Fraktale Bildkompression: Adaptive Partitionierungen und Komplexität

Matthias Ruhl

Diplomarbeit

Mathematische Fakultät
Albert-Ludwigs-Universität Freiburg
Prof. Dr. Dietmar Saupe

April 1997

Inhaltsverzeichnis

Vorwort

In dieser Arbeit behandele ich ganz unterschiedliche Aspekte einer besonderen Methode zur Bildkompression – der fraktalen Bildkompression. Diese Arbeit besteht aus drei Teilen: einer kurzen Einführung in die Grundlagen der fraktalen Kompression, einem praktischen Teil und einem theoretischen Teil. Der praktische und der theoretische Teil sind dabei völlig unabhängig voneinander.

Im praktischen Teil dokumentiere ich ein Programm, das im Rahmen dieser Diplomarbeit geschrieben wurde. Es zeigt, daß fraktale Kompression in der Praxis gut funktioniert: viele schöne (teilweise auch bunte) Bilder finden sich im Anhang.

Aber da dies eine Mathematik-Diplomarbeit ist, folgt sogleich ein theoretischer Teil. Hier wird fraktale Kompression aus einem eher ungewohnten Blickwinkel betrachtet.

Zuerst wird dort fraktale Kompression aus komplexitätstheoretischer Sicht untersucht. Das wichtigste Resultat ist, daß das Finden des optimalen fraktalen Codes für ein Bild **NP**-hart ist.

Im zweiten, einfacheren, Kapitel des theoretischen Teils geht es dann um die Ausdrucksfähigkeit von fraktalen Codes für kontinuierliche Träger.

Ich danke allen, die beim Entstehen dieser Arbeit geholfen haben. Insbesondere danke ich Dietmar Saupe für die Ermöglichung und Betreuung dieser Arbeit, Raouf Hamzaoui für zahlreiche Anmerkungen und Verbesserungsvorschläge und last, but not least, Hannes Hartenstein. Er hatte die Idee, die Komplexität von fraktaler Kompression zu untersuchen, und hat durch zahlreiche Diskussionen die Darstellung des theoretischen Teils beeinflußt (und damit verbessert :).

Freiburg, im April 1997 Matthias Ruhl

Erklärung

Ich bestätige hiermit, daß ich diese Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Freiburg, 15. April 1997

Teil I

Grundlagen

Kapitel 1

Einleitung

1.1 Fraktale Bildkompression

Wir sind von Bildern umgeben. In Zeitungen und Zeitschriften, in der Werbung und in Diplomarbeiten, überall vermitteln sie Informationen, für die Worte nicht ausreichen.

Wir sind von Computern umgeben. Immer mehr Tätigkeiten werden heute mit Computern ausgeführt. Es war also nur eine Frage der Zeit, bis auch Bilder mit und von Computern verarbeitet wurden. So kann man mit Computern Fotos nachbearbeiten, Trickfilme erzeugen oder Texte bebildern. Auch im weltweiten Computernetzwerk Internet finden Bilder immer mehr Anwendungen. Fast alle Seiten des World Wide Webs sind mit Bildern 'verschönert'.

Bei der Verarbeitung von Bildern auf Computern stößt man immer wieder auf das gleiche Problem. Selbst kleine Bilder brauchen sehr viel Speicherplatz. Das führt dazu, daß Festplatten vollaufen und Übertragungszeiten sehr lang werden. Aus diesem Grund beschäftigt man sich schon lange mit Methoden zur Bildkompression.

Programme zur Bildkompression versuchen, die in einem Bild enthaltene Information möglichst *kurz* durch einen Code auszudrücken. Dieser Code kann dann anstelle des ursprünglichen Bildes platzsparend gespeichert oder zeitgünstig übermittelt werden. Es gibt eine Vielzahl von Bildkompressionsverfahren. Sie haben Namen wie GIF, JPEG, TIFF, und sind so weit verbreitet, daß man heutzutage fast überhaupt keine unkomprimierten Bilder mehr findet.

Eine interessante Eigenschaft der genannten Kompressionsverfahren ist es, daß sie die Bilder *verlustbehaftet* kodieren. Das heißt, daß sich aus dem Code nicht mehr das Originalbild, sondern nur ein dem Originalbild ähnliches Bild zurückerhalten läßt. Das ist jedoch nicht weiter schlimm, da kleine Fehler den visuellen Eindruck meist nicht stören.

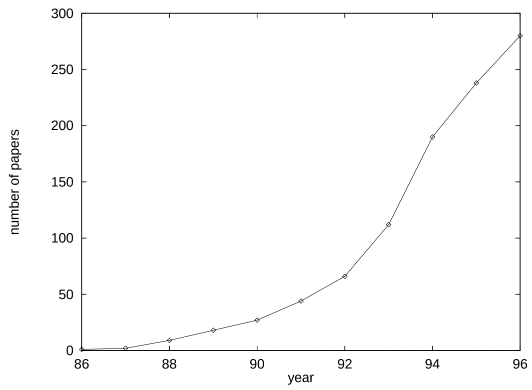


Abbildung 1.1: Anzahl der Veröffentlichungen zur fraktalen Bildkompression

1987 erfand Michael Barnsley, ein Professor am Georgia Institute of Technology in Atlanta, die fraktale Bildkompression. In mehreren Artikeln [BaSl87, BaSl88] beschrieb er die Vorzüge der neuen Methode, unter anderem sollte das Verfahren Bilder auf ein Zehntausendstel ihrer Originalgröße komprimieren können. Angesichts bis dahin üblicher Kompressionsraten von vielleicht 50:1 war das beachtlich.

Barnsleys Artikel waren jedoch mit Details zur Funktionsweise seines Algorithmus sehr zurückhaltend. So wurde es 1989, bis die Fachwelt durch die Dissertation von Arnaud Jaquin [Jaquin89], einem Doktoranden von Barnsley, erfuhr, wie fraktale Kompression wirklich funktionierte. Es stellte sich schnell heraus, daß Kompressionsraten von 10000:1 nur unter sehr speziellen Umständen mit eigens dafür konstruierten Bildern erreicht werden konnten. Trotzdem war und ist fraktale Kompression eine ungewöhnliche, aber praktikable Methode, um Bilder zu komprimieren.

In den folgenden Jahren wurde eine stetig wachsende Anzahl von Veröffentlichungen auf dem Gebiet der fraktale Kompression gemacht (siehe Abbildung 1.1). Heute ist die fraktale Kompression in der Bildverarbeitung allgemein bekannt, allerdings wird sie noch nicht sehr häufig angewandt. Ob sich dies noch ändert, und wie sie sich gegenüber anderen vielversprechenden Kompressionsverfahren wie z.B. Wavelets [Shapiro93, SaPe96] hält, wird sich in den nächsten Jahren zeigen.

1.2 Diese Arbeit

Diese Arbeit behandelt ganz unterschiedliche Aspekte der *fraktalen Bildkompression*. Der erste Teil der Arbeit dient der Einführung in das Gebiet. In Kapitel 2 wird erklärt, was fraktale Kompression ist, die mathematischen Grundlagen dargestellt und die Notation festgelegt.

Der zweite Teil beschäftigt sich mit der Anwendung der fraktalen Bildkompression in der Praxis. In Kapitel 3 wird ein Programm vorgestellt, das im Rahmen dieser Arbeit entwickelt wurde. Zunächst werden die Designentscheidungen dargelegt und motiviert. Anschließend wird der zugrundeliegende Algorithmus beschrieben und mit anderen Verfahren verglichen. Das Kapitel endet mit der Darstellung der Ergebnisse, die mit dem Programm erzielt werden konnten. Diese sind vergleichbar mit den besten bereits existierenden Verfahren, unser Programm ist jedoch deutlich schneller.

Der abschließende dritte Teil ist den theoretischen Grundlagen der fraktalen Kompression gewidmet, die in der Literatur bislang noch gar nicht oder nur ansatzweise untersucht wurden. Zuerst wird in Kapitel 4 das Problem, den fraktalen Code mit dem geringsten Qualitätsverlust zu einem Bild zu finden, aus komplexitätstheoretischer Sicht betrachtet. Es zeigt sich, daß das Problem **NP**-hart ist. Des weiteren wird bewiesen, daß der Standardalgorithmus zum Finden fraktaler Codes, Collage Coding, kein Approximationsalgorithmus für dieses Problem ist.

In Kapitel 5 steht dann die Ausdrucksfähigkeit von fraktalen Codes im Mittelpunkt. Wir zeigen, daß sich für kontinuierliche Träger sogar Bilder beschreiben lassen, die in einem gewissen Sinn nicht mehr berechenbar sind.

Im Anhang finden sind 'manual pages', die die Bedienung des Programms aus Kapitel 3 erklären, sowie einige Bilder, die mit dem Programm komprimiert wurden.

Diese Arbeit ist so gestaltet, daß die Kapitel des zweiten und dritten Teils unabhängig voneinander gelesen werden können. Da der erste Teil lediglich die Grundlagen bereitstellt, reicht es, bei entsprechender Vertrautheit mit dem Thema, ihn nur zu überfliegen.

Kapitel 2

Fraktale Bildkompression

In diesem Kapitel wird die Funktionsweise der fraktalen Kompression erläutert, und die dazu notwendigen mathematischen Definitionen bereitgestellt. Für ausführlichere Darstellungen siehe z.B. [Fisher94a, SaHaHa96].

2.1 Bilder

Was sind Bilder aus mathematischer Sicht? Bilder sind Abbildungen zwischen einem Träger und einer Menge von Farben.

Definition 2.1 (Bild)
Ein Bild *ist ein Tripel* $(T,(F,d),b)$, *wobei* T *eine beliebige Menge (die* Trägermenge*),* (F,d) *ein kompakter metrischer Raum (die Menge der* Farben*) und* b *eine Funktion von* T *nach* F *ist. Falls* T *und* F *aus dem Zusammenhang klar sind, sprechen wir oft auch nur von einem Bild* b*. Gelegentlich nennen wir Bilder auch* Signale. □

Definition 2.2 ($\mathcal{B}(T,F)$)
Sei T *eine Trägermenge und* (F,d) *ein Farbraum. Dann ist*

$$\mathcal{B}(T,F) := \{b \mid b : T \to F\}$$

die Menge der Bilder über T *(mit Farbraum* F*). Falls* T *und* F *klar (oder unwichtig) sind, schreiben wir auch kurz* $\mathcal{B}$. □

Beispiel 2.3
In dieser Arbeit kommen drei verschiedene Arten von **Trägern** T vor:

Diskret, 2-dimensional: $T = \{1,\ldots,m\} \times \{1,\ldots,n\} \quad (m,n \in \mathbb{N})$

Diskret, 1-dimensional: $T = \{1,\ldots,n\} \quad (n \in \mathbb{N})$

Kontinuierlich, 2-dimensional: $T = [0,x[\times[0,y[\quad (x,y \in \mathbb{R}^+)$

Wir werden die weiteren Definitionen deshalb auch gezielt auf diese Anwendung hin fassen. Allerdings sollte klar sein, wie man fraktale Kompression gegebenenfalls auch auf andere Signalarten (z.B. höher dimensionale Signale) anwenden kann.

Auch bei den **Farbmengen** (F,d) werden wir nur einige spezielle Fälle brauchen. Nämlich die letzten drei der folgenden Beispiele.

