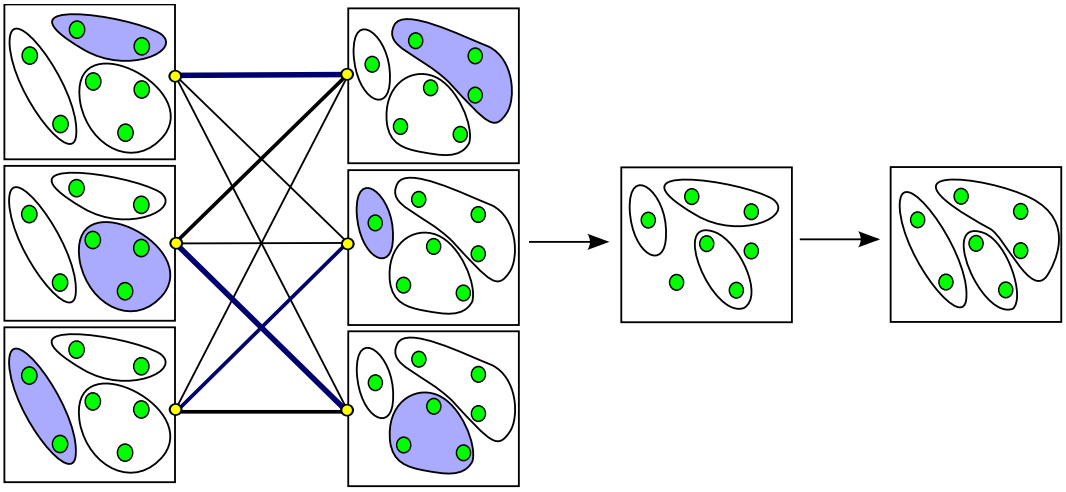


Abbildung 4.14: *Neukombination von Gruppen im Gen-Pool mittels Maximalem Matching- es werden die Teilcluster bestimmt, die die maximale Anzahl von gemeinsamen Nodes haben, und aus der Schnittmenge neue Cluster gebildet.*

die Qualität des initialen Pools und damit die Berechnungsgeschwindigkeit zu verbessern.

In der Auswertung am Ende des Kapitels werden zwei verschiedene Implementierungen bewertet: die eine Variante („Genetic“) benutzt als Bewertungsfunktion die Anzahl der gesparten Bytes, die zweite Variante („Genetic2“) die Anzahl der gesparten Bytes pro Rechenzeit (unter der Annahme, dass die Kompression einer Bildmenge von n Bildern Rechenzeit in der Größenordnung n^2 benötigt). „Genetic2“ erstellt ausgewogenere Cluster als „Genetic“, und benötigt demzufolge insbesondere auch weniger Zeit für das Clustering selber (da die Bewertung von ausgewogeneren Clustern auch selber wieder schneller ist).

4.3.9 Perfektes Clustering

Alle bisher vorgestellten Algorithmen optimieren anhand bestimmte Kriterien gewisse Eigenschaften der zu erstellenden Clustermenge. z.B. optimiert nCut das Verhältnis von Gewichten innerhalb und außerhalb des Clusters, MST ein graphtheoretisches Kriterium, etc.

Wenn in der Anwendung allerdings ein konkretes globales Qualitätsmaß (wie

hier die endgültige Dateigröße) vorliegt, dann lässt sich natürlich auch die Menge der „besten“ Cluster berechnen bzw. angeben.

Dies ist insbesondere sinnvoll für den Vergleich bzw. Test von Algorithmen auf kleinen Bildbeständen, da dadurch eine endgültige Ober- bzw. Untergrenze für die Qualität (hier: die Dateigröße) einer Clustermenge angegeben werden kann.

Auf großen Bildbeständen (und insbesondere komplexen Distanzmetriken bzw. Kompressionsalgorithmen) lässt sich das perfekte Clustering natürlich praktisch nicht mehr berechnen.

Die Basis der Berechnung bildet die bereits vorgestellte b^+ Metrik, die für jeden Cluster angibt, wie viele Bytes gespart wurden gegenüber einer Einzelbildkompression von allen Bildern. D.h. für eine Menge von Bildern $\mathcal{I} = I_1, I_2, \ldots, I_n$ und k Cluster $C_i = \{I_{i,1}, I_{i,2}, \ldots I_{i,n_i}\} \subset \mathcal{I}, i \in 1 \ldots k$ gibt

$$r = \sum_{i=1\ldots k} b^+(I_{i,1}, I_{i,2}, \ldots, I_{i,n_i})$$

die Anzahl der insgesamt durch die Bildung der k Cluster $C_1, \ldots, C_k$ eingesparten Bytes an.

Da für die Cluster $C_i, i \in 1 \ldots k$ gilt:

$$C_i \cap C_j \neq \emptyset \rightarrow i = j$$

sowie

$$\bigcup_{i \in 1\ldots k} C_i = \mathcal{I}$$

ist jeder Cluster gleichzeitig eine Kante des Hypergraphs auf den Bildern $\mathcal{I}$, und die Menge der Cluster $\mathcal{I}$ ein **Matching** (d.h. eine Überdeckung aller Knoten/Bilder durch eine Menge von Kanten) auf diesem Hypergraphen, siehe Abbildung 4.15.

Ermittelt man nun den insgesamt größten möglichen Wert eines solchen Matchings, so entspricht die Summe der Kantengewichte genau dem Maximalbetrag an Bytes, die man durch „geschickte“ Zusammenfassung von Bildern sparen kann. D.h. das maximale Matching auf dem durch b^+ gewichtete Hypergraph gibt den bestmöglichen Kompressionsgewinn an.

Maximales Matching auf Hypergraphen ist ein bekannter Algorithmus aus der Graphentheorie [23], allerdings ist dieser leider NP-vollständig, und benötigt auch in jedem Fall die gesamte Menge aller $n^k - 1$ Kantengewichte,

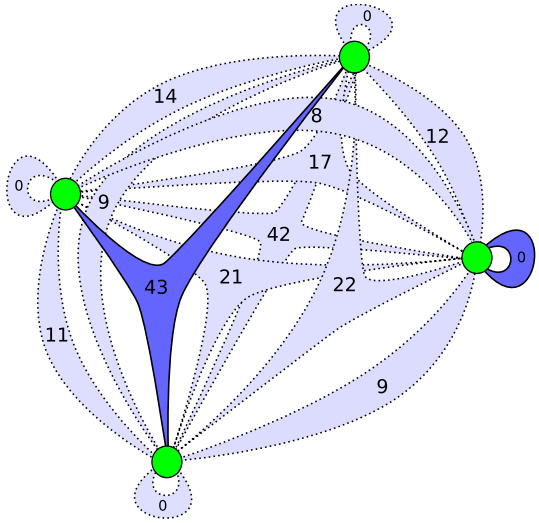


Abbildung 4.15: *Ein maximales Matching für den Hypergraph aus 4.1. Durch die Überdeckung aller Knoten (Bilder) durch Hyperkanten (komprimierte Bildmengen) werden die Bilder so komprimiert, dass sich der maximale Kompressionsgewinn (hier:* $43 + 0$*, also* 43*) ergibt*

weshalb man diesen Algorithmus nur zu Testzwecken auf sehr kleinen Bildbeständen (in der Größenordnung von ca. 10 Bildern) praktisch anwenden kann.

4.4 Bestimmung der Qualität von Clusteringalgorithmen

Wie in der Einleitung dieses Kapitels beschrieben, kann man anhand des insgesamt für die Speicherung eines Bildbestandes benötigten Speichers sehr gut verschiedene Clusterergebnisse miteinander vergleichen. Hierbei sieht man dann das Ergebnis eines Clustering-Verfahrens, welches weniger Platz benötigt, als besser an.

Um derartige Vergleiche durchzuführen, benötigt man allerdings immer einen Kompressionsschritt eines (möglicherweise sehr großen) Clusters, was bei manchen der hier vorgestellten Algorithmen (z.B. fraktale Kompression) durchaus einiges an Berechnungszeit mit sich ziehen kann.

Dies ist nicht zu verwechseln mit Bildähnlichkeitsrelationen, die sehr gut

approximiert werden können — in realen Tests möchte man aber zumeist den konkreten Wert (sprich die tatsächlich gesparten Bytes) eines gebildeten Clusters berechnen, um Algorithmen realistisch vergleichen zu können.

Deswegen kann es von Interesse sein, die Algorithmen mittels einfacherer Kompressionsalgorithmen sowie auf „schnellen“ Datenbeständen zu testen, die aber möglichst gut die Eigenschaften der „großen“ Algorithmen approximieren sollen. Im Folgenden sind deswegen drei Methoden beschrieben, mit denen das Verhalten der Distanzrelation von Bildkompressionsalgorithmen approximiert werden können, und die auch im nachfolgenden für die Messung von Effizienz von Clusteringalgorithmen herangezogen werden.

4.4.1 Teilmengen

Eine einfache — und nicht zu unterschätzende — Methode, um Clusteringalgorithmen zu evaluieren, stellt die Anwendung auf kleine Teildatensätze eines Datenbestandes dar. Die dadurch gewonnenen Erkenntnisse können i.d.R. dann auf größere Datenbestände „extrapoliert“ werden.

Dies ist analog zu Methoden in der Clusteranalyse, in denen (z.B. beim k-Means Algorithmus) eine zufällige Anzahl von Punkten aus dem zu verarbeitenden Datensatz selektiert werden, um nur auf diesem Teildatensatz eine Clusteranalyse durchzuführen, und den eigentlichen Datensatz erst danach auf den kleinen Datensatz zu „mappen“ [41].

Wenngleich man nun bei der Evaluation eines Algorithmus diesen letzten Schritt (das „Mappen“ auf den kleineren Datensatz) nicht durchführt, so kann man doch aus der Tatsache, dass bei k-Means eine Teilmenge der eigentlichen Punkte ausreicht, um Aussagen über Clusterbildung treffen zu können, die Vermutung ableiten dass dementsprechend auch für die Evaluation von Clusteringalgorithmen derartige Teilmengen ausreichen sollten.

Insbesondere hat man bei dieser Art der Approximation den Vorteil, dass sich die Metrik, auf der das Clustern passiert, genauso verhält wie der „eigentliche“ Kompressionsalgorithmus, ein Vorteil der bei den in den nachfolgenden Unterkapiteln vorgestellten Verfahren nicht unbedingt mehr gegeben ist.

Ein Nachteil stellt allerdings dar, dass sich Clusteralgorithmen bei größeren Datensätzen, für die das Verfahren dieses Kapitels zu aufwendig ist, eventuell anders verhält — darüber kann man dann letztlich nur Vermutungen anstellen.

Ein weiteres Problem ist die Berechnungszeit — auch wenn man nur auf einer kleinen Bildmengen arbeitet, muss man doch eine Reihe von „normalen“ Kompressionsschritten durchführen, was bei einigen der komplexeren Algorithmen eine nicht zu unterschätzende Rechenzeit nach sich zieht.

Dementsprechend gehen die nächsten beiden Kapitel darauf ein, wie man die entsprechenden Kompressionsschritte umgeht, und die Kompressionsdistanz zwischen Bildern auf Basis anderer Verfahren nur abschätzt.

4.4.2 Token-Sets

Um Kompressionsstatistiken abstrakt zu modellieren, hilft die folgende Überlegung: Damit zwei Bilder gut miteinander komprimieren, benötigen sie irgendeine Form von gemeinsamen statistischen Elementen — dies können, im Fall von klassischer verlustloser Kompression, gemeinsame Strings sein, oder, im Fall von fraktaler Kompression, ähnliche Bildausschnitte, oder gemeinsame Texturen bei der Polygonkompression, etc.

Gleichzeitig haben Dateien, die schlecht miteinander komprimieren, meist solche Gemeinsamkeiten nicht. D.h. Strings, Bildausschnitte oder Texturen unterscheiden sich hier, und haben wenig Gemeinsamkeiten.

Diese gemeinsamen Elemente werden im Nachfolgenden „Tokens“ genannt. D.h. wenn zwei Kompressionsobjekte ein gemeinsames „Token“ haben, so ist ein Teil ihrer (wie auch immer gearteten) Datenstruktur identisch, und muss im Falle einer Zusammenfassung der beiden Dateien nur noch einmal gespeichert werden, anstatt zweimal.

Um dies zu modellieren, gehen wir vereinfacht von einer endlichen Menge von Tokens $t_1, \ldots, t_n \in \mathcal{T}$ aus. Jedem „Bild“ wird nun des Weiteren durch eine Wahrscheinlichkeitskonstante p eine bestimmte Anzahl von Tokens zugewiesen:

$$P(t_i \in I_j) = p$$

Gleichzeitig hat jedes Token t_i eine Größe $size(t_i)$. Dies ist z.B. die Länge eines Strings, oder die Anzahl Bytes, die benötigt wird um eine Textur zu beschreiben.
Zusätzlich hat noch jede Datei I_j eine „Basisgröße“ $size(I_j)$, die die Menge an nicht komprimierbaren Elementen oder Eigenschaften der Datei modelliert.

Die Kompressionsgröße eines Clusters $C = I_1, I_2, \ldots, I_k$ ist nach diesem Mo-

dell dann:

$$\sum_{I \in C} size(I) + \sum_{t \in I_1 \cup I_2 \ldots I_k} size(t)$$

Dass heißt, man geht davon aus, dass jedes in einem Cluster vorkommende Token insgesamt nur einmal im gesamten Cluster gespeichert werden muss, was intuitiv der Vorgehensweise der bisher vorgestellten Kompressionsalgorithmen entspricht.

Durch dieses Modell lassen sich relativ einfach und effizient Metriken auf Graphen und Hypergraphen aufbauen, mit denen Clusteralgorithmen für ihren Einsatz auf Kompressionsdistanzen getestet werden können. Insbesondere lässt sich diese Statistik extrem schnell auswerten.

4.4.3 zlib auf Textbausteinen

Die Eigenschaften von Bildkompression lassen sich noch genauer approximieren als mittels der gerade vorgestellten Token-Statistik, indem man einen echten Kompressionsalgorithmus benutzt, und diesen statt auf (großen) Bildern auf kleinen alternativen Datensätzen laufen lässt.

Insbesondere der auf PNG Bildern verwendete zlib-Algorithmus (siehe Kapitel 2.1.3) bietet sich hier an, da er problemlos auch z.B. Textstrings komprimieren kann.

In der im nachfolgenden Kapitel vorgenommenen Auswertung wurde zlib auf Absätze eines in englischer Sprache verfassten Romans angewendet. Alle in Kapitel 4.1 beschriebenen Kompressionsdistanzmetriken lassen sich auch auf diese Datensätze anwenden, was Tests von sehr großen Clustermengen möglich macht.

In Abbildung 4.16 sind einige Beispiele der b^+ Distanzmetrik zwischen Textblöcken eingezeichnet. Zusätzlich ist noch zu Illustrationszwecken in Abbildung 4.17 das Clustering eingezeichnet, das die Daten optimal (d.h., am Platzsparensten) in zwei Cluster zerlegt. Die Berechnung dieses Clusterings erfolgte durch den Algorithmus aus Kapitel 4.3.9.

Im folgenden wird nun anhand dieser Metrik untersucht, wie gut die verschiedenen Clustering-Algorithmen in der Praxis arbeiten.

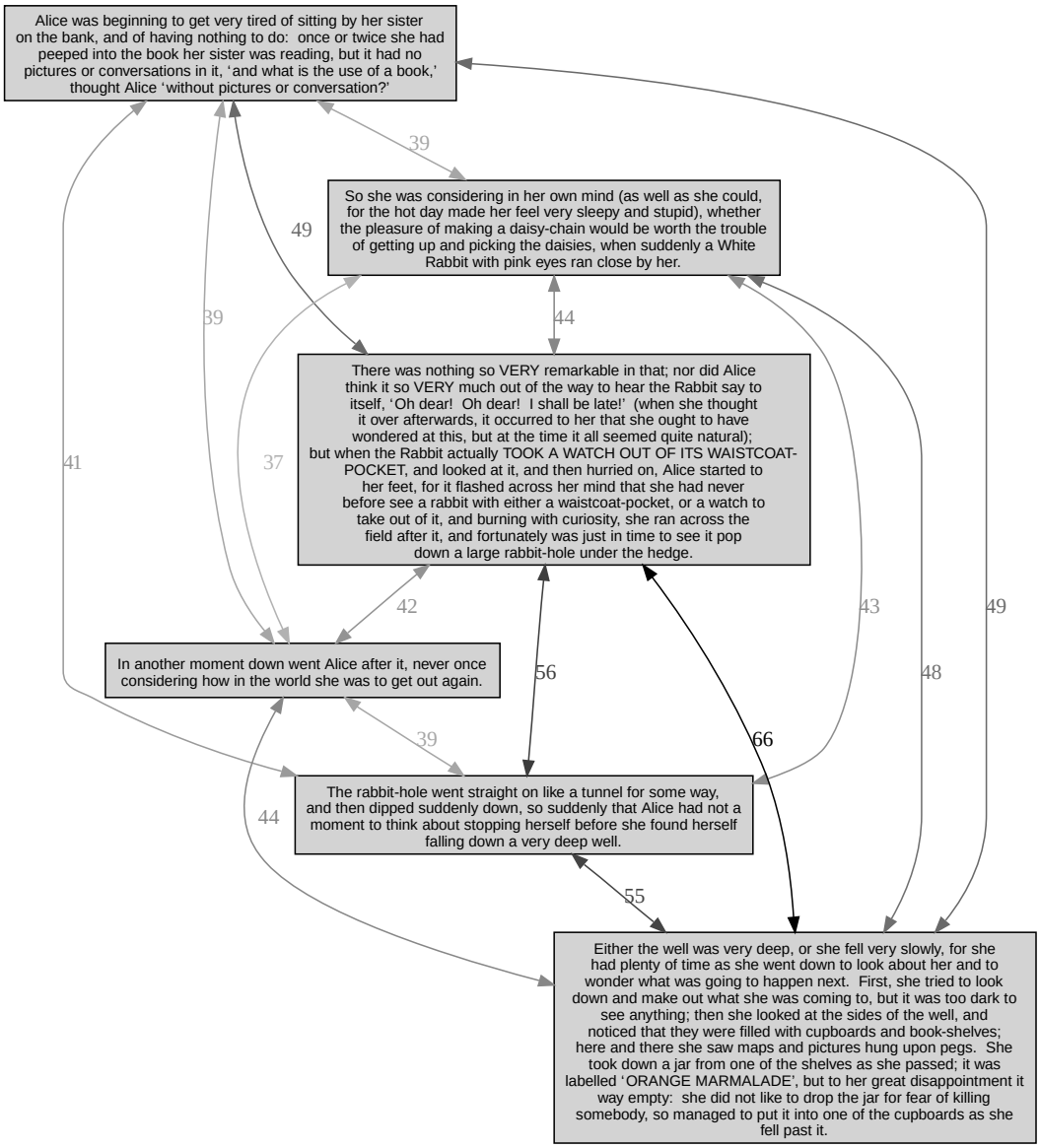


Abbildung 4.16: *Textblöcke sowie b^+ Distanzrelationen— zu jeder Node (jedem Textblock) ist an der Kante die Anzahl der Bytes angegeben, die man spart, wenn man die beiden Blöcke mit zlib zusammen komprimiert statt einzeln. z.B. spart eine gemeinsame Kompression der obersten beiden Textstücke 39 Bytes.*