Schwarz-Weiß-Bilder: $F = \{0,1\}$. 0 entspricht schwarz, 1 entspricht weiß.

Graustufenbilder, diskret: $F = \{0,1,\ldots,255\}$. 0 entspricht schwarz, 255 bedeutet weiß, die restlichen Werte sind die Graustufen dazwischen.

Graustufenbilder, kontinuierlich: $F = [0,1]$. 0 entspricht schwarz, 1 bedeutet weiß.

Farbbilder, diskret: $F = \{0,1,\ldots,255\}^3$. Die einzelnen Komponenten geben den Rot-, Grün- und Blauanteil der Farbe an.

Als Metrik d wählen wir in den ersten drei Fällen jeweils die absolute Differenz der Werte. Bei den Farbbildern kann man z.B. den euklidischen Abstand der Punkte im $\mathbb{R}^3$ nehmen. □

Die Metrik des Farbraumes brauchen wir, um Bilder miteinander vergleichen zu können. Eine Möglichkeit, dies zu tun, ist die folgende.

Definition 2.4 (L_∞-Metrik)
Seien b_1, b_2 zwei Bilder mit Träger T und Farbraum (F,d). Dann setzen wir:

$$L_\infty(b_1,b_2) := \sup_{x\in T} d(b_1(x), b_2(x)). \quad \Box$$

Die L_∞- oder auch *Supremumsmetrik* legt also den Abstand zweier Bilder als den größten Unterschied an einer Stelle fest. Wir werden später noch eine weitere, für den Vergleich realer Bilder geeignetere Metrik kennenlernen. Doch die L_∞-Metrik reicht uns momentan aus, um die Funktionsweise der fraktalen Kompression einzuführen. Davor noch folgender Satz.

Satz 2.5
Seien T ein Träger und (F,d) ein Farbraum. Dann ist $(\mathcal{B}(T,F), L_\infty)$ ein vollständiger metrischer Raum.

Beweis: Das folgt direkt daraus, daß (F,d) nach Voraussetzung ein vollständiger metrischer Raum ist. ■

2.2 Abbildungen

Die Grundlage der fraktalen Kompression sind *kontraktive* Abbildungen zwischen Bildern.

Definition 2.6 (Kontraktive Abbildung)
Eine Abbildung $f : \mathcal{B} \to \mathcal{B}$ heißt kontraktiv *bzgl. einer Metrik L auf $\mathcal{B}$, wenn es ein $\delta < 1$ gibt, so daß $\forall\, b_1, b_2 \in \mathcal{B} : L(f(b_1), f(b_2)) \leq \delta \cdot L(b_1, b_2)$ gilt. δ heißt* Kontraktionsfaktor *von f.* □

Satz 2.7 (Fixpunktsatz)
Sei $f : \mathcal{B} \to \mathcal{B}$ kontraktiv bzgl. L_∞. Dann besitzt f einen eindeutig bestimmten Fixpunkt $\Omega_f \in \mathcal{B}$ mit $f(\Omega_f) = \Omega_f$. Außerdem konvergiert für jedes $b \in \mathcal{B}$ die Folge $b, f(b), f(f(b)), f(f(f(b))), \ldots$ gegen Ω_f.

Beweis: Anwendung des Banachschen Fixpunktsatzes mit Satz 2.5. ■

Definition 2.8 (Ω_f)
Der Fixpunkt einer kontraktiven Abbildung f heißt Ω_f. □

Die Idee der fraktalen Kompression ist folgende. Wir kodieren ein Bild $b \in \mathcal{B}$ durch eine kontraktive Funktion f auf $\mathcal{B}$, deren Fixpunkt möglichst nahe bei b liegt. Und Satz 2.7 zeigt, wie wir den Fixpunkt Ω_f zu einer kontraktiven Abbildung f schnell bestimmen können: durch iteriertes Anwenden der Funktion auf ein beliebiges Ausgangsbild.

Wir beschränken uns dabei auf spezielle, *stückweise affine Funktionen f*. Denn diese haben den Vorteil, daß man sie sehr effizient abspeichern kann – im Gegensatz zu allgemeinen kontraktiven Funktionen.

Fraktale Kompression funktioniert nach folgendem Schema:

1. Zuerst wird der Träger T in disjunkte Teilmengen $T = R_1 \dot\cup R_2 \dot\cup \ldots \dot\cup R_k$ zerlegt. Die R_i heißen **Ranges**.[1] Beispiele für die Aufteilung eines zweidimensionalen Trägers in Ranges sieht man in Abbildung 3.1 auf Seite 19.

2. Jeder Range R_i wird eine Menge $D_i \subset T$, genannt **Domain**, zugeordnet.[2] Diese wird mittels einer kontraktiven Abbildung auf die Range abgebildet (siehe Abbildung 2.1). Diese kontraktive Abbildung hat dabei zwei Teile:

[1] In der Praxis sind Ranges meist *Intervalle*, d.h. von der Form $\{x_1, \ldots, x_2\} \times \{y_1, \ldots, y_2\}$ bzw. $\{a, \ldots, b\}$ bzw. $[x_1, x_2[\times [y_1, y_2[$, oder aber *endliche, zusammenhängende Vereinigungen* von Intervallen.

[2] Im diskreten Fall sind die Domains meist doppelt so groß wie die Range, und von der gleichen Form (siehe Abbildung 2.1).

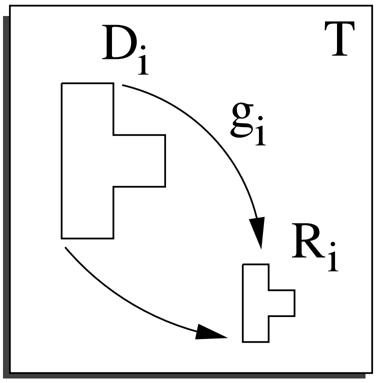


Abbildung 2.1: Auf jede Range R_i wird kontraktiv eine Domain D_i abgebildet.

Geometrisch: Zunächst wird die Domain mittels einer *geometrischen Abbildung* auf die Größe der Domain gebracht, wie in Abbildung 2.1 zu sehen. Diese surjektive (!) Abbildung von D_i nach R_i nennen wir g_i. [3]

Farbwerte: Die Farbwerte der so auf Rangegröße gebrachten Domain werden dann mittels einer **kontraktiven affin linearen Abbildung** $x \mapsto s_i \cdot x + o_i$ auf die Farbwerte der Range abgebildet. Die Parameter hierbei sind der **Skalierungsfaktor** s_i, der in $]-1,1[$ liegen muß, und der **Offsetparameter** o_i.

3. Diese Parameter zusammen liefern uns eine **kontraktive Abbildung** f auf $\mathcal{B}$. Falls alle g_i injektiv sind (wie meist bei kontinuierlichem Träger), wird $f(b)$ wie folgt definiert:

$$f(b)(x) := s_i \cdot b(g_i^{-1}(x)) + o_i, \text{ wobei } i \text{ das } i \text{ mit } x \in R_i \text{ ist}$$

Oder, in mengentheoretischer Schreibweise, bei der wir eine Funktion mit ihrem Graphen gleichsetzen:

$$f(b) := \bigcup_{i=1}^{k} \bigcup_{x \in R_i} (x, s_i \cdot b(g_i^{-1}(x)) + o_i)$$

[3] Während die g_i für kontinuierliche Träger normalerweise bijektiv sind, werden im diskreten Fall i.a. mehrere Elemente von D_i auf ein Element von R_i abgebildet. Man spricht dabei auch von *Downfiltering*.

Sind die g_i jedoch nicht injektiv und haben wir einen endlichen Träger, so sieht die Definition wie folgt aus: [4]

$$f(b)(x) := s_i \cdot \left(\frac{1}{|g_i^{-1}(x)|} \sum_{y \in g_i^{-1}(x)} b(y) \right) + o_i, \text{ wobei } i \text{ das } i \text{ mit } x \in R_i \text{ ist}$$

Die Funktion f setzt sich also aus lauter kontraktiven Funktionen für die einzelnen Ranges zusammen. Deswegen nennen wir f gelegentlich auch ein **PIFS**, kurz für **P**artitioniertes **I**teriertes **F**unktionen **S**ystem.

Um einem verbreiteten Mißverständnis vorzubeugen: die *Kontraktivität* der Abbildung beruht auf der Kontraktivität der affin linearen Abbildungen. Die geometrischen Abbildungen müssen *nicht* kontraktiv sein (in Kapitel 5 werden sie es sogar explizit nicht sein), allerdings werden sie in der Praxis gerne so gewählt.

2.3 Metriken

Fraktale Bildkompression ist fehlerbehaftet. Denn die meisten Bilder lassen sich nicht exakt als Attraktor einer wie oben gebauten kontraktiven Funktion schreiben, sondern nur approximieren. Der dabei entstehende Fehler sollte natürlich möglichst wenig den visuellen Eindruck verändern. Nun mag die L_∞-Metrik aus mathematischer Sicht ganz nützlich sein, sie gibt aber nicht den visuellen Unterschied zweier Bilder wieder. Deshalb sollten wir uns noch einmal Gedanken über die Wahl einer geeigneten Metrik für den Bildervergleich machen.

Um es gleich vorweg zu nehmen: es gibt keine Metrik, die in dieser Hinsicht perfekt der menschliche Wahrnehmung entspricht. Das liegt schon daran, daß sich so etwas wie ‘visuelle Ähnlichkeit’ nicht formal fassen läßt. In der Bildverarbeitungspraxis arbeitet man daher mit der L_2-Metrik, die zwar auch nicht der Weisheit letzter Schluß ist, aber zumindest ihren Zweck ganz gut erfüllt.

Definition 2.9 (L_2-Metrik, $||\cdot||$)
Sei $T = \{t_1, \ldots, t_n\}$ und $b_1, b_2 \in \mathcal{B}(T, (F, d))$. Dann setzen wir:

$$L_2(b_1, b_2) := \sqrt{\sum_{i=1}^{n} d(b_1(t_i), b_2(t_i))^2}$$

Wir schreiben auch $||b_1 - b_2||$ für $L_2(b_1, b_2)$. $\square$

[4] Man beachte, daß hierbei $g_i^{-1}(x)$ für alle $x \in R_i$ endlich und nicht leer ist.

Wir haben die L_2-Metrik nur für endliche Träger definiert, da wir sie nur dort brauchen werden. Für unendliche Träger bleiben wir bei der L_∞-Metrik.

Zum Vergleich zweier Bilder werden wir oft den PSNR (**P**eak **S**ignal to **N**oise **R**atio) angeben. Er ist vom L_2-Abstand abgeleitet:

Definition 2.10 (PSNR)
Der PSNR *zwischen zwei verschiedenen Bildern* $b_1, b_2 \in \mathcal{B}(T,(F,d))$ *ist definiert als*

$$20 \cdot \log_{10} \left(\frac{\max\limits_{x,y \in F} d(x,y)}{L_2(b_1,b_2)} \right)$$

Er wird in Dezibel (db) gemessen. □

Je größer der PSNR zwischen zwei Bildern ist, desto ähnlicher sind sie sich. Um ein Gefühl dafür zu bekommen, betrachte man die Abbildungen ab Seite 83. Dort sind einige komprimierte Bilder mit verschiedenen PSNR-Werten zu sehen. Schon bei einem PSNR ab 40 db läßt sich meist kein Unterschied zum Original mehr feststellen.