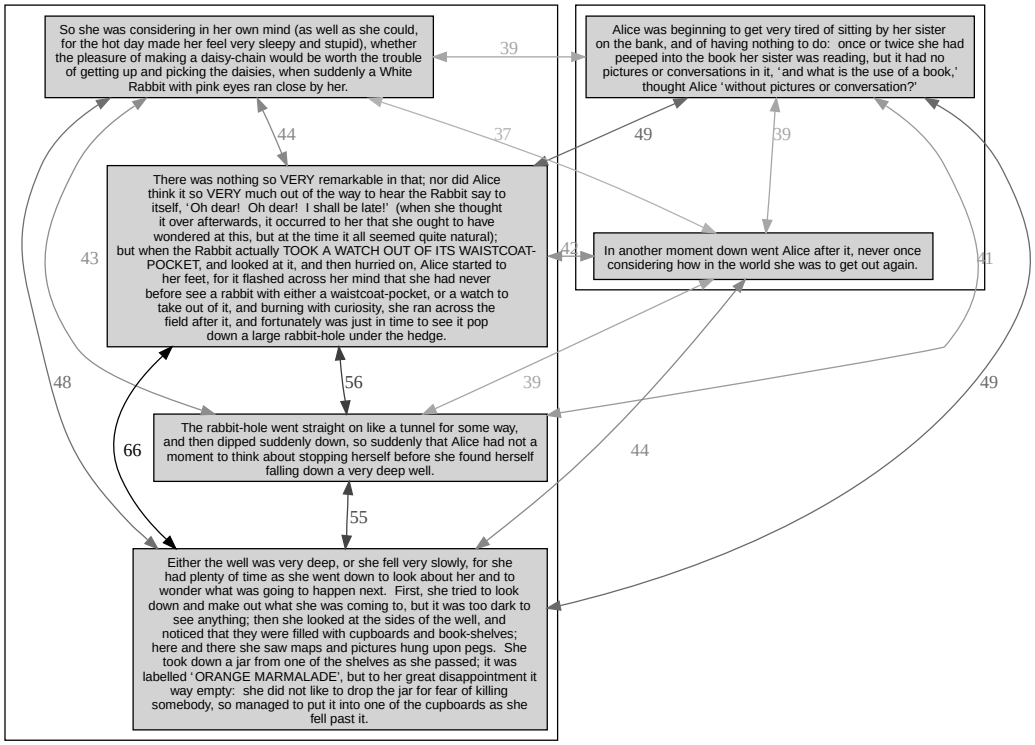


Abbildung 4.17: *Optimales Clustering des Graphs 4.16 in zwei Partitionen. Der eigentliche Kompressionsgewinn der 6 Textblöcke lässt sich danach über die Summe der Gewichte aller Kanten, die im Inneren von Clustern verlaufen (hier: 351), abschätzen. (Der „reale“ Kompressionsgewinn ist in diesem Beispiel 281)*

4.5 Vergleich der Clustering-Algorithmen

Um einen Eindruck zu gewinnen, wie die bisher vorgestellten Clustering-Algorithmen in der Praxis, sprich an realen Daten, arbeiten, wurde einmal für 128 Nodes und einmal für 512 Nodes ein komplettes Clustering durchgeführt. In Abbildung 4.18 sind die Auswertung sowie die jeweiligen (gesparten) Dateigrößen zu sehen. Ebenfalls angegeben sind die (reziproken) Berechnungszeiten, die sich folgendermaßen ergeben: Für jeden Cluster der Größe n, für den eine Gewichtung vorgenommen werden muss, wird n^2 zur Berechnungszeit addiert. Schließlich müssen alle bisher vorgestellten erweiterten Kompressionsalgorithmen zur Kompression von n Bildern bei jedem einzelnen Bild n andere Bilder berücksichtigen, wodurch sich ein Berechnungsaufwand von n^2 ergibt.

Optimal arbeitet in Bezug auf Clusterberechnungszeit sowie Ausgewogenheit der einzelnen Cluster natürlich der „Random“ Algorithmus, der das Bild zufällig auf Cluster verteilt, da er keinerlei Berechnungen zur Bildung der eigentlichen Cluster durchführen muss.

Ansonsten haben bezüglich Kompressionsgewinn aber die Algorithmen, die mit Gewichten auf Hyperkanten arbeiten (und dadurch auch mehr Information zur Verfügung haben) bei der Größe der endgültigen Cluster die Nase vorn, wie z.B. SAHN oder Genetische Algorithmen — allerdings auf Kosten eines angestiegenen Zeitaufwands für das eigentliche Clustering.

Von den „nur“ auf Basis von 1:1 Bildvergleichen arbeitenden Algorithmen erzielen die „balanced“ MST-Variante sowie kMeans vernünftige Ergebnisse bezüglich gesparter Bytes sowie Ausgewogenheit der Cluster.

In der im nächsten Kapitel vorgestellten Softwaresystem werden balancierte Genetische Algorithmen (Genetic2) verwendet, da sie ein „gesundes Mittelmaß“ an Ausgewogenheit der Cluster sowie Zeitaufwand und Kompressionsgewinn bieten, und des Weiteren noch andere Vorteile haben — insbesondere dass ein einmal gewonnenes Clustering weiter verfeinert werden kann, sowie dass „Modifikationen“ (Löschoperationen, Hinzugefügte Bilder) an Clustern durch weitere Rechenschritte „ausgebügelt“ werden können. Für die einmalige (nicht iterative) Kompression wird des Weiteren SAHNb verwendet, welcher den Vorteil hat, dass die Granularität der Bildmengen in den ersten Berechnungsschritten des Algorithmus klein bleibt. (D.h. die Größe der Bildmengen und damit die Berechnungszeit ist durch die Laufzeit, die dem Clustering-Algorithmus zugeteilt wird, steuerbar.)

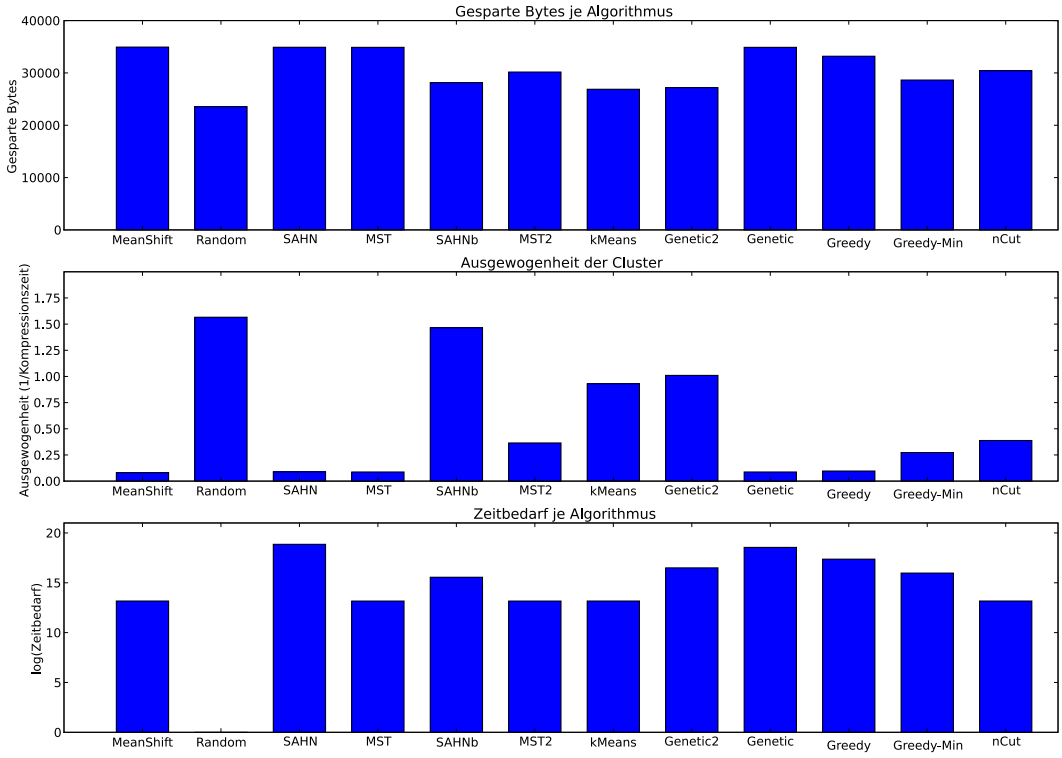


Abbildung 4.18: *Vergleich der verschiedenen Clusteralgorithmen. Oben: Gesparte Bytes pro Clusteralgorithmus. Eine große Anzahl von Algorithmen (MST, SAHN) erreicht hier gute Werte durch sehr unausgewogene Cluster, was in der Grafik in der Mitte deutlich gemacht wird. Diese gibt das reziproke des Zeitbedarf an, den eine Kompression des errechneten Clusters kosten würde, d.h. wie ausgewogen die Cluster sind. Unten ist zusätzlich noch der (logarithmische) Zeitaufwand aufgetragen, den der Clusteralgorithmus selber benötigt hat.*

Kapitel 5

Praxisbeispiel: Integration der neuen Algorithmen

5.1 Die mediaTUM Bilddatenbank der TU-München

Die hier beschriebenen Algorithmen wurden in das mediaTUM System, einer zentralen Bild- und Multimedia-Datenbank an der Technischen Universität München, integriert. mediaTUM wurde parallel zu dieser Arbeit entwickelt, und speichert nun zentralisiert Bilder, Videos und Dokumente der Hochschule.
Es ist streng genommen mehr als ein reines Bildverwaltungs-System, u.A. fungiert es mittlerweile als Open-Access Repository, Veröffentlichungsarchiv, Videoplatform sowie Dokumenten-Server. Auf der lokalen Installation der TU-München sind allerdings mit rund 26.000 Fotos (und nur knapp 5000 Dokumenten) Bilder das zentrale Datenformat.

mediaTUM benutzt zur Speicherung der Bilddaten optional einen oder mehrere der hier vorgestellten Kompressionsverfahren. (Fraktale Mehrbildkompression, KLT-Bildgruppenkompression oder Texturdatenbanken. Bei verlustfreier Kompression wird der erweiterte PNG-Algorithmus verwendet)

Im folgenden werden die wichtigsten Eigenschaften des mediaTUM-Systems erläutert. Genauere Details sind [83]) zu entnehmen.

In Kapitel 5.2 wird schließlich darauf eingegangen, inwieweit der mediaTUM-

Bilddatenbestand der TU-München mit den hier vorgestellten Algorithmen komprimiert werden konnte.

5.1.1 Architektur

mediaTUM ist in Python implementiert, und stellt alle Verwaltungsfunktionalität über ein Web-Frontend bereit.

Im Backend arbeitet eine SQL-Datenbank (sqlite), eine Volltext-Suchmaschine (magpy oder fts3) sowie diverse Bild- und Dokumentkonvertierungs–Plugins (u.A. die Python Imaging Library, swftools, xpdf sowie einige andere). Die genaue Architektur ist in Abbildung 5.1 aufgezeigt.

Wichtig ist hier insbesondere das bereits erwähnte Plugin-System, mittels dem man zusätzliche Komponenten an das System anschließen kann, die dann z.B. neue Objekttypen, Berechtigungsstrukturen oder — hier — Datenspeicherstrategien implementieren.

5.1.2 Datenmodell

mediaTUM legt Bilder in einer hierarchischen, Dateisystem-ähnlichen Struktur ab, in dem sich sowohl die mediaTUM-Objekte als auch zur Verwaltung verwendete abstrakte Objekte, wie virtuelle Directories, Metadaten-Informationen, Berechtigungsregeln sowie Userdaten befinden.

Die Struktur, die mediaTUM benutzt, ist hierbei ein „DAG“ (directed acyclic graph), sprich eine Art Baumstrukur, bei der die Blätter des Baumes allerdings an mehreren übergeordneten Nodes hängen können. (Um es zu ermöglichen, dass ein Bild in mehr als ein virtuelles Directory „einsortiert“ wird.) Dieses Datenmodell ist rein „virtuell“, sprich spiegelt nicht die tatsächliche Einordnung der Bilder im Filesystem (oder in Cluster im Kompressionsalgorithmus) wieder. Das hierarchische Datenmodell ermöglicht u.A. Rechteverwaltung, Kollektions-zuweisung von Bildern u. Dokumenten sowie Suchanfragen auf Teildatenbestände.

Bei der Anfrage eines Users auf beispielsweise ein Bild wird sowohl die Position dieses Bildobjekts als auch der jeweilige Datentyp abgefragt. Die eigentliche Auslieferung der Bilddaten läuft dann (bei Vorhandensein einer entsprechenden Erweiterung) über die Plugin-Schnittstelle (siehe Abbildung

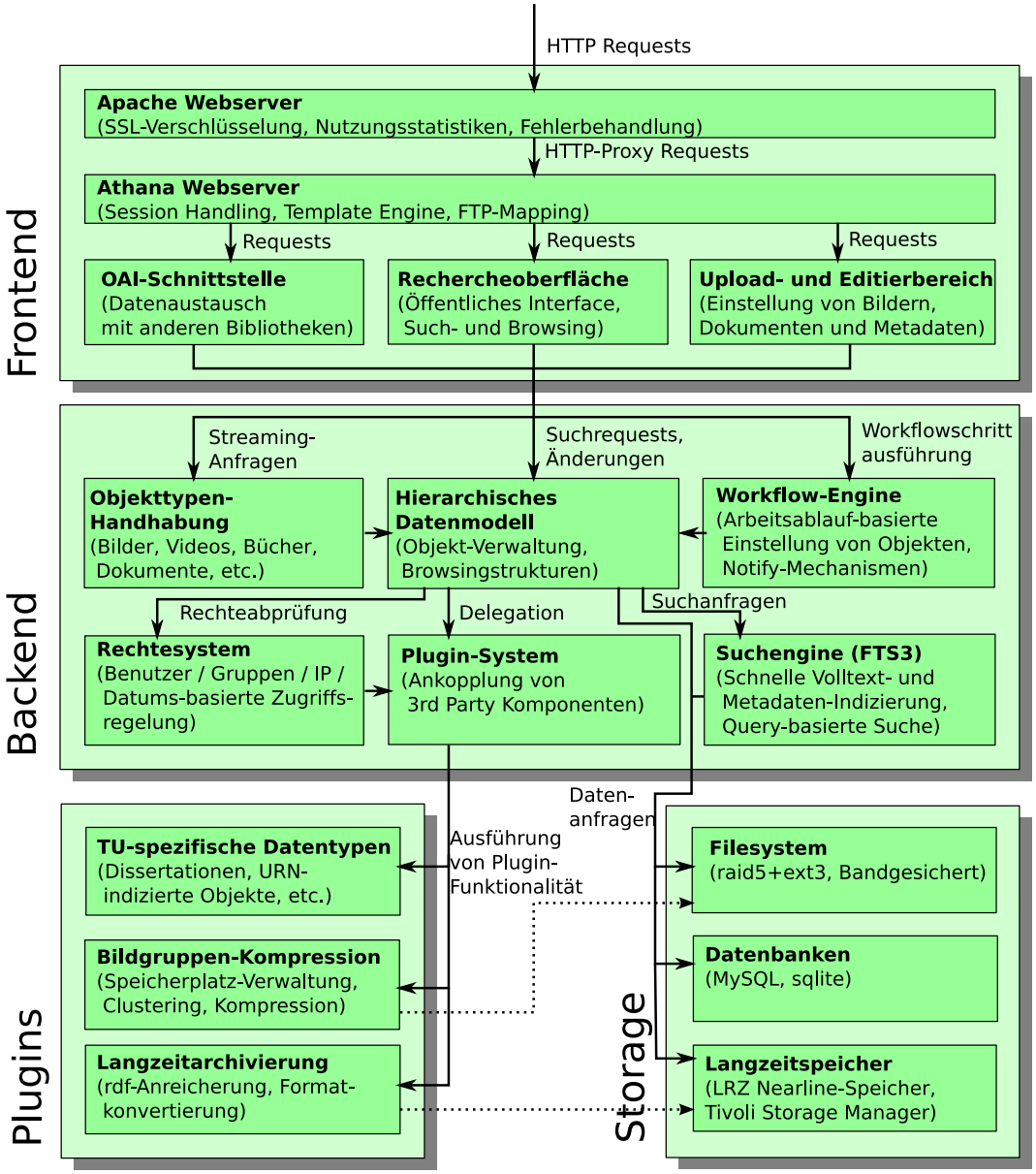


Abbildung 5.1: *Die wichtigsten Komponenten der mediaTUM-Architektur*

5.1), an die die in dieser Arbeit vorgestellten Bildverwaltungs-Komponenten gekoppelt sind. Momentan hat das laufende System rund 1000 Bildobjektanfragen pro Stunde.

5.1.3 Features

mediaTUM generiert für jedes Bild zusätzlich eine Reihe von Thumbnails, damit in Übersichtsseiten Bilder klein nebeneinander abgebildet werden können, siehe Screenshot 5.2. Diese Thumbnails werden, um die Ausführungsgeschwindigkeit von mediaTUM zu steigern sowie die Bilddatenbank nicht zu sehr zu belasten, auf „traditionelle“ Art und Weise gespeichert, d.h. immer normal im Dateisystem abgelegt, und nicht durch eventuelle Dateikompressions oder -archivierungs Plugins bearbeitet.

Des Weiteren besitzt das System die Möglichkeit, über eine Web-Schnittstelle ZIP-Archive von Bildern hoch zu laden und ins System einzuspeisen. Hierbei werden auch automatisch Metadaten aus den Bildern (wie z.B. EXIF oder iptc Daten [94]) extrahiert. Da die Metadaten von Bildern separat abgelegt werden, ist es möglich, bei der Speicherung von Bildern auf dem Datenträger eine Formatkonvertierung durchzuführen, ohne diese EXIF oder iptc Metadaten in das neue Bildformat „mitziehen“ zu müssen.

5.2 Datenbestand in mediaTUM

Den Großteil des Speicherplatzes in der mediaTUM-Installation der TU-München beanspruchen im „Original“ Datenbestand die Bilder des Lehrstuhls für Landtechnik im Wissenschaftszentrum Weihenstephan, siehe Abbildung 5.3. Diese sind bisher mittels des TIF-Algorithmus (siehe Kapitel 2.1.5) verlustlos komprimiert.
Da es wünschenswert ist, verlustlose Bilder weiterhin mit einem verlustlosen Codec zu komprimieren, wurde in der Mehrbildkompression-Variante für diese Bilder der erweiterte PNG-Algorithmus gewählt, siehe Abbildung 5.4.