2.4 Collage Coding

Ziel der fraktalen Kompression ist es, zu einem Bild b eine Kontraktion f zu finden, die den Abstand $||b - \Omega_f||$ minimiert. Dieses Problem wird oft auch als das **inverse Problem** der fraktalen Kompression bezeichnet. In diesem Abschnitt stellen wir die Funktionsweise von ‘Collage Coding’ vor, der Methode, die meistens zu dessen Lösung benutzt wird.

2.4.1 Kodierung

Um ein Bild fraktal kodieren zu können, müssen wir im wesentlichen drei Fragen beantworten.

1. Wie soll das Bild in Ranges R_i partitioniert werden?
2. Wie wählen wir die Domains D_i und Parameter g_i, s_i, o_i?
3. Wie wird dies alles abgespeichert?

Die erste und dritte Frage werden wir in Kapitel 3 diskutieren. Die Antwort auf die zweite können wir jedoch jetzt schon geben. Die optimale Wahl der Parameter ist, wie wir in Kapitel 4 sehen werden, **NP**-hart. Da wir an einer effizienten Lösung interessiert sind, geben wir uns daher mit suboptimalen Parametern zufrieden.

Unser Verfahren für deren Bestimmung heißt **Collage Coding**. Es wird motiviert durch folgenden Satz.

Satz 2.11 (Collage Theorem)
Sei $f : \mathcal{B} \to \mathcal{B}$ kontraktiv bzgl. L_2 mit Kontraktionsfaktor δ und habe den Fixpunkt Ω_f. Dann gilt für alle $b \in \mathcal{B}$:

$$||b - \Omega_f|| \leq \frac{1}{1-\delta} \cdot ||b - f(b)||$$

Beweis: Das ist im wesentlichen eine Folgerung aus dem Banachschen Fixpunktsatz. Ein Beweis findet sich z.B. in [Fisher94a], Korollar 2.1. ■

Satz 2.11 wird zum Anlaß genommen, für ein Bild b diejenige kontraktive Funktion f zu suchen, die $||b - f(b)||$ minimiert. Denn dies beschränkt den Fehler, den man eigentlich minimieren will, nämlich $||b - \Omega_f||$, nach oben.

In Abschnitt 4.6 werden wir untersuchen, wie schlecht das von Collage Coding gelieferte Ergebnis sein kann. In der Praxis sind die damit erzielten Resultate jedoch recht gut.

2.4.2 Collagen

Wie funktioniert Collage Coding genau? Das Ziel ist es, für ein Bild b eine Kontraktion f zu finden, die $||b - f(b)||$ minimiert.

Da die Kontraktion f auf jeder Range R_i unabhängig definiert wird, reicht es dazu, für jede Range den Ausdruck $||(b \upharpoonright R_i) - f(b \upharpoonright D_i)||$ zu minimieren. Das heißt, für jede Range R_i müssen eine Domain D_i und optimale Transformationsparameter s_i und o_i gefunden werden.

Die beste Domain findet man durch Durchprobieren aller (normalerweise polynomiell vieler) Möglichkeiten. Die optimalen s_i und o_i lassen sich dann jeweils leicht berechnen. Denn der zu minimierende Ausdruck $||(b \upharpoonright R_i) - f(b \upharpoonright D_i)||^2$ ist eine quadratische Form in s_i und o_i. Die besten Parameterwerte lassen sich daher mit der Methode der kleinsten Quadrate finden. Details hierzu finden sich z.B. in [Fisher94a], Kapitel 1.

Dieses Verfahren nennt sich *Collage Coding*, da ein Bild b durch eine *Collage* transformierter Teile seiner selbst (eben $f(b)$) approximiert wird. Aus diesem Grund heißt $||b - f(b)||$ auch *Collagefehler*.

Teil II

Praxis

Kapitel 3

Adaptive Partitionierungen

3.1 Einleitung

Fraktale Bildkompression ist eine Methode, Bilder zu komprimieren. Nun werden wir sehen, wie gut das funktioniert. Denn dieses Kapitel dokumentiert den praktischen Teil dieser Diplomarbeit. Das Ziel war es, einen fraktalen Kodierer zu entwickeln, der mit den bislang Besten mithalten kann.

Interessanterweise ist dabei die Konkurrenz gar nicht so groß. Denn trotz mehreren hundert Arbeiten zur fraktalen Bildkompression (siehe Abbildung 1.1) kann man die Anzahl der veröffentlichten Programme an einer Hand abzählen. Zudem sind die meisten davon nur Modifikationen des Programms, das Fisher in [Fisher94b] vorstellte.

Gründe für diesen Mangel an Programmen gibt es mindestens zwei. Erstens ist fraktale Kompression von Michael Barnsley patentiert worden (unter anderem mit [BaSl90]). Die Rechtslage bei solchen Programmen ist also etwas unklar.

Der zweite Grund ist, daß es aufwendig ist, einen praxistauglichen Kodierer zu schreiben. Es ist zwar relativ leicht, einen Kodierer zu entwickeln, der eine sehr gute Bildqualität erreicht. Nur leider geht das meist auf Kosten der Rechenzeit, die dann Stunden oder gar Tage für ein Bild beträgt. Unser Programm braucht demgegenüber nur ein paar Minuten. Das ist sicherlich ein Fortschritt, aber leider noch keine Konkurrenz zu anderen modernen Bildkompressionsverfahren, z.B. basierend auf Wavelets [Shapiro93, SaPe96].

Der Aufbau dieses Kapitels orientiert sich an den Schritten, die die Entwicklung unseres Programms durchlief. Zuerst werden in Abschnitt 3.2 die grundlegenden Designentscheidungen beschrieben und motiviert. Dabei werden vor allem die Unterschiede zu bereits vorhandenen Programmen hervorgehoben.

Im folgenden Abschnitt 3.3 wird der Algorithmus zur Lösung des inversen

Problems dargestellt. Abschnitt 3.4 beschreibt das Abspeichern der Transformationsparameter.

Anschließend wird in Abschnitt 3.5 das Programm besprochen. Nach einigen Worten zur Implementation steht dabei vor allem der Vergleich der gewonnenen Kompressionsresultate mit anderen Verfahren im Vordergrund.

Im letzten Abschnitt 3.6 werden einige Anregungen gegeben, wie man das Programm zukünftig noch erweitern oder verbessern könnte.

3.2 Designentscheidungen

Die in Kapitel 2 dargestellten Grundlagen der fraktalen Bildkompression lassen viele Möglichkeiten offen, wie ein solches Verfahren konkret implementiert werden kann. Auf welche Weise soll das Bild in Ranges unterteilt werden? Wie sollen die zugelassenen Transformationen aussehen? Und wie wird die gefundene Kontraktion dann in einen Code umgesetzt, der letztlich nur aus Nullen und Einsen bestehen darf? Und welche Art von Bildern wollen wir überhaupt zulassen?

Es gibt viele Antworten auf diese Fragen. In diesem Abschnitt werden wir unsere Antwort geben, und sie mit anderen vergleichen.

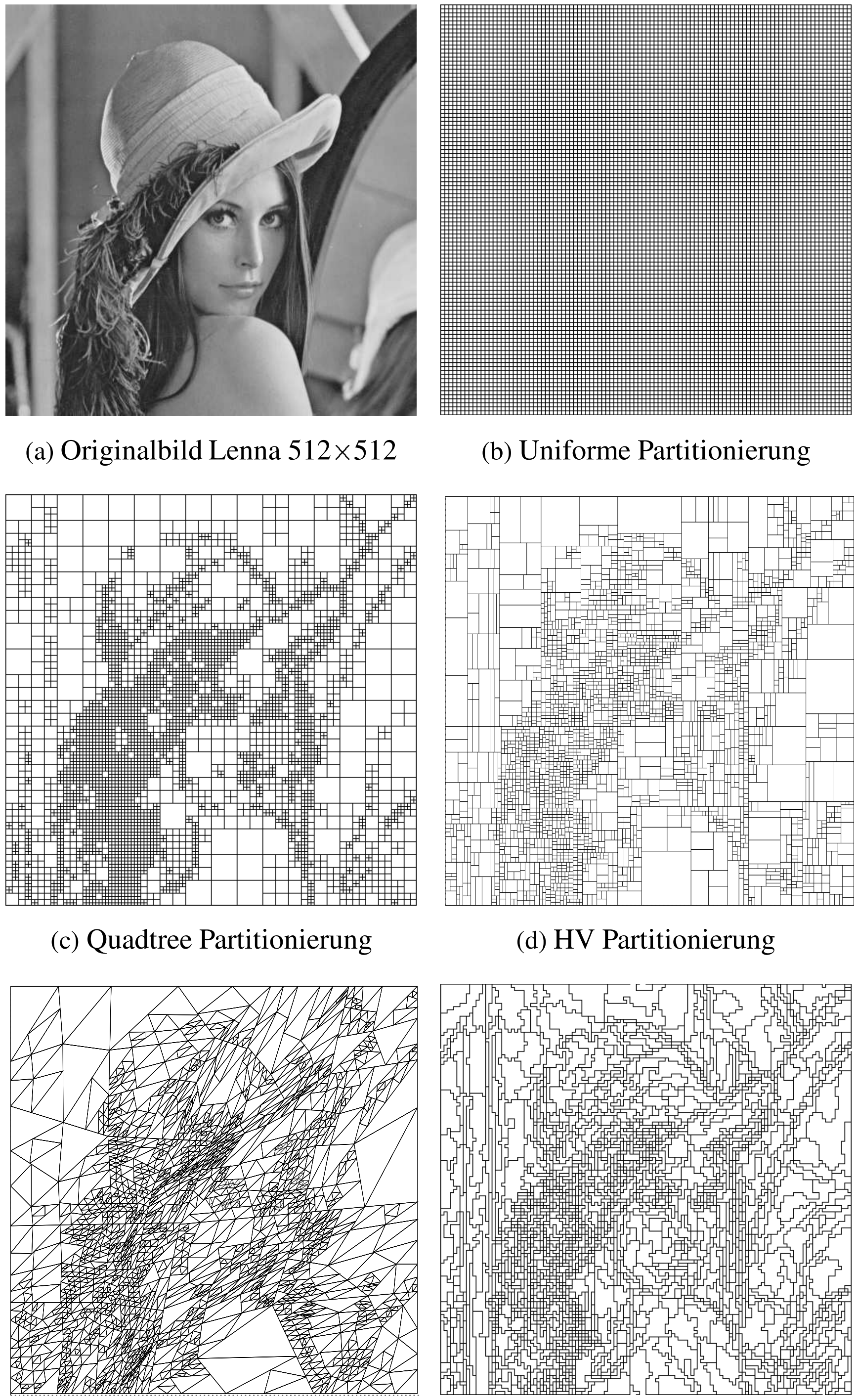
3.2.1 Partitionierungen

Zunächst wollen wir die wohl folgenreichste Designentscheidung betrachten: Welche Partitionierungen des Bildes in Ranges lassen wir zu? Dazu ein kurzer Überblick, wie dieses Problem von anderen gelöst wurde.