Einen deutlich kleineren Anteil machen verlustbehaftet komprimierte Bilder aus. Diese wurden in der erweiterten mediaTUM Variante mittels des fraktalen Algorithmus komprimiert — auch KLT Kompression wäre hier möglich gewesen. Die Wahl fiel auf die fraktale Kompression, da der Originaldatenbestand bereits verlustbehaftet JPEG komprimiert war, und die vorhandenen

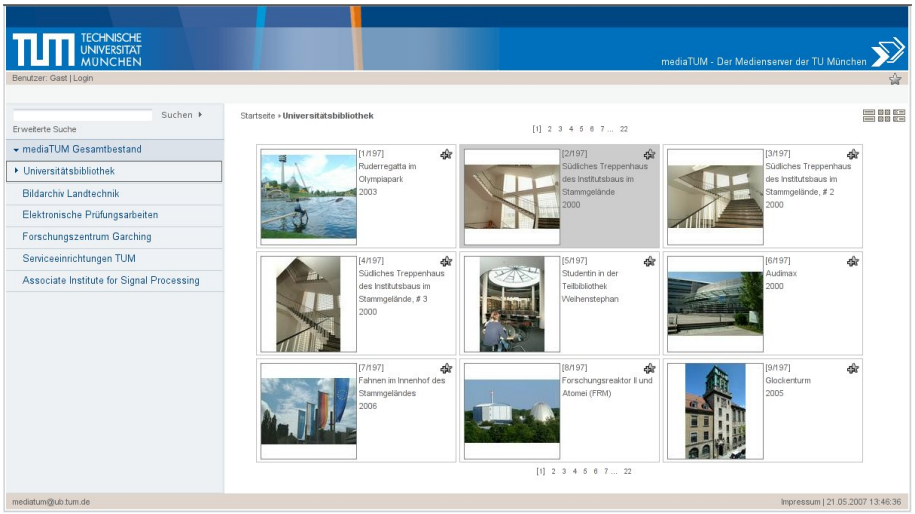


Abbildung 5.2: *Ein Screenshot der Einstiegsseite von mediaTUM*

Kompressionsartefakte (Ringing und Blocking) durch einen Algorithmus derselben Familie (wie KLT) verstärkt worden wäre — deswegen erzeugt fraktale Kompression auf diesen Daten bessere Qualität. Bei der verlustbehafteten Kompression muss natürlich ein Qualitätsfaktor festgesetzt werden. Dieser wurde hier analog zu den JPEG-Originaldaten gewählt, d.h. es wurde bei jedem Bild die jeweilige JPEG-Qualität (gemessen in maximaler Abweichung pro Pixel) auf den fraktalen Algorithmus übertragen.

Das Clustering der Bilder (sowohl der verlustfreien als auch der verlustbehafteten Gruppe) fand über den SAHNb Algorithmus (siehe Kapitel 4.3.2) statt, auf Basis der in Kapitel 4.2.1 motivierten Ähnlichkeitsmetriken. Ein paar typische Cluster von gemeinsam komprimierten Bildern sind in Abbildung 5.5 zu sehen. mediaTUM benutzt wahlweise SAHNb oder genetische Algorithmen zum Ermitteln von guten Clustern, je nachdem ob die Bilder iterativ zum Datenbestand hinzugefügt werden, oder (wie hier) ein bereits bestehendes Bildarchiv „nachkomprimiert“ wird.

Insgesamt würde in dieser Beispieldatenbank durch die neue Kompresssion der Bilder ein Datenvolumen von 205 Gb gespart, wovon ein Großteil durch den Wechsel vom (relativ ineffizienten) TIF-Kompressionsverfahren hin zu (Mehrbild-)PNG gewonnen wurde. Der „Mehrwert“ der Mehrfachbildkompression ist hier bei den verlustlosen Daten 0,18 Gb (im Vergleich zu „normalem“ PNG) und bei den verlustbehafteten Daten 0.7 Gb.

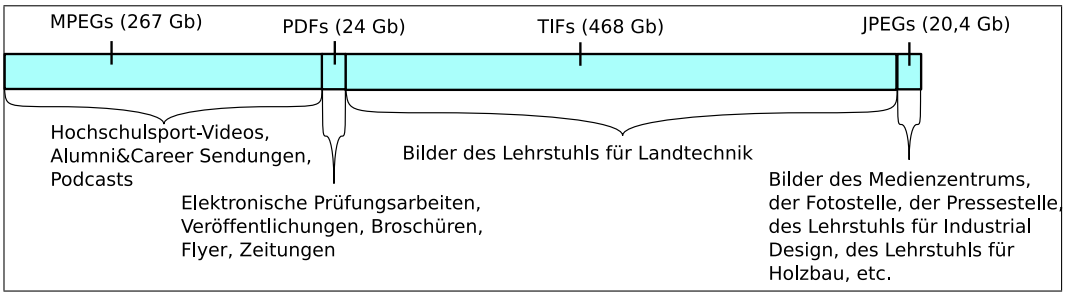


Abbildung 5.3: *Die Dateiformate sowie der Speicherplatzverbrauch in mediaTUM im "Originaldatenbestand".*

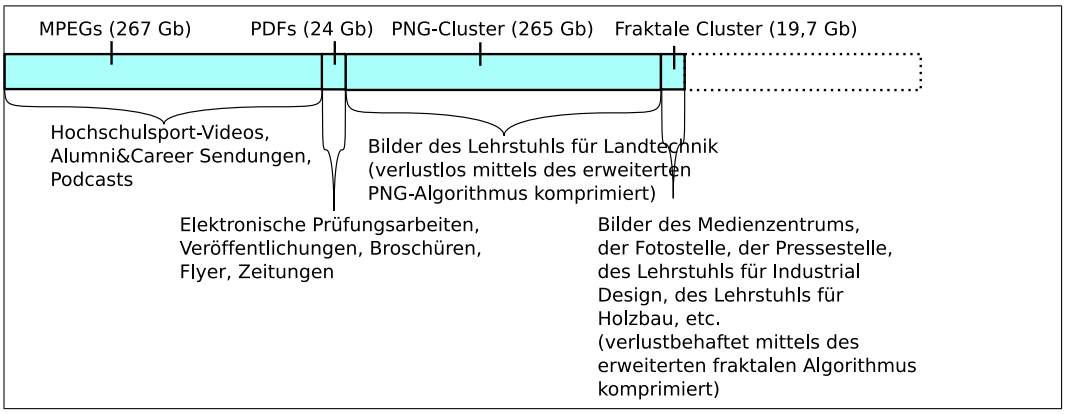


Abbildung 5.4: *Die Dateiformate sowie der Speicherplatzverbrauch in mediaTUM in einer Instanz, in der mit den hier vorgestellten Algorithmen gearbeitet wurde.*

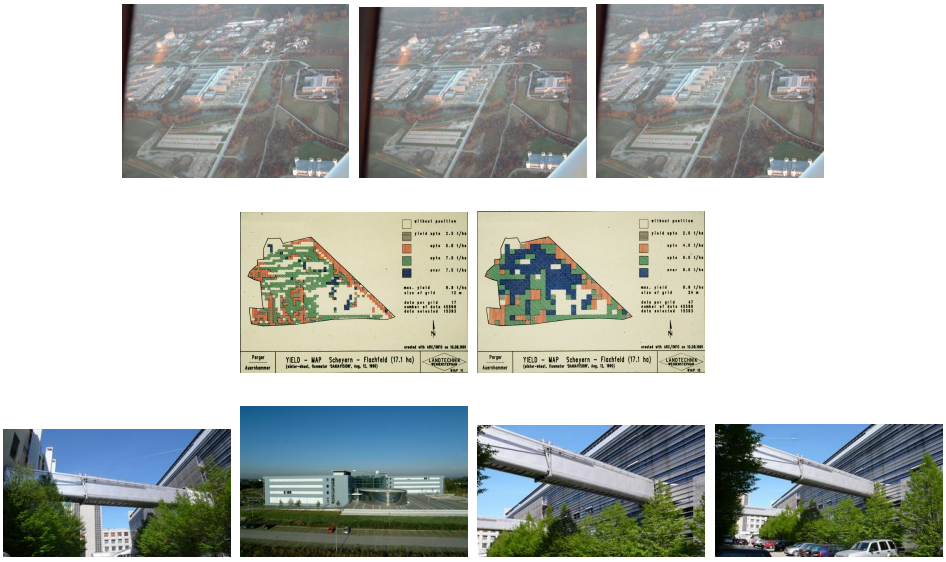


Abbildung 5.5: *Eine Reihe von „typischen“ Bildmengen von ähnlichen Bildern, die zusammen komprimiert wurden.*

Kapitel 6

Zusammenfassung

In dieser Arbeit wurden Verfahren für die Kompression von Bildmengen in großen Multimediadatenbanken untersucht, unter Ausnutzung von redundanten Bildgruppen, sprich Teilmengen von sehr ähnlichen Bildern in einer derartigen Datenbank.

Zu diesem Zweck wurden mehrere neue Kompressionsalgorithmen präsentiert, die solche Mengen von ähnlichen Bildern besser komprimieren können als die nur auf Einzelbildern arbeitende Verfahren.

Der in Kapitel 3.1 beschriebene Algorithmus ist eine Neuentwicklung, die das (bisher hauptsächlich in der Theorie existierende) Prinzip der fraktalen Kompression auf Bildmengen ausdehnt. Dieser neue Algorithmus liefert bei genügend langer Rechenzeit sehr gute Kompressionsraten.