Uniform: Die einfachste Möglichkeit ist es, das Bild gleichmäßig in Ranges derselben Größe aufzuteilen (siehe Abbildung 3.1(b)). Dafür übliche Rangegrößen sind 4×4 Pixel oder 8×8 Pixel. Der offensichtliche Nachteil dieser Methode ist es, daß das Verfahren nicht adaptiv ist. Detailreiche Gebiete des Bildes werden mit genau demselben Aufwand kodiert wie detailarme. Deshalb bevorzugt man Verfahren, die die Partitionierung abhängig vom Bildinhalt wählen.

Quadtree: Das am häufigsten verwendete Verfahren ist die Quadtree-Unterteilung [BeDeKe92, JaFiBo92]. Die weite Verbreitung liegt nicht zuletzt daran, daß diese Methode für Fishers Programm [Fisher94b] benutzt wurde. Das Verfahren funktioniert wie folgt:

Man beginnt mit einer uniformen Unterteilung des Bildes, etwa in 32×32 Pixelblöcke, und bestimmt eine Collage für diese Partitionierung. Ist der Fehler in einem der Blöcke größer als ein vorgegebener Toleranzwert, so



(a) Originalbild Lenna 512×512

(b) Uniforme Partitionierung

(c) Quadtree Partitionierung

(d) HV Partitionierung

(e) Partitionierung durch Dreiecke

(f) Adaptive Partitionierung

Abbildung 3.1: Partitionierungen von Lenna wird dieser Block in 4 gleichgroße, kleinere Blöcke unterteilt und für diese wiederum eine Collage gesucht (Abbildung 3.1(c)). Das macht man solange weiter, bis entweder überall die Toleranzbedingung erfüllt ist, oder die Blöcke eine gewisse Mindestgröße erreicht haben.

HV: Eine deutlich bessere, da noch adaptivere, Methode ist die HV-Partitionierung (HV steht für **H**orizontal-**V**ertikal), die von Fisher und Menlove eingeführt wurde [FiMe94]. Hier wird das Bild wie bei der Quadtree-Methode wiederholt in immer kleinere Ranges unterteilt, bis der Collagefehler unterhalb einer vorgegebenen Toleranz bleibt. Die Art der Unterteilung ist jedoch eine andere: am Anfang ist das gesamte Bild eine Range. Dann werden die Ranges, solange nötig, durch horizontale oder vertikale Schnitte in kleinere Ranges aufgeteilt (Abbildung 3.1(d)). Die Schnittgerade wird dabei so gewählt, daß sie möglichst entlang einer Kante in der Range verläuft.

Die mit dieser Methode erzielten Ergebnisse sind unter den besten, die man mit Techniken fraktaler Kompression bis heute erzielt hat. Das Verfahren ist jedoch sehr zeitaufwendig, weshalb bezeichnenderweise auch nur der Dekodierer frei verfügbar ist.[1]

Sonstige: Als weitere Möglichkeiten sind die Unterteilung des Bildes in Dreiecke (siehe [DACB96, Nova93] und Abbildung 3.1(e)) oder andere Polygone [Reus94] untersucht worden. Wegen der auftretenden Probleme, wie z.B. der Vermeidung von degenerierenden Ranges, und aufwendiger Algorithmen, spielen sie jedoch keine wichtige Rolle in der Praxis.

In dieser Arbeit folgen wir Thomas und Deravi [ThDe95]. Dort sind Ranges zusammenhängende Mengen von kleinen Bildblöcken, im folgenden *atomare Blöcke* genannt. Abbildung 3.1(f) zeigt eine solche Partitionierung für Lenna. Typische Werte für die Größe der atomaren Blöcke sind etwa 4×4 oder 8×8 Pixel. Die möglichen Domains für eine Range sind dann jeweils doppelt so groß, und von der gleichen Form. Sie liegen auf einem Raster mit doppelt so großer Schrittweite wie bei den Ranges, also z.B. 8×8 oder 16×16 Pixel.

Der Vorteil dieses Ansatzes ist es, daß sich die Ranges sehr gut an den Bildinhalt anpassen können, was die Qualität der Kodierung erhöht. Allerdings ist das Abspeichern der Partitionierung aufwendiger als bei den anderen genannten Verfahren. In Abschnitt 3.4 werden wir uns mit diesem Problem befassen.

Eine Besonderheit sollte noch bemerkt werden. Für diese Partitionierungsmethode fassen wir das Bild als Torus auf. Das heißt, die untere Bildkante schließt direkt an die obere, und der linke Bildrand schließt direkt an den rechten an. Das

[1] `http://inls3.ucsd.edu/y/Fractals/NewSP/`

erklärt die ‘Löcher’ im Rand von Abbildung 3.1(f): diese Ranges setzen sich am gegenüberliegenden Bildrand fort.

3.2.2 Transformationen

Als Transformationen lassen wir die ‘üblichen’ Abbildungen zu. Das heißt, zu jeder Range wird eine Domain gesucht, die unter einer kontraktiven linearen Abbildung die Range möglichst gut approximiert. Zusätzlich erlauben wir dabei die acht Symmetrieabbildungen (Abbildung 3.2), d.h. die Domain kann auch gedreht oder gespiegelt auf die Range abgebildet werden.

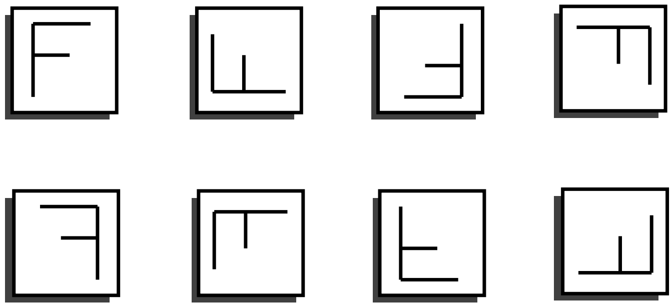


Abbildung 3.2: Die acht Symmetrieabbildungen des Quadrates (obere Reihe: Drehung um $0°, 90°, 180°, 270°$, untere Reihe: Spiegelungen der oberen Reihe)

3.2.3 Bilder

Um das Programm möglichst praxistauglich zu machen, wollen wir sowohl Graustufenbilder als auch Farbbilder kodieren. Außerdem sollen sie beliebige Maße haben dürfen. Das mag jetzt selbstverständlich klingen, aber die wenigsten fraktalen Kodierer leisten dies bislang. Also scheint es sinnvoll, dies einzubauen.

Zur Vereinfachung des Programms beschränken wir uns jedoch darauf, nur Bilder im PPM-Format[2] zu verarbeiten. Dies ist eines der am einfachsten zu handhabenden Grafikformate und zudem recht weit verbreitet.

3.3 Der Algorithmus

Nachdem wir die Art der möglichen Partitionierungen und Transformationen festgelegt haben, müssen wir nun klären, wie wir gute fraktale Codes finden wollen.

[2] `ftp://wuarchive.wustl.edu/graphics/graphics/packages/NetPBM/`

Wir beschränken uns dabei zunächst auf Graustufenbilder. Die Änderungen für Farbbilder werden in Abschnitt 3.3.4 besprochen.

3.3.1 Die Grundidee

Der Algorithmus dafür sieht im Prinzip wie folgt aus. Zunächst wird das Bild in lauter atomare Blöcke unterteilt (wie in Abbildung 3.1(b)). Dann wird, solange nötig, jeweils ein benachbartes Rangepaar zu einer neuen, größeren Range vereinigt (beispielhaft in Abbildung 3.3). Das Programm erzeugt also eine Folge von Partitionierungen, wobei sich die Anzahl der Ranges in jedem Schritt um eins reduziert (siehe Abbildung 3.4 für einige Partitionierungen aus einer solchen Folge). Damit erhöht sich die Kompressionsrate, während sich die Bildqualität verringert. Der Prozeß wird beendet, sobald die gewünschte Kompressionsrate erreicht wurde, oder aber eine Toleranzschranke für die Bildqualität unterschritten wurde.

Beschreiben wir nun den Algorithmus genauer: Die Datenstruktur, die wir zur Verwaltung der auftretenden Partitionierungen verwenden, heißt *Konfiguration*. Sie enthält folgende Elemente:

- Eine *Partitionierung* des Bildes in Ranges $R_1, \ldots, R_n$
- Für jede Range R_i eine Liste von d Domains $D_{i,1}, \ldots, D_{i,d}$.

Dabei ist d ein noch zu wählender Parameter. Der Algorithmus läßt sich damit wie folgt präzisieren:

1. Unterteile das Bild in atomare Blöcke als Ranges. Berechne für jede der Ranges d ‘gute’ Domains. Dies liefert unsere erste Konfiguration. In Abschnitt 3.3.2 wird dieser Schritt genauer beschrieben.
2. Solange die gewünschte Kompressionsrate noch nicht erreicht ist bzw. ein vorgegebener Collagefehler noch nicht überschritten wurde, wiederholen wir folgendes:

   (a) Wir wählen zwei benachbarte Ranges der Partitionierung aus. (Nach welchen Kriterien das geschieht, besprechen wir in Abschnitt 3.3.3.)

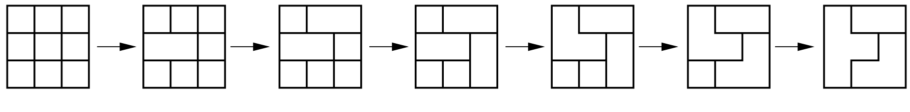
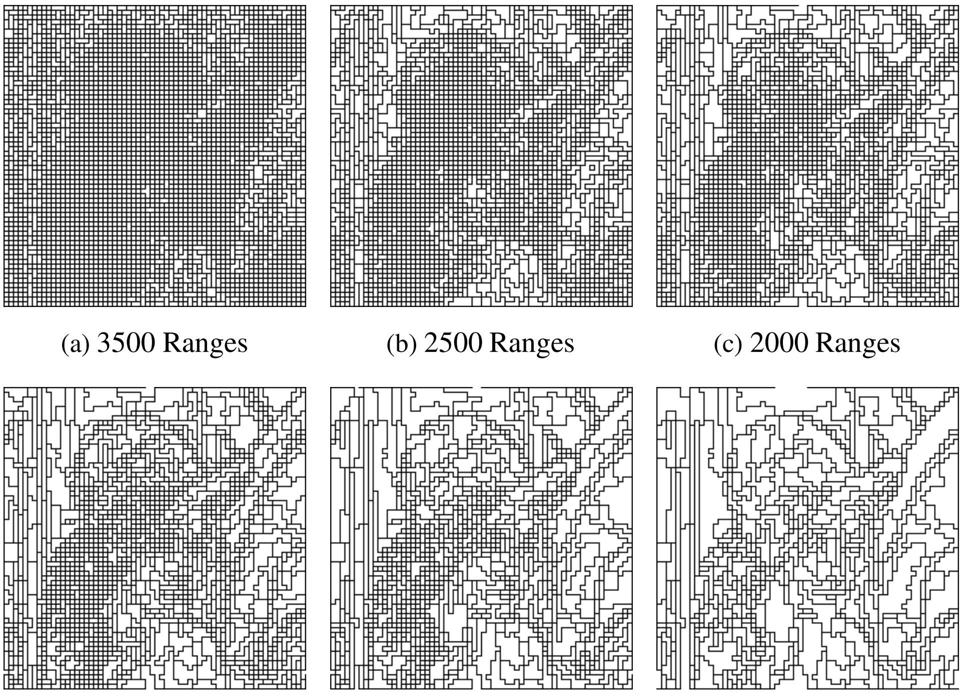


Abbildung 3.3: Folge von Partitionierungen, die durch wiederholtes Vereinigen benachbarter Ranges entstehen



(a) 3500 Ranges (b) 2500 Ranges (c) 2000 Ranges

(d) 1500 Ranges (e) 1000 Ranges (f) 500 Ranges

Abbildung 3.4: Adaptive Partitionierungen von 256×256 Lenna

(b) Dann erzeugen wir eine neue Konfiguration, in der diese beiden Ranges zu einer gemeinsamen Range vereinigt wurden. Der Rest der Konfiguration sieht so aus wie bei der alten Konfiguration.

(c) Die d Domains für die neue Range (Abbildung 3.5(a)) bestimmen wir dann so: Von den Vorgängern der Range haben wir $2d$ Domains geerbt (Abbildungen 3.5(b),3.5(c)). Diese werden analog der zugehörigen Range erweitert, so daß sie doppelt so groß wie die neue Range sind (Abbildung 3.5(d)). Die so erhaltenen Domains werden dann mit der neuen Range verglichen, und die besten d gelangen in die neue Konfiguration.

3. Man gibt die Partitionierung und für jede Range die beste Domain aus der mitgeführten Liste (einschließlich Transformationsparametern) aus. Das wird in Abschnitt 3.4 beschrieben.

3.3.2 Split

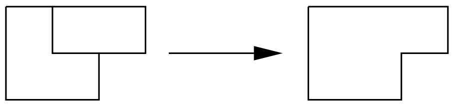
Am Anfang wird das Bild in lauter atomare Blöcke unterteilt, und zu jeder dieser Ranges muß eine Liste von d 'guten' Domains gefunden werden. Wenn die atomaren Blöcke z.B. die Größe 4×4 haben, so sind die möglichen Domains die nicht überlappenden Blöcke der Größe 8×8.

Natürlich gibt es mehrere Möglichkeiten, wie dieses Problem gelöst werden kann. Im folgenden werden wir einige davon beschreiben. Um eine begründete Wahl für eines dieser Verfahren zu treffen, wurden sie alle implementiert. Durch Vergleich von Laufzeit und resultierender Bildqualität werden wir den günstigsten Algorithmus wählen.

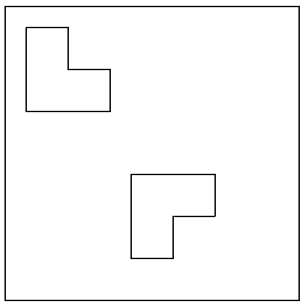
Alle Verfahren wurden dazu auf das Testbild Lenna der Größe 512×512 Pixel (siehe Abbildung B.1(a)) angewandt. Die Größe der atomaren Blöcke war 4×4, die Anzahl der Domainindizes d pro Range war auf 10 gesetzt.[3] Die Zeiten wurden auf einer Silicon Graphics O_2 ermittelt, die mit einem auf 180 MHz getakteten MIPS R5000 Prozessor arbeitet. Die Ergebnisse sind in Tabelle 3.1 zusammengefaßt.

Full search: Die einfachste Methode zuerst. Wir vergleichen jede Range mit jeder möglichen Domain, und nehmen dann für jede Range die d besten Domains. Das liefert sicherlich die beste Bildqualität, dauert aber andererseits durch die vielen Vergleiche auch sehr lange: über vier Stunden (siehe Tabelle 3.1). Deswegen kommt dieses Verfahren für uns zwar nicht in Frage, ist aber trotzdem interessant als Vergleich für die folgenden Methoden.

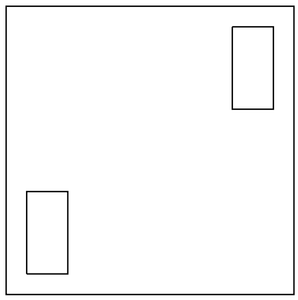
[3] Die Wahl $d = 10$ hat sich in [SaRu96] als vernünftig erwiesen. Deshalb werden wir diesen Wert ab jetzt immer benutzen.



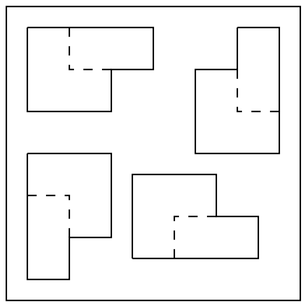
(a) Zwei Ranges werden vereinigt



(b) Domains zur ersten (alten) Range



(c) Domains zur zweiten (alten) Range



(d) Mögliche Domains für die neue vereinigte Range

Abbildung 3.5: Mögliche Domains für eine vereinigte Range

Methode	**Laufzeit**	**PSNR**
Full search	4:33:39	37,38
Variance (75%)	0:57:14	36,06
Variance (90%)	0:20:16	34,14
Variance (95%)	0:09:22	32,33
Variance (99%)	0:01:36	29,74
NN-Search	0:01:36	37,12

Tabelle 3.1: Laufzeit und Bildqualität für verschiedene Splitverfahren

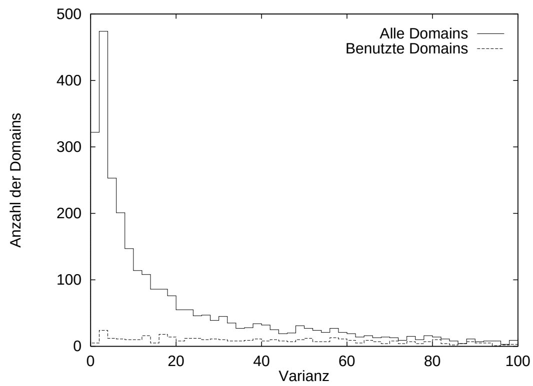


Abbildung 3.6: In der Kodierung wirklich benutzte Domains von Lenna

Variance based speedup: In [Saupe96] bemerkte Saupe, daß bei den üblichen fraktalen Codern Domains mit niedriger Varianz nur selten verwendet werden (Abbildung 3.6). Wir erreichen also eine Beschleunigung, wenn wir die volle Suche so modifizieren, daß wir nur nach Domains mit hoher Varianz suchen. Mit der Laufzeit sinkt natürlich auch die Bildqualität.

In Tabelle 3.1 sind einige Ergebnisse zusammengestellt. Die Prozentzahl in Klammern gibt dabei immer den Anteil der *weggelassenen* Domains an. Der Qualitätsverlust ist unerwartet stark, das Verfahren für uns also ebenfalls nicht geeignet. Anscheinend funktioniert diese Methode besser, wenn sie gleichzeitig mit der Wahl einer Partitionierung benutzt wird, wie auch in [Saupe96] geschehen.

Nearest Neighbor Search: Ranges und (auf Rangegröße verkleinerte) Domains lassen sich als Vektoren auffassen. Sind die atomaren Blöcke zum Beispiel 4×4 groß, so kann man sie auf natürliche Weise als Vektoren des $\mathbb{R}^{16}$, allgemein eines $\mathbb{R}^k$, auffassen.

In [Saupe95] wird folgendes Verfahren beschrieben: Zunächst normalisiert man Range- und Domainvektoren x mittels[4]

$$x \mapsto \frac{x - \langle x, e\rangle e}{||x - \langle x, e\rangle e||}, \quad \text{wobei } e = \frac{1}{\sqrt{k}}(1, 1, \ldots, 1)^T \in \mathbb{R}^k.$$

Der Abstand von normalisierten Range- und Domainvektoren entspricht dann in ordnungserhaltender Weise dem Collagefehler, der bei der betreffenden Abbildung entsteht. Die beste Domain findet man somit, indem man zu einem Rangevektor den nächstliegenden Domainvektor sucht.[5]

[4] Im Falle $||x - \langle x, e\rangle e|| = 0$ ‘normalisieren’ wir den Vektor zum Nullvektor.

[5] Genaugenommen muß man zu einem Rangevektor x den Domainvektor finden, der x oder $-x$ am nächsten liegt.

Dieses Problem ist als ‘Nächster Nachbar Suche’ bereits gut untersucht worden, und es gibt schnelle Algorithmen dafür. In unserer Implementation verwenden wir den approximierenden Algorithmus von Arya et al. [AMNSW94], da er auch als Sourcecode zur Verfügung steht.

Dieser Algorithmus liefert nicht unbedingt den Domainvektor mit dem geringsten Abstand zu einem Rangevektor, sondern einen, dessen Abstand höchstens $(1+\varepsilon)$-mal so groß wie der beste Abstand ist. In unseren Versuchen hat es sich gezeigt, daß es auch gar nicht nötig ist, wirklich den nächsten Nachbarn zu finden, also $\varepsilon = 0$ zu setzen. Die in Tabelle 3.1 gezeigten Ergebnisse wurden erzielt, indem der Algorithmus mit einer Fehlerschranke von $\varepsilon = 10$ lief. Dabei lieferte er eine Liste von $5d$ Domains, von denen dann jeweils die d besten behalten wurden.

Das Ergebnis: eine gute Qualität bei sehr kurzer Laufzeit von unter zwei Minuten.

Die letzte Methode liefert also das beste Qualitäts/Zeit-Verhältnis. Sie wird deshalb in unserem Programm verwandt.

3.3.3 Merge

Wir wissen nun, wie wir unsere Startkonfiguration erhalten. In der daran anschliessenden Iteration wird in jedem Schritt ein Rangepaar vereinigt. Die uns hier interessierende Frage: Wie bestimmen wir das Rangepaar, welches vereinigt wird?

Wiederum gibt es dafür verschiedene denkbare Strategien, die wir nun diskutieren wollen. Wir werden dabei wieder praktische Tests heranziehen, wobei Rechner und Testbild wie im vorangehenden Abschnitt gewählt wurden.

Greedy: Thomas und Deravi [ThDe95], von denen auch die Idee stammt, diese Art von Partitionierungen zu betrachten, schlugen folgende Vereinigungsmethode vor.

Man beginnt mit irgendeinem atomaren Block, dieser wird zur *aktuellen Range*. Zu dieser fügt man (in einer festgelegten Reihenfolge) solange angrenzende atomare Blöcke hinzu, bis der Collage Fehler für die Range einen Toleranzwert überschreitet. Dann ist diese Range fertig. Wenn noch atomare Blöcke übriggeblieben sind, so nimmt man einen von ihnen als neue aktuelle Range und beginnt dasselbe von vorne. Man ist fertig, wenn alle atomaren Blöcke Teil einer Range sind.

Ein wesentlicher Nachteil dieser ‘greedy’-Strategie ist, daß das Ergebnis sehr von der Reihenfolge abhängt, in der die einzelnen atomaren Blöcke

Methode	Laufzeit	PSNR	Kompressionrate
Evolution	1:04:08	32,86 db	26,22:1
Deterministisch	0:01:50	32,57 db	26,57:1

Tabelle 3.2: Laufzeit verschiedener Mergeverfahren bei Iteration von 16384 zu 2000 Ranges

betrachtet werden. Thomas und Deravi schlagen zwar einige Ausgleichsstrategien vor, die atomare Blöcke einer von zwei ‘konkurrierenden’ Ranges zuweisen. Das Resultat läßt jedoch trotzdem (siehe Abbildung 3.7) zu wünschen übrig, was aber auch teilweise an der Implementierung liegen kann.