Auch der in Kapitel 3.3 vorgestellte Algorithmus ist ebenfalls eine neue Entwicklung, die darauf basiert, aus Bildmengen eine optimale Basis für Bildtransformationen zu ermitteln. Dieser Algorithmus erzielt auf Bildmengen bessere Kompressionsraten als beispielsweise JPEG, arbeitet aber trotzdem sowohl bei der Kompression als auch bei der Dekompression sehr performant.

Der in Kapitel 3.4 vorgestellte Algorithmus ist dementgegen „nur“ eine Erweiterung des standardisierten Dateiformat PNG. Hier wurde die (verlustlose) normale PNG-Kompression insofern erweitert, dass sich damit auch Gruppen von Bildern komprimieren lassen. Da die Dekompression der so komprimierten Bilder mit der „normalen“ PNG-Implementierung stattfinden kann, ermöglicht diese Vorgehensweise eine sehr transparente Handhabung

von Bildmengen.

Ein wiederrum neuer Algorithmus wurde in 3.5 vorgestellt, bei dem die Kompression von Bildgruppen durch Ermittlung ähnlicher Texturen stattfindet. Für diesen Algorithmus wurde sowohl eine neue Texturanalyse und -synthese als auch eine neue Bildsegmentierung entwickelt und beschrieben.

Zusätzlich zu der Entwicklung der neuen Kompressionsalgorithmen wurde auch gezeigt, wie sich verschiedene Ähnlichkeitsmetriken zwischen Bildern sowie Clusteringalgorithmen dazu verwenden lassen, Bildgruppen aus einer Bilddatenbank zu extrahieren, für die die neuen Kompressionsverfahren möglichst gewinnbringend arbeiten. Neben der Vorstellung und Evaluation von einer Reihe von „klassischen“ Clustering-Algorithmen (MST, SAHN, k-Means, MeanShift, nCut) wurde auch ein neuer (genetischer) Clustering-Algorithmus entwickelt, der gegenüber den anderen Algorithmen einige Vorteile bietet, wie z.B. iterative Updates von Clustering-Konfigurationen.

Des Weiteren wurden die Implementierungsdetails eines parallel zu dieser Arbeit implementierten Bildverwaltungssystems aufgezeigt, in das die vorgestellten Algorithmen integriert wurden und dort für eine messbare Verringerung des Speicherbedarfs gesorgt haben.

Kapitel 7

Ausblick

Wenngleich im Rahmen dieser Arbeit bei allen der hier vorgestellten Algorithmen schon eine komplette Implementierung und Integration vorgenommen wurde, sprich die einzelnen Kompressionsverfahren in einer funktionsfähigen Variante implementiert wurden und in ein eigenes Bildverwaltungssystem Eingang gefunden haben, so gibt es doch natürlich noch viele Möglichkeiten für Verbesserungen und weitere Arbeiten.

Insbesondere lassen sich die vorgestellten Algorithmen durchaus noch verfeinern — beispielsweise bei der fraktalen Kompression und der Texturanalyse würde hier eine genauere Betrachtung der Streameigenschaften der verschiedenen Kodierungsparameter (sprich der Texturen, oder der fraktalen Abbildungen) noch Spielraum bieten, um die Kompressionseffizienz weiter zu erhöhen.

Da sich bei der Analyse der bestehenden Bilddatenbank in Kapitel 5.2 gezeigt hat, dass auch die verlustlose Kompression von Bildern sehr wichtig ist, und verlustlose Bilder zumindest in diesem Fall einen Großteil des Datenbestandes ausmachen, bietet es sich an, noch weitere Verfahren zu entwickeln die Bildmengen verlustfrei komprimieren können. Nennenswert ist in diesem Zusammenhang auch die in Kapitel 3.5.4 angedachte Nachkompression von JPEG-Bildern, die es ermöglichen würde, die verlustbehaftete „Vorkompression“ von JPEG auszunutzen und verlustlos darauf aufbauend zusätzlich Redundanzen in Bildmengen auszunutzen.

Jenseits der Datenkompression eignen sich die vorgestellten Implementierungen aber auch, um in neue Forschungsbereiche vorzudringen. Das „duale

Problem“, die Analyse von Ähnlichkeiten durch Kompression, ist ein neuer und vielversprechender Forschungszweig [97]. Auch die Analyse der Bildqualitätswahrnehmung von Menschen in Bezug auf Texturen und Texturähnlichkeiten ist bisher noch unerforscht (siehe auch [55]), und kann durch den funktionieren Segmentierungs- und Texturzerlegungs-Codec dieser Arbeit, mit dem sich Testbilder in beliebiger Quantität produzieren lassen, genauer in „Augenschein“ genommen werden.

Abbildungsverzeichnis

1.1 Zwei ähnliche Bilder aus der Bilddatenbank der TU-München 8

2.1 Arbeitsweise von LZ78 und LZW 13

2.2 Prediktoren in PNG Bildern 14

2.3 PNG gefiltertes Bild 15

2.4 Vergleich von LZ77 und RLE 16

2.5 DCT Frequenzanteile in Bildern 22

2.6 Vergleich Fourier und Gabor 23

2.7 Wavelet-Transformation eines Bildes 24

2.8 JPEG-Artefakte 26

2.9 Bewegungsvektoren in der Videokompression 28

2.10 Steerable Pyramid 31

2.11 Beispiel einer Filterbank 33

2.12 Zerlegung eines Bildes in Textons 34

2.13 Kompression eines Bildes mittels Textons 35

2.14 Ein Mandelbrot-Fraktal 37

2.15 Beispiel einer fraktalen PIFS Abbildung 38

2.16 Textursynthese nach Portilla & Simoncelli 42

2.17 Polygonalisierung von zwei Bildern 44

2.18 Zerlegung eines Bildes in Segmente 46

2.19 Mapping Bitmap 48

3.1 Dreidimensionale fraktale Bildmengenkompression 53

3.2 Fraktale Kompression mit Kreuzverweisen 54

3.3 Verschränkt rekursive Rückentwicklung von zwei fraktal miteinander komprimierten Bildern 55

3.4 Aufteilung eines Bildes mittels des Quadtree-Verfahrens 56

3.5 Kompressionsgewinn bei fraktaler Kompression 58

3.6 Fraktale Kreuzverweise 60

3.7 Fraktale Ähnlichkeits-Metrik 62

3.8 Fraktale Kreuzverweise bei identischen Bildern 62

3.9 Approximation fraktaler Distanzen durch Herunterskalierung . 63

3.10 Vergleich zwischen fraktaler Kompression und Videokompression 65

3.11 Auflösungstufen von Bildern 66

3.12 Ähnliche Bildpaare 68

3.13 KLT Basisbilder von 4000 Fotos 71

3.14 KLT Basisbilder von 4000 Textseiten 71

3.15 Abschätzung der KLT Kompressionsrate durch die Kovarianzmatrix 74

3.16 Arbeitsweise des Mehrbild PNG Algorithmus 77

3.17 Beispiel für den Mehrbild PNG Algorithmus 78

3.18 PNG-Komprimierbarkeit in Abhängigkeit zur Bildähnlichkeit . 81

3.19 Herunterskalierung von Bildern zur Abschätzung der PNG Komprimierbarkeit 81

3.20 Korrelation zwischen PNG-Kompressioneffizienzen 82

3.21 Perspektivische Einflüsse bei der Texturzerlegung 84
3.22 Zwei Bilder, mit einer Reihe ähnlicher Texturen 84
3.23 Synthese von Texturen 86
3.24 Extraktion von Texturen 87
3.25 Zuweisung von Pixeln zu Texturen 90
3.26 Beispiel einer Texturbasierten Kompression 92
3.27 Auftretenshäufigkeit von JPEG-Koeffizienten in Texturen . . . 94
4.1 Die b^+-Statistik auf Graphen und Hypergraphen 100
4.2 RGB-Farbwürfelbelegung eines Bildes 103
4.3 RGB-Farbwürfelbelegung eines anderen Bildes 104
4.4 Farbhistogramme von zwei Bildern 105
4.5 Ein „Difference of Gaussians“ Kantenfilter 106
4.6 Approximation der NCD durch Bildähnlchkeit 109
4.7 Der MST Algorithmus 112
4.8 Entartetes MST Clustering 112
4.9 Der SAHN Algorithmus 113
4.10 Der Greedy Clustering-Algorithmus 115
4.11 Der nCut Clustering-Algorithmus 116
4.12 Der k-Means Clustering-Algorithmus 121
4.13 Der MeanShift Clustering-Algorithmus 122
4.14 Neukombination von Gruppen im genetischen Algorithmus . . 125
4.15 maximaler Kompressionsgewinn durch maximales Matching . 127
4.16 Textblöcke sowie b^+ Distanzrelationen 131
4.17 Optimales Clustering von 4.16 132

4.18 Vergleich verschiedener Clusteralgorithmen 134

5.1 Die wichtigsten Komponenten der mediaTUM-Architektur . . 137

5.2 Screenshot der Einstiegsseite von mediaTUM 139

5.3 Dateiformate und Speicherplatzverbrauch in mediaTUM . . . 140

5.4 Dateiformate und Speicherplatzverbrauch in mediaTUM mit der neuen Kompression 140

5.5 Typische Bildmengen von ähnlichen Bildern in mediaTUM . . 141

Index

Ähnlichkeitsmetrik, 97
    Approximation, 108
    fraktale, 61
    KLT, 73

Archivierung, 78
Arithmetische Kodierung, 59
Auflösung, 64

b $^{+}$, 99
B-Frame, 65
Bildähnlichkeitsanalyse, 100
Blocking, 25

CCITT, 16
Cluster, 96
Clustering, 110
    Qualität, 127

DAG, directed acyclic graph, 136
Daubechies 9/7, 25
DCT, 22

EXIF, 138

Farbhistogramme, 101
Featureextraktion, 101
Fensterfouriertransformation, 88
FFT, 20
FIF, 37
Filter, 21
Filterhistogramme, 105
Fraktale Bildmengenkompression, 52
    im Dreidimensionalen, 52
    mittels Kreuzverweisen, 53
Fraktale Kompression, 36
fts3, 136

Gabor, 21
Gabor-Filter, 88
Genetic, 125
Genetic2, 125
Genetische Algorithmen, 123
GIF, 12
Graph, 98
Greedy, 78, 114

Helligkeit, 58
Hilbert, 41
Huffman, D.A., 11
Hypergraph, 99
Hyperspektrale Kompression, 51

I-Block,-Frame, 27
I-Frame, 65
ICT, 25
IFS, Iterated Function Systems, 36
Integral Image, 89
iptc, 138

JFIF, 24
JPEG, 51
    Koeffizienten, 91
    Qualitätsparameter, 139
    verlustbehaftet, 24
    verlustlos, 17
JPEG2000, 51
    verlustbehaftet, 25
    verlustlos, 18

k-Means, 119
Kacheln, 86
Kantendichte, 102
Kantenrichtung, 102
KLT, 29
    auf Bildmengen, 69
Kompressionsoptimierung
    durch Bildgruppensuche, 96
Kontrast, 58
Kovarianzmatrix, 69
Kreuz-Korrelation, 85
Kreuzkorrelation, 41
Kurtosis, 41, 85

Legall 5/3, 25
Lempel&Liv, 11
LOCO-I, 17
LZ77, 11
LZ78, 11
LZW, 11

magpy, 136
Mapping Bitmap, 47
Maximales Matching, 126
MeanShift, 120
mediaTUM, 135
MEncoder, 72
MNG, 14
Modellbasierte Kompression, 45
Motion Block, 72
Motion Estimation, 26
MST, 110
Multidimensional Scaling, 117

Nachkompression, 91
NCD, 98
nCut, 114
Nebendiagonale, 73

Open Access, 135
optimale Transformation, 69

P-Block,-Frame, 27
P-Frame, 65
PCA, Principal Components Analysis, 29
Perfektes Clustering, 125
PIFS, Partitioned Iterated Function Systems, 36
PNG, 12
    auf Bildmengen, 76
Polygonalisierung, 42
Power Spectrum, 85
Prediktoren, 12

Quad-Tree, 57

RCT, 25
RGB-Farbwürfel, 102
Ringing, 25
RLE, 16, 25

Sättigung, 70
SAHN, 111
SAHNb, 113
Scanlines, 80
Schräge, 41, 85
Segmentierung, 83
    Bereichsverfahren, 47
    Iterative Verfahren, 47
    Kantenbasierte Methoden, 47
    nach Farben, 42
    nach Textur, 88
    nach Texturen, 45
Selbstreferenz, 59
Shannon, Claude, 10
Singular Value Decomposition, 29
sqlite, 136
Steerable, 30
SVD, 29

Textons, 30
Texturanalyse, 39, 85
Texturdatenbank, 83
Texturkompression, 39

Textursynthese, 39, 83
TIFF, 16
Tiling, 26
Token-Sets, 129

Verlustbehaftete Kompression, 19
Verlustlose Kompression, 10
Videokompression, 26, 64
Videoplatform, 135

Wavelets, 21

X-Skalierung, 80
xpdf, 136

YUV, 54

Zerotrees, 26
zlib, 130

Literaturverzeichnis

[1] M.D. Adams and F. Kossentini. Reversible Integer-to-Integer Wavelet Transforms for Image Compression: Performance Evaluation and Analysis. *IEEE Transactions on Image Processing*, 9(6), 2000.

[2] H. Al-Muhairi, M. Fleury, and A. F. Clark. Computationally Efficient Quantitative Testing of Image Segmentation with a Genetic Algorithm. In *SITIS 2007*.

[3] C.J. Alpert and A.B. Kahng. Recent directions in netlist partitioning: a survey. *Integration, the VLSI Journal*, 19(1-2):1–81, 1995.

[4] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies. Image coding using wavelet transform. *Image Processing, IEEE Transactions on*, 1(2):205–220, 1992.

[5] T. Asano, B. Bhattacharya, M. Keil, and F. Yao. Clustering algorithms based on minimum and maximum spanning trees. In *SCG '88: Proceedings of the fourth annual symposium on Computational geometry*, pages 252–257, New York, NY, USA, 1988. ACM Press.

[6] T. Asano, D.Z. Chen, N. Katoh, and T. Tokuyama. Polynomial-time solutions to image segmentation. pages 104–113, January 1996.

[7] V. Atalay, E. Gelenbe, and N. Yalabik. The random neural network model for texture generation. *International Journal of Pattern Recognition and Artificial Intelligence*, 6(1):131–141, 1992.

[8] M. Barnsley and H. Rising. *Fractals Everywhere*. Academic Press, 1993.

[9] KU Barthel and T. Voye. Three-dimensional fractal video coding. *Image Processing, 1995. Proceedings., International Conference on*, 3, 1995.

[10] S. Battiato, G. Barbera, G. di Blasi, G. Gallo, and G. Messina. Advanced SVG triangulation/polygonalization of digital images. *Proc. SPIE*, 5670(1), 2005.

[11] Mary Bellis. The history of the telegraph and telegraphy. *About Inventors*, 2007.

[12] Jeremy S. De Bonet and Paul Viola. A non-parametric multi-scale statistical model for natural images. In *NIPS '97: Proceedings of the 1997 conference on Advances in neural information processing systems 10*, pages 773–779, Cambridge, MA, USA, 1998. MIT Press.

[13] I. Borg and P.J.F. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, 1997.

[14] C. Bouman and B. Liu. Multiple resolution segmentation of textured images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(2):99–113, 1991.

[15] R. Chellappa and S. Chatterjee. Classification of textures using gaussian markov random fields. *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on*, 33:959–963, 1985.

[16] J. Chen, TN Pappas, A. Mojsilovic, and B. Rogowitz. Adaptive image segmentation based on color and texture. *Image Processing. 2002. Proceedings. 2002 International Conference on*, 3, 2002.

[17] Charilaos Christopoulos, Athanassios Skodras, and Touradj Ebrahimi. The jpeg 2000 still image coding system: An overview. *IEEE Transaction on Consumer Electronics*, November 2000.

[18] R. Cilibrasi and PMB Vitani. Clustering by Compression. *Information Theory, IEEE Transactions on*, 51(4):1523–1545, 2005.

[19] A. Civril, M. Magdon-Ismail, and E. Bocek-Rivele. Graph Drawing using Spectral Distance Embedding. *Proc. Graph Drawing*, pages 512–513, 2005.

[20] D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5):603–619, 2002.

[21] R. De Maesschalck, D. Jouan-Rimbaud, and DL Massart. The Mahalanobis distance. *Chemometrics and Intelligent Laboratory Systems*, 50(1):1–18, 2000.

[22] Peter Deutsch. Deflate compressed data format specification. RFC 1951, 1996.

[23] P. Duchet. *Handbook of Combinatorics*. Elsevier (North-Holland), 1995.

[24] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 341–346, New York, NY, USA, 2001. ACM Press.

[25] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. *iccv*, 02:1033, 1999.

[26] Y. Fisher et al. *Fractal image compression*. Springer, 1995.

[27] Y. Fisher, D. Rogovin, and TP Shen. Fractal (self-VQ) encoding of video sequences. *Visual Communications and Image Processing*, pages 1359–1370, 1994.

[28] TC Folsom, RB Pinter, Q.I. Inc, and WA Kent. Primitive features by steering, quadrature, and scale. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(11):1161–1173, 1998.

[29] WT Freeman and EH Adelson. The design and use of steerable filters. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 13(9):891–906, 1991.

[30] K. Fukunaga, S. Yamada, T. Kasai, and H. Stone. A representation of hypergraphs in the Euclidean space(such as airlines network). *IEEE Transactions on Computers*, 33:364–367, 1984.

[31] M. Ghazel, GH Freeman, and ER Vrscay. Fractal image denoising. *IEEE Transactions on Image Processing*, 12(12):1560–1578, 2003.

[32] L. Ghouti, A. Bouridane, and MK Ibrahim. Image compression using texture modeling. *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 2313–2316, 2005.

[33] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley Publishing Company, Inc., 1989.

[34] JC Gower and P. Legendre. Metric and Euclidean properties of dissimilarity coefficients. *Journal of Classification*, 3(1):5–48, 1986.

[35] L. Grady. Random walks for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(11):1, 2006.

[36] U. Grenander and G. Szego. *Toepliz forms and their applications*. University of California Press Berkeley, 1958.

[37] D. Harel and Y. Koren. Clustering spatial data using random walks. *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 281–286, 2001.

[38] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 229–238, New York, NY, USA, 1995. ACM Press.

[39] K. Hindistan. Video Playback and Encoding with MPlayer and MEncoder, 2003.

[40] DA Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[41] A. Jain, M. Murty, and P. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31, 1999.

[42] AK Jain. Image data compression: A review. *Proceedings of the IEEE*, 69(3):349–389, 1981.

[43] M. Johnson, GJ Brostow, J. Shotton, O. Arandjelovic, V. Kwatra, and R. Cipolla. Semantic Photo Synthesis. *Computer Graphics Forum*, 25(3):407–413, 2006.

[44] B. Julesz. Visual Pattern Discrimination. *Information Theory, IEEE Transactions on*, 8(2):84–92, 1962.

[45] B. Julesz. Experiments in the visual perception of texture. *Sci Am*, 232(4):34–43, 1975.