Leider konnte zu diesem Verfahren in Tabelle 3.2 kein Eintrag gemacht werden. Die in [ThDe95] genannten Ergebnisse wurden auf anderen Rechnern mit anderer Größe der atomaren Blöcke gemacht. Vermutlich wäre dieses Verfahren zwar schneller als die beiden folgenden, würde andererseits aber auch deutlich schlechtere Bildqualität liefern.

Evolution: Die Anzahl aller möglichen Partitionierungen eines Bildes ist riesig. Deshalb machten wir in [SaRu96] den Vorschlag, Ranges nicht-deterministisch zu vereinigen. Das macht auch eine Änderung am Grundalgorithmus vonnöten. Angelehnt an *genetische Algorithmen* verläuft der Merge-Prozeß wie folgt:

Statt nur einer Konfiguration halten wir immer N_p (z.B. 10) Konfigurationen vor, alle mit der gleichen Rangeanzahl. In jedem Schritt werden dann in jeder der N_p Konfigurationen zufällig σ (z.B. 20) Nachbarpaare ausgewählt und zu einer neuen Range vereinigt. Dadurch erhalten wir $N_p \cdot \sigma$ neue Konfigurationen, mit einer Range weniger als zuvor. Von diesen wählen wir die N_p Konfigurationen mit dem geringsten Collage Fehler aus. Damit haben wir eine neue ‘Generation’ von Partitionierungen, mit der wieder das gleiche geschieht, usw.

Wie Abbildung 3.7 zeigt, liefert diese Methode die besten Ergebnisse aller Ansätze. Ihr großer Nachteil ist jedoch die dazu benötigte Rechenzeit. Wie man Tabelle 3.2 entnehmen kann, ist die Laufzeit mit über einer Stunde für praktische Belange nicht akzeptabel.

Deterministisch: Dieser Ansatz wird erstmals hier beschrieben, und findet sich auch in [RuHaSa97]. Er ist an das vorangehende Verfahren angelehnt, aber deterministisch.

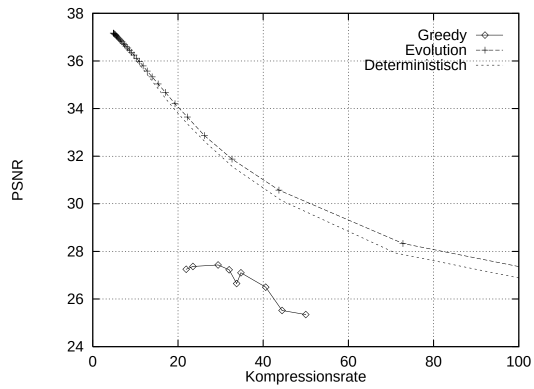


Abbildung 3.7: PSNR-Kurve verschiedener Mergeverfahren, Split bei ‘Evolution’ und ‘Deterministisch’ mit voller Suche

Zusätzlich zu der aktuellen Partitionierung merken wir uns noch eine ‘priority queue’ (PQ), in der alle benachbarten Rangepaare mit dem zusätzlichen Fehler, den die jeweilige Vereinigung bringen würde, gespeichert sind. Auf diese Weise läßt sich bei jedem Schritt leicht das Rangepaar ablesen, dessen Zusammenschluß den geringsten Fehler verursacht.

Wie wird die PQ verwaltet? Am Anfang wird die PQ für die Startkonfiguration erzeugt, indem man für alle Nachbarpaare den Vereinigungsfehler ausrechnet, und sie in die PQ einfügt.

Schwieriger wird die Verwaltung der PQ, wenn zwei Ranges vereinigt werden, da dadurch einige Einträge ungültig werden oder hinzukommen. Durch eine geeignete Pointerverwaltung (siehe unten) können wir jedoch aufwendige Updateoperationen vermeiden. Genauer gesagt machen wir gar keine Updates, sondern ändern nicht mehr aktuelle Einträge erst dann, wenn wir sie aus der PQ lesen.

Um das zu vereinigende Nachbarpaar zu bestimmen, gehen wir wie folgt vor:

1. Wir entnehmen das vorderste Element der PQ. Dann prüfen wir, ob es die beiden zu diesem Element gehörigen Ranges noch gibt, oder ob sie inzwischen in eine Vereinigung verwickelt waren.
2. Falls es sie noch gibt, ist alles in Ordnung. Wir vereinigen die beiden Ranges und sind fertig.
3. Falls sie jedoch durch eine Vereinigung verändert wurden, gibt es zwei Möglichkeiten:
   (a) Die beiden alten Ranges sind inzwischen Teil einer gemeinsamen

Range. In diesem Fall werfen wir diesen Eintrag der PQ einfach fort, und gehen zu Punkt 1 zurück.

(b) Zumindest eine der Ranges liegt inzwischen in einer größeren Range. Falls das eintritt, erzeugen wir einen neuen PQ-Eintrag für die beiden Ranges, in denen die zwei alten Ranges nun liegen, und fügen ihn in die PQ ein. Anschließend gehen wir zu Punkt 1 zurück.

Dieses Vorgehen stellt sicher, daß wir am Ende wirklich das Rangepaar mit dem geringsten Fehler vereinigen. Das liegt an der einfachen Tatsache, daß nicht mehr aktuelle Elemente allenfalls *zu weit vorne* in der PQ eingeordnet sind. Denn sind die beteiligten Ranges inzwischen gewachsen, so kann auch der Vereinigungsfehler nur *größer* geworden sein. Bei Schritt 3b rutscht das Element also im allgemeinen nach hinten.

Außerdem macht man sich induktiv leicht klar, daß jedes benachbarte Rangepaar direkt oder indirekt (durch ein Vorgängerrangepaar, das dann bei Schritt 3b umgewandelt wird) in der PQ vorkommt. Zusammen mit dem 'höchstens zu weit vorne'-Einordnen zeigt das die Korrektheit des Algorithmus.

Nun ein Wort zur Pointerverwaltung. Wir merken uns in einem großen Array für jeden atomaren Block, zu welcher Range er momentan gehört. Nach jeder Vereinigung wird dieses Array aktualisiert, wobei man die anfänglichen und später neu entstehenden Ranges durchnumeriert. Wenn wir dann ein Element der PQ entnehmen, können wir anhand des Arrays leicht feststellen, ob sich die zwei Ranges verändert haben und ob sie inzwischen in eine gemeinsame Range vereinigt wurden.

Wie gut ist dieses deterministische Verfahren in der Praxis? Die Ergebnisse in Abbildung 3.7 zeigen, daß diese Methode zwar etwas schlechtere Qualität als die Evolution liefert. Dafür ist sie aber deutlich schneller (siehe Tabelle 3.2). Da die Qualitätsverluste angesichts des Geschwindigkeitsvorteils vernachlässigbar sind, werden wir dieses Verfahren in unserem Programm benutzen.

3.3.4 Farben

Es gibt mehrere Verfahren, um Farbbilder fraktal zu kodieren. Eine gute Übersicht liefert Abschnitt 2.3 von [DaHa97]. In unserem Programm verwenden wir nur eine ganz einfache Methode, die aber trotzdem erstaunlich gute Ergebnisse erreicht.

Farbbilder liegen im allgemeinen im RGB-Format vor, d.h. das Bild ist in die Komponenten **R**ot, **G**rün und **B**lau aufgespalten. Für die fraktale Kompression ist jedoch das YUV-Format praktischer. Dieses Verfahren wird auch in kommerziellen Implementationen benutzt [Lu96]. Man erhält dieses Format mittels einer linearen Transformation aus dem RGB-Format:

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.148 & -0.289 & 0.437 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Nach dieser Transformation enthält die Y-Komponente des Bildes die Helligkeitsinformation, die U- und V-Komponenten enthalten die Farbinformation. Da das Auge Unterschiede in der Helligkeit stärker wahrnimmt als Farbunterschiede, ist es unser Ziel, vor allem die Y-Komponente gut zu kodieren.

Wir verwenden folgendes Verfahren: die Y-Komponente wird wie ein Graustufenbild fraktal kodiert. Zusätzlich zu dieser Information geben wir für jede Range den *Durchschnitt* der U- und V-Komponenten auf der Range aus. Sie werden dabei im Wertebereich $[-127, 128]$ linear quantisiert auf jeweils 8 Bits. Anschließend wird die Folge dieser Werte `gzip`-komprimiert.

Obwohl dann im resultierenden Bild jede Range einen konstanten Farbwert hat, reicht diese Information aus, um eine visuell sehr gute Bildqualität zu erreichen (siehe Abbildung B.7 auf Seite 89).

3.4 Kodierung

Inzwischen wissen wir, wie wir eine Partitionierung und den zugehörigen fraktalen Code für ein Bild finden. Um diese Informationen jedoch abspeichern zu können, müssen wir sie geeignet kodieren. Wir müssen also einerseits die Parameter der Transformationen und andererseits die Partitionierung übermitteln. Wenden wir uns zunächst dem ersten, leichteren, Problem zu.

3.4.1 Transformationsparameter

Für jede Range R_i müssen die Position der Domain D_i und die Parameter der Abbildung (s_i, o_i und die Art der Symmetrieabbildung) abgespeichert werden. Hierbei folgen wir dem üblichen Vorgehen in der fraktalen Bildkompression. So wird die Position einfach durch die x- und y-Koordinaten kodiert. Die Art der Symmetrieabbildung (siehe Abbildung 3.2) wird durch drei Bits kodiert. Die Parameter s und o quantisieren wir wie in [Fisher94b]. Das heißt, wir quantisieren s mit 5 Bits und o mit 7 Bits wie folgt.

- s wird linear quantisiert, der Wert also zur nächsten Zahl aus $\{k/2^4 \mid k = -2^4, \ldots, 2^4\}$ gerundet:

$$\widehat{s} = \frac{\lfloor 2^4 \cdot (s+1) \rfloor}{2^4} - 1$$

- Anschließend wird o wie folgt quantisiert:

$$\widehat{o} = \begin{cases} \frac{255(1+\widehat{s})}{2^7-1} \lfloor \frac{2^7-1}{255(1+\widehat{s})}(o + 255\widehat{s}) \rfloor - 255\widehat{s} & \text{falls } \widehat{s} > 0, \\ \frac{255(1+\widehat{s})}{2^7-1} \lfloor \frac{2^7-1}{255(1+\widehat{s})} o \rfloor & \text{falls } \widehat{s} \leq 0 \end{cases}$$

  Diese etwas seltsam anmutende Quantisierung nutzt die unregelmäßige Verteilung der Parameter s und o aus (siehe Abbildung 3.15). Genaueres dazu in [Fisher94b].

Was die momentan unmotivierte Wahl der Bitlängen 5 und 7 anbetrifft, so wird sich in Abschnitt 3.6 zeigen, daß es auf den genauen Wert dieser Parameter überraschenderweise gar nicht ankommt.

3.4.2 Partitionierung

Wir folgen hier der Darstellung von [SaRu96]. Die Frage, wie man Partitionierungen effizient abspeichert, wurde aber auch schon in [Freeman61, EdKo85, Leonardi87] behandelt.

Im folgenden vergleichen wir mehrere Verfahren zur Abspeicherung der Partitionierung. Sie sind dabei in der Reihenfolge ansteigender (Implementations-) Schwierigkeit aufgeführt. Abbildung 3.8 zeigt das Ergebnis des Vergleichs. Dort wurde die Codelänge der komprimierten Partitionierungen für Lenna mit 100 bis 16384 Ranges aufgetragen. Die Laufzeit ist bei diesem Teil des Programms erfreulicherweise nebensächlich - sie beträgt in allen Fällen nur Sekundenbruchteile.