[46] B. Julesz. Textons, the elements of texture perception, and their interactions. *Nature*, 290(5802):91–97, 1981.

[47] J. Jung, M. Antonini, and M. Barlaud. Optimal JPEG decoding. *Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*, 1, 1998.

[48] AB Kahng. Fast hypergraph partition. *Proceedings of the 1989 26th ACM/IEEE conference on Design automation conference*, pages 762–766, 1989.

[49] LM Kaplan and C.C.J. Kuo. Texture roughness analysis and synthesis via extended self-similar (ESS) model. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(11):1043–1056, 1995.

[50] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(1):69–79, 1999.

[51] HP Karypis. Hypergraph based clustering in high-dimensional data sets: a summary of results. *IEEE Bulletin of Technical Committee on Data Engineering*, 21(1):105–140, 1998.

[52] J.J. Koenderink and A.J. van Doorn. Representation of local geometry in the visual system. *Biological Cybernetics*, 55(6):367–375, 1987.

[53] M. Kramm. Image Cluster Compression using Partitioned Iterated Function Systems and efficient Inter-Image Similarity Features. In *SITIS 2007*.

[54] M. Kramm. Compression of image clusters using Karhunen Loeve transformations. In *Electronic Imaging, Human Vision*, volume XII, pages 101–106, 2007.

[55] M. Kramm. Image Cluster Compression using Texture Databases. In *Electronic Imaging, Human Vision*, volume XIII, 2008.

[56] R. Krovi. Genetic algorithms for clustering: a preliminary investigation. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, 4, 1992.

[57] LI Kuncheva and JC Bezdek. Nearest prototype classification: clustering, genetic algorithms, or random search? *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, 28(1):160–164, 1998.

[58] V. Kwatra, A. Schoedl, I.A. Essa, G. Turk, and A.F. Bobick. Graphcut textures: image and video synthesis using graph cuts. *ACM Transactions on Graphics*, 22(3):277–286, 2003.

[59] W.M. Lawton. Tight frames of compactly supported affine wavelets. *Journal of Mathematical Physics*, 31:1898, 1990.

[60] MS Lazar and LT Bruton. Fractal block coding of digital video. *Circuits and Systems for Video Technology, IEEE Transactions on*, 4(3):297–308, 1994.

[61] HS Lee, NH Younan, and RL King. Hyperspectral image cube compression combining JPEG-2000 and spectral decorrelation. *Geoscience and Remote Sensing Symposium, 2002. IGARSS'02. 2002 IEEE International*, 6, 2002.

[62] A. Levy and M. Lindenbaum. Sequential karhunen-loeve basis extraction and its application to images. volume 2, pages 456–460 vol.2, 1998.

[63] W.Y. Ma and BS Manjunath. EdgeFlow: a technique for boundary detection and image segmentation. *Image Processing, IEEE Transactions on*, 9(8):1375–1388, 2000.

[64] J. Malik, S. Belongie, T. Leung, and J. Shi. Contour and Texture Analysis for Image Segmentation. *International Journal of Computer Vision*, 43(1):7–27, 2001.

[65] B.B. Mandelbrot and J.A. Wheeler. The Fractal Geometry of Nature. *American Journal of Physics*, 51:286, 1983.

[66] B. Manjunath, J. Ohm, V. Vasudevan, and A. Yamada. Color and texture descriptors. *Circuits and Systems for Video Technology, IEEE Transactions on*, 11:703–715, 2001.

[67] TM Martinetz, SG Berkovich, and KJ Schulten. Neural-gas' network for vector quantization and its application totime-series prediction. *Neural Networks, IEEE Transactions on*, 4(4):558–569, 1993.

[68] U. Maulik and S. Bandyopadhyay. Genetic algorithm-based clustering technique. *Pattern Recognition*, 33(9):1455–1465, 2000.

[69] H. Müller, N. Michoux, D. Bandon, and A. Geissbuhler. A review of content-based image retrieval systems in medical applications — clinical benefits and future directions. *International Journal of Medical Informatics*, 73(1):1–23, 2004.

[70] H. Muller, D.M. Squire, W. Muller, and T. Pun. Efficient access methods for content-based image retrieval with inverted files. *Multimedia Storage and Archiving Systems IV (VV02)*, 3846:20–22.

[71] TN Pappas. An adaptive clustering algorithm for image segmentation. *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, 40(4):901–914, 1992.

[72] W.B. Pennebaker and J.L. Mitchell. *JPEG: Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

[73] Javier Portilla and Eero P. Simoncelli. A parametric texture model based on joint statistics of complex wavelet coefficients. *Int. J. Comput. Vision*, 40(1):49–70, 2000.

[74] J. Puzicha, T. Hofmann, and J. Buhmann. Non-parametric similarity measures for unsupervised texture segmentation and image retrieval. *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pages 267–272, 1997.

[75] E. Rasmussen. Clustering algorithms. *Information retrieval: data structures and algorithms table of contents*, pages 419–442, 1992.

[76] G. Roelofs. PNG: The Definitive Guide. 2003.

[77] J.G. Rosiles and MJT Smith. Compression of SAR and ultrasound imagery using texture models. *Data Compression Conference, 1999. Proceedings. DCC'99*, 1999.

[78] J. Rossignac. Edgebreaker: connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[79] Y. Rui, T. Huang, and S. Chang. Image retrieval: current techniques, promising directions and open issues, April 1999.

[80] TW Ryan, LD Sanders, HD Fisher, and AE Iverson. Image compression by texture modeling in the wavelet domain. *Image Processing, IEEE Transactions on*, 5(1):26–36, 1996.

[81] H.A.B. Saip and C.L. Lucchesi. Matching Algorithms for Bipartite Graphs. *Relatorio Tecnico*, 700(03), 1993.

[82] David Salomon. *Data compression, the complete reference*. 2000.

[83] Arne Seifert. *Strategien der Langzeitarchivierung am Beispiel eines Medienservers*. TU-München, 2009. Dissertation, geplante Einreichung: Anfang 2009.

[84] M. J. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, 41:3445–3462, 1993.

[85] L. Shen and RM Rangayyan. A segmentation-based lossless image coding method for high-resolution medical image compression. *Medical Imaging, IEEE Transactions on*, 16(3):301–307, 1997.

[86] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[87] J. Shotton, J. Winn, C. Rother, and A. Criminisi. Textonboost: Joint appearance, shape and context modeling for multi-class object recognition and segmentation. *Europ. Conf. Comp. Vision*, 2006.

[88] T. Sikora. MPEG Digital Video Coding Standards. *Compressed Video Over Networks*, 2000.

[89] E. Simoncelli and W. Freeman. The steerable pyramid: a flexible architecture for multi-scale derivative computation. volume 3, pages 444–447 vol.3, 1995.

[90] D.M.G. Squire, W. Müller, H. Müller, and T. Pun. Content-based query of image databases: inspirations from text retrieval. *Pattern Recognition Letters*, 21(13-14):1193–1198, 2000.

[91] M.J. Swain and D.H. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

[92] W. Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Appl. Comput. Harmon. Anal*, 3(2):186–200, 1996.

[93] T. Tan and H. Yan. Face recognition by fractal transformations. *Acoustics, Speech, and Signal Processing, 1999. ICASSP'99. Proceedings., 1999 IEEE International Conference on*, 6, 1999.

[94] J. Tesic. Metadata practices for consumer photos. *Multimedia, IEEE*, 12(3):86–92, 2005.

[95] M. Tkalcic and JF Tasic. Colour spaces: perceptual, historical and applicational background. *EUROCON 2003. Computer as a Tool. The IEEE Region 8*, 1, 2003.

[96] OJ Tobias and R. Seara. Image segmentation by histogram thresholding using fuzzy sets. *Image Processing, IEEE Transactions on*, 11(12):1457–1465, 2002.

[97] N. Tran. The normalized compression distance and image distinguishability. *Proceedings of SPIE*, 6492:64921D, 2007.

[98] Greg Turk. Texture synthesis on surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 347–354, New York, NY, USA, 2001. ACM Press.

[99] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *Proc. CVPR*, 1:511–518, 2001.

[100] Michael Wakin, Justin Romberg, Hyeokho Choi, and Richard Baraniuk. Image compression using an efficient edge cartoon + texture model. *Proceedings of the DCC*, 00:0043, 2002.

[101] G.K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

[102] L.Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 479–488, 2000.

[103] MJ Weinberger, G. Seroussi, and G. Sapiro. The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS. *Image Processing, IEEE Transactions on*, 9(8):1309–1324, 2000.

[104] T. Welch. A Technique for High-Performance Data Compression. *Computer*, 17(6):8–19, 1984.

[105] M. Wertheimer. *Gestalt Theory.* s. n, 1944.

[106] T. Wiegand, GJ Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H. 264/AVC video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, 2003.

[107] I.H. Witten, T.C. Bell, and A. Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 1999.

[108] Y. Xu, V. Olman, and D. Xu. Minimum Spanning Trees for Gene Expression Data Clustering. *Genome Informatics*, 12:24–33, 2001.

[109] J. Zhang and JW Modestino. A model-fitting approach to cluster validation with application to stochastic model-based image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):1009–1017, 1990.

[110] D. Zhou, J. Huang, and B. Scholkopf. Beyond pairwise classification and clustering using hypergraphs. Technical report, Max Planck Institute for Biol. Cybernetics, TR-143, August 2005.

[111] S.C. Zhu, C. Guo, Y. Wang, and Z. Xu. What are Textons? *International Journal of Computer Vision*, 62(1):121–143, 2005.

[112] J. Ziv and A. Lempel. An universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, 1977.

[113] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.