Methode 1: Der einfachste Weg, eine Partitionierung wie in Abbildung 3.1(f) zu kodieren, ist der folgende: Für jeden atomaren Block werden zwei Bits abgespeichert. Diese geben an, ob der Block mit seinen rechten und unteren Nachbarn verbunden ist, d.h. ob er mit diesen in einer Range liegt. Die resultierende Bitfolge wird dann mit `gzip`[6] komprimiert. Dies ist eine besonders schnelle Implementation des LZ77-Algorithmus [LeZi77].

Methode 2: Die folgenden Verfahren verwenden einen DCC, kurz für **D**erivative **C**hain-**C**ode, um die Partitionierungsinformationen abzuspeichern. Die Idee

[6] `ftp://ftp.uu.net/pub/archiving/zip/zlib/`

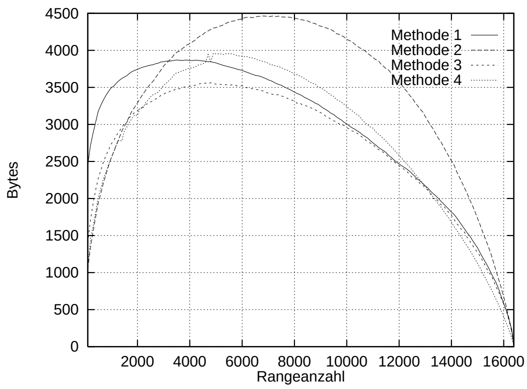


Abbildung 3.8: Codegrößen in Bytes für verschiedene Partitionskodierungsverfahren

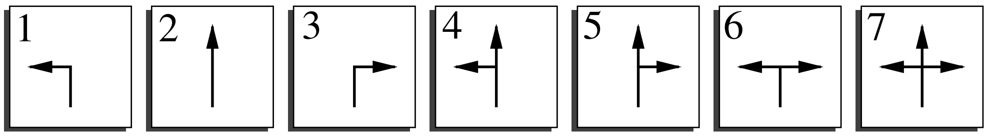


Abbildung 3.9: Symbole zum Kodieren der Rangebegrenzungen

ist es, die Rangegrenzen, also die schwarzen Linien in 3.1(f), entlangzulaufen und die dabei gemachten Bewegungen (geradeaus, links, rechts) auszugeben. Durch Nachvollziehen dieser Bewegungen läßt sich dann die ursprüngliche Partitionierung zurückerhalten. Konkreter sieht der Algorithmus wie folgt aus.

Solange die Partitionierung noch nicht vollständig beschrieben ist, tue folgendes:

Wähle eine noch nicht besuchte Stelle auf den Rangebegrenzungen, speichere ihre Position und vier weitere Bits, die angeben, ob die Rangebegrenzung nach oben, rechts, unten oder links weitergeht. Die erste dieser Richtungen, in der eine Fortsetzung existiert, wird zugleich unsere *aktuelle Bewegungsrichtung*.

Nun bewegen wir uns in der aktuellen Bewegungsrichtung ein Feld vorwärts. Mit der Ausgabe eines der in Abbildung 3.9 gezeigten sieben Symbole geben wir an, in welche der drei Richtungen links, geradeaus, rechts wir unsere Bewegung fortsetzen können. Die erste mögliche dieser Richtungen wird unsere neue aktuelle Bewegungsrichtung. Gibt es noch andere Fortsetzungsmöglichkeiten, so werden diese auf einen Stack gelegt, aber zunächst nicht weiter verfolgt.

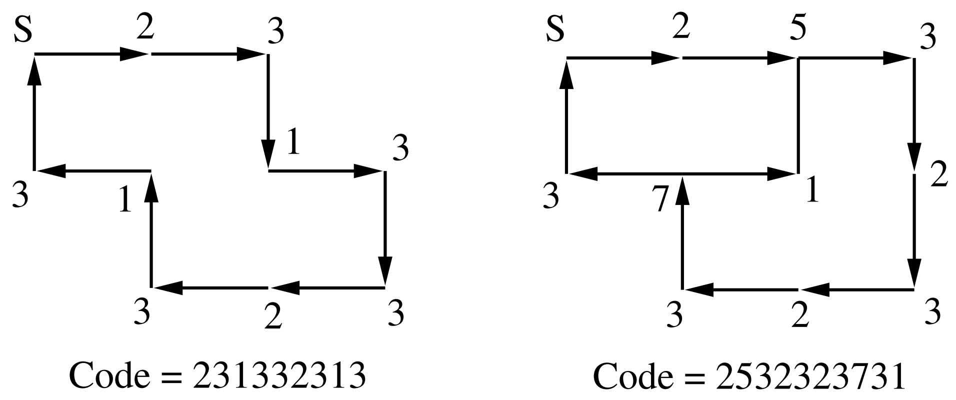


Abbildung 3.10: Zwei Rangezüge mit zugehörigen Codes nach Methode 2. Man startet jeweils an Position **S** und läuft die Begrenzung wie abgebildet entlang. Das Ausgeben der Fortsetzungsmöglichkeiten an jedem Punkt liefert die gezeigten Codes. Im rechten Beispiel wird bei jeder Abzweigung die nicht genommene Richtung auf einen Stack gelegt. Das führt dazu, daß nach Rückkehr zu **S** die zweite Möglichkeit am mit **7** bezeichneten Punkt genommen wird, was die abschließende **1** im Code ergibt.

Irgendwann werden wir bei unseren Bewegungen wieder an eine Stelle kommen, die wir bereits besucht haben. Falls der Stack leer ist, sind wir fertig. Im anderen Fall holen wir das oberste Element von Stack herunter und machen dort weiter, wo wir damals aufgehört haben. In Abbildung 3.10 sind beispielhaft zwei Rangebegrenzungen mit zugehörigem Code gezeigt.

Der resultierende Code besteht also aus

- den Positionierungsanweisungen für jede Zusammenhangskomponente der Partitionierung, und
- einer Kette der sieben Symbole, die die Bewegung beschreiben.

Letztere komprimieren wir noch weiter mittels eines arithmetischen Entropiekodierers. Das Resultat sieht man in Abbildung 3.8. Sie ist jedoch meistens schlechter als die erste Methode. Woran liegt das?

Methode 3: Der wesentliche Nachteil des soeben vorgestellten Verfahrens ist die Tatsache, daß Liniensegmente oft mehrfach (redundant) abgespeichert werden, wie Abbildung 3.11 zeigt. Denn durchläuft man die Begrenzung wie dort gezeigt, so wird das Segment S zweimal kodiert. Das erste Mal beim Erreichen von Punkt A, das zweite Mal beim Passieren von Punkt B.

Bei Methode 3 achten wir deshalb während des Kodierens darauf, nur Segmente abzuspeichern, deren Existenz aus dem bislang geschriebenen Co-

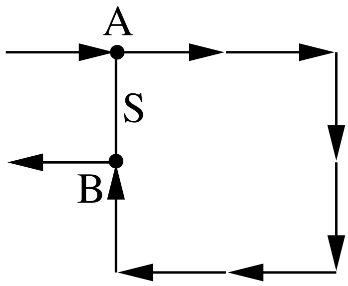


Abbildung 3.11: Segment S wird sowohl beim Passieren von A als auch von B abgespeichert

de noch nicht erschlossen werden können. Dies macht den Code deutlich kürzer.

Wir speichern dabei nach jedem Bewegungsschritt zwischen null und drei Bits ab. Diese geben für die noch nicht Erschließbaren der Richtungen links, geradeaus, rechts an, ob es dorthin eine Fortsetzung gibt. Der resultierende Bitstream wird dann `gzip`-komprimiert.

Methode 4: Ähnlich wie im gerade vorangegangenen Verfahren speichern wir nur die Richtungen ab, deren Existenz noch unbekannt ist. Wir benutzen dazu allerdings keinen Bitstream, sondern nehmen wieder Symbole.

Genauer drei Mengen von Symbolen: je nachdem, ob eine, zwei oder alle drei der Richtungen noch unbekannt sind, benutzen wir ein Codebuch der Größe 2, 4 oder 7. Vor dem Abspeichern werden jedoch die beiden kleineren Codewörterbücher in das große, 7-elementige, abgebildet. Dabei werden die Codeworte des 4-elementigen Wörterbuchs auf die vier häufigsten Codeworte aus dem großen Buch, und die zwei Codeworte des kleinsten Wörterbuchs dann auf die zwei häufigsten Elemente des resultierenden Codebuchs abgebildet. Da wir außerdem abspeichern, wie häufig jedes der 13 Symbole war, ist diese Abbildung umkehrbar. Denn durch das Hinzufügen der Symbole der kleinen Wörterbücher werden die häufigsten Symbole im 7-elementigen Codebuch allenfalls noch 'häufiger'. Weiterhin weiß der Dekodierer zu jedem Zeitpunkt, aus welchem Codebuch er ein Symbol zu erwarten hat.

Dieses 'Abbilden auf die häufigsten Codeworte' dient dazu, die Entropie des resultierenden Symbolstrings zu erhöhen. Dieser wird dann wie in Methode 2 mit einem arithmetischen Entropiekodierer komprimiert.

Wie man Abbildung 3.8 entnehmen kann, liefern in fast allen Fällen entweder Methode 3 oder Methode 4 den kürzesten Code. Deswegen werden in unserer

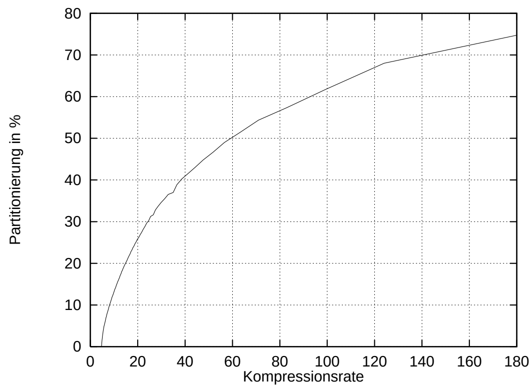


Abbildung 3.12: Anteil der Partitionierungsinformationen am Gesamtcode

Implementierung zunächst beide Codes berechnet. Dann wird ein Bit ausgegeben, das angibt, welche Kodierungsmethode den kürzeren Code liefert. Anschließend wird der entsprechende Code ausgegeben.

In Abbildung 3.12 sieht man, wie groß der Anteil der Partitionierungsinformationen an Gesamtcode ist (den Rest machen die Transformationsinformationen aus). Es zeigt sich, daß er vor allem bei stark komprimierten Bildern den Großteil des Codes ausmacht.

3.5 Das Programm

3.5.1 Implementation

Der in den letzten Abschnitten geschilderte Algorithmus wurde implementiert. In der Programmiersprache C++ geschrieben, umfaßt der Sourcecode etwa 1500 Zeilen, den `gzip`-Code nicht mitgerechnet. Es ist portabel geschrieben, müßte also auf jedem vernünftigen UNIX-Derivat laufen. Mit kleinen Änderungen sollte er aber auch auf Windows oder anderen Betriebssystemen funktionieren. Die ‘manual pages’, die die Bedienung des Programms erläutern, sind in Anhang A wiedergegeben.

Es ist geplant, das Programm demnächst im Internet zur Verfügung zu stellen.

3.5.2 Ergebnisse

Wie gut sind die Ergebnisse, die unser Programm erzielt? Wie hält es sich im Vergleich zu anderen Verfahren? Fragen, auf die wir nun kurz eingehen wollen.

Unser Ziel war es, einen Kodierer zu entwickeln, der mit den bislang besten fraktalen Kodierern mithalten kann. Einer der besten ist Fishers HV-Coder

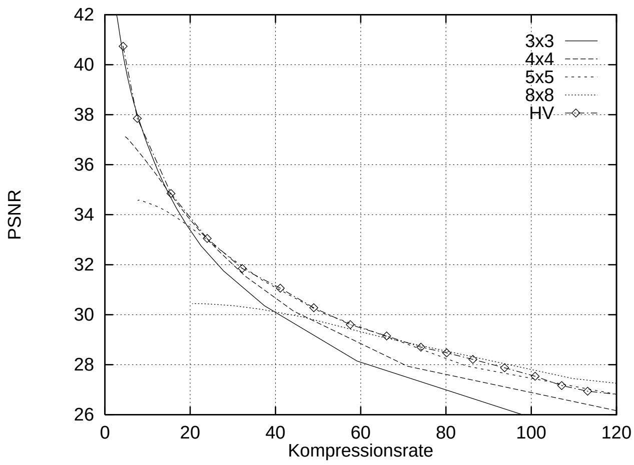


Abbildung 3.13: Verschiedene Größen der atomaren Blöcke

([FiMe94]). In Abbildung 3.13 sieht man einen Vergleich zwischen diesem und unserem Kodierer. Man sieht, daß wir die HV Ergebnisse durch Variieren der Größe der atomaren Blöcke immer erreichen oder übertreffen können. Ab einem Kompressionsgrad von mehr als 70:1 ist unser Verfahren sogar immer besser.

Bei kleineren Kompressionsraten und einer festen Größe der atomaren Blöcke sind wir jedoch manchmal schlechter. Ist es dann fair, von einem Erreichen der Qualität von HV zu sprechen? Ja, denn dieses Diagramm vernachlässigt zweierlei. Erstens benutzt Fisher ein ausgeklügeltes Postprocessing nach dem Dekodieren, um Kompressionsartefakte zu beseitigen. Wir machen nichts dergleichen.

Und zweitens haben wir noch nichts über die Rechenzeit gesagt, die zum Erreichen der gezeigten Ergebnisse nötig ist. Ein genauer Vergleich mit [FiMe94] ist wegen der unterschiedlichen verwendeten Rechner nicht möglich. Doch auch bei Berücksichtigung dieser Problematik kann man aus den dort angegebenen Zeiten von teilweise mehreren Tagen schließen, daß unsere Implementation deutlich schneller ist. Es ist also realistisch, unser Programm mehrfach laufen zu lassen, um eine optimale Blockgröße zu bestimmen, ohne insgesamt länger als HV zu brauchen.

In Anhang B sind die Ergebnisse einiger Testläufe gesammelt, und (soweit möglich) mit anderen Verfahren zur fraktalen Kompression verglichen. Im einzelnen findet sich dort:

- Auf den Abbildungen B.1, B.2, B.3 und B.4 sind neben einem Originalbild die PSNR-Kurve und vier dekomprimierte Bilder mit verschiedenen PSNR-Werten zu finden. Die hier verwendeten Bilder (Lenna, Boat, Mandrill, Barbara) sind ‘Standardbilder’, die in der Bildverarbeitung gerne zu

Tests herangezogen werden.

- Abbildung B.5 macht deutlich, daß ein PSNR-Wert allein sehr wenig über die Qualität eines Bildes aussagt. Diese Abbildung zeigt Lennas mit gleichem PSNR, jeweils von unserem Verfahren und mit Fishers Quadtree Verfahren ([Fisher94b]) komprimiert. Man sieht jedoch deutlich, daß das von uns komprimierte Bild 'besser aussieht'. Ein Grund dafür könnten die adaptiven Partitionierungen sein, die nicht nur besser auf den Bildinhalt eingehen können, sondern durch ihre Unregelmäßigkeit auch weniger Blockstrukturen erkennen lassen.

- In Abbildung B.6 sind einige Partitionierungen mit dem zugehörigen Bild gezeigt. Dies zeigt sehr schön, wie sich die Partitionierungen an das Bild anpassen.

- Und schließlich sind in Abbildung B.7 einige komprimierte Farbbilder zu sehen. Hierbei sind keine PSNR-Werte oder ähnliches angegeben. Das hat folgenden Grund: wir haben eben schon gesehen, daß ein PSNR-Wert nur mäßig gut ist für den Vergleich zweier Grauwertbilder. Für Farbbilder ist das ganze noch viel fraglicher – hier gibt es keine anerkannte Methode, um Bilder zu vergleichen.

  Aber auch in Abwesenheit eines numerischen Vergleichs kann man sich von der Bildqualität überzeugen.

3.6 Nicht das letzte Wort

Wie kann man unser Programm noch verbessern? Hierzu einige Vorschläge.

Quantisierung von s, o: Die Parameter s und o wurden bei uns im wesentlichen linear quantisiert mit 5 bzw. 7 Bits. Naheliegend wäre es, hierfür andere Bitlängen zu wählen. Entsprechende Tests zeigten jedoch überraschenderweise, daß die Wahl der Parameter kaum eine Rolle spielt (siehe Abbildung 3.14). Denn auch für die Wahlen 5/6, 4/6, 4/7 erhalten wir fast das gleiche Ergebnis.

Einsparungen wären jedoch möglich, wenn man berücksichtigt, daß die beiden Parameter nicht gleichmäßig über ihren Wertebereich verteilt sind (siehe Abbildung 3.15). Durch eine aufwendigere Quantisierung sind hier noch Verbesserungen möglich (siehe z.B. [HaSaBa97]).

Quantisierung der Position von D_i: Auch beim Abspeichern der Domainposition könnte eine Korrelation zwischen verschiedenen Werten vorhanden sein,

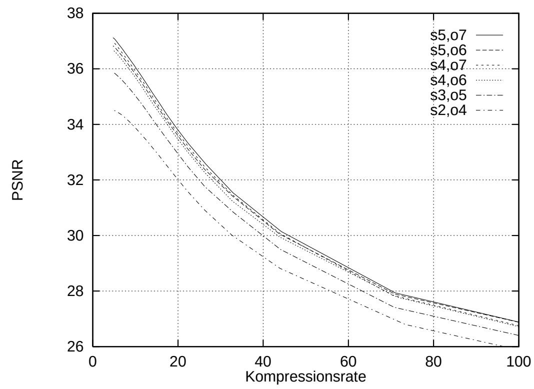


Abbildung 3.14: Verschiedene Bitlängen zur Quantisierung der Parameter s, o (für Lenna)

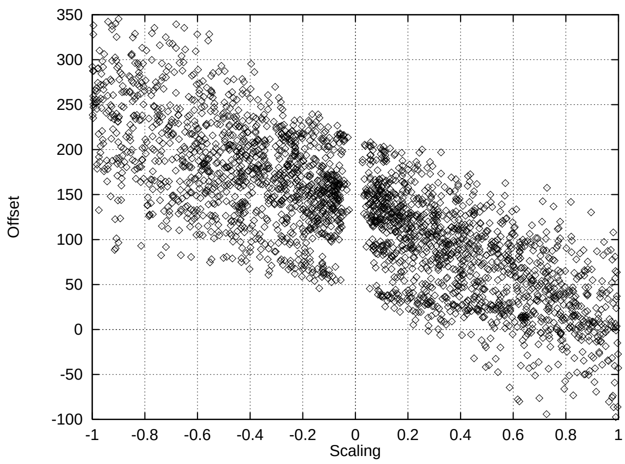


Abbildung 3.15: Verteilung der s, o-Parameter für Lenna

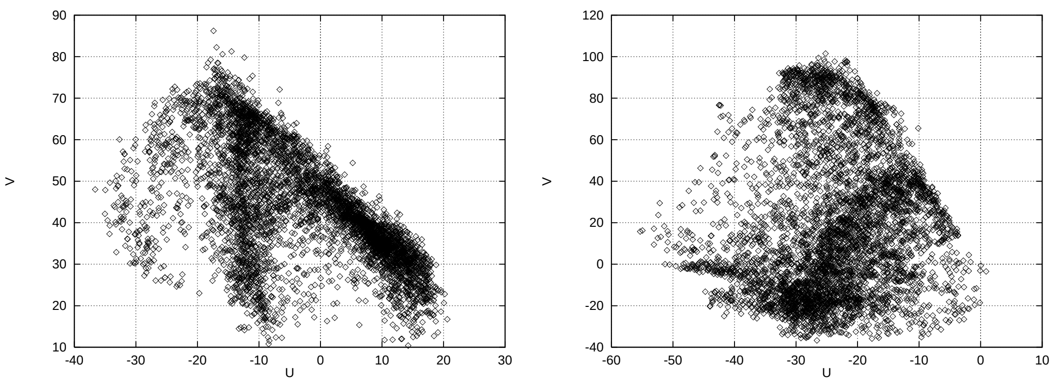


Abbildung 3.16: Verteilung der U,V-Parameter für Lenna (links) und Peppers (rechts)

deren Ausnutzung zu einer verkürzten Darstellung führt. Im Falle der uniformen Rangeaufteilung konnten damit schon einige Erfolge erzielt werden [BaVoNo93]. Es ist jedoch nicht klar, ob und wie diese Ergebnisse auf adaptive Partitionierungen übertragen werden können.

Quantisierung der Farbwerte: Die Quantisierung der U- und V-Werte bei der Farbkodierung wurde weitgehend 'ad hoc' vorgenommen. Die guten Ergebnisse geben dem zwar teilweise recht, aber ein Ausnutzen der unregelmäßigen Verteilung der Parameter (siehe Abbildung 3.16) könnte die Kompression noch verbessern.

Größe der atomaren Blöcke: Wie Abbildung 3.13 zeigt, variiert die optimale Größe der atomaren Blöcke bei verschiedenen Kompressionsraten. Man könnte sich ein Verfahren überlegen, welches diese Größe je nach gewünschter Kompressionsrate oder Bildqualität optimal wählt.

Partitionierungskodierung: Auch die Kodierung der Partitionierung kann sicherlich noch verbessert werden. Wir haben in Abbildung 3.12 gesehen, daß gerade bei großen Kompressionsraten das Abspeichern der Partitionierung den Großteil der Codelänge ausmacht. Allerdings sollte man sich davon nicht zuviel versprechen, da die Gesamtcodelänge in diesen Fällen nicht sehr groß ist.

Teil III

Theorie

Kapitel 4

Optimale fraktale Kompression ist NP-hart

4.1 Einleitung

Wie bei jedem verlustbehafteten Kompressionsverfahren ist es bei der fraktalen Bildkompression das Ziel, mit einer hohen Kompressionsrate eine möglichst gute Annäherung des ursprünglichen Bildes zu liefern. Aus dieser Sicht stellt sich Kompression als Optimierungsproblem dar. Natürlich spielt nicht nur die 'Qualität' der Kodierung eine Rolle, sondern auch die Zeit, die man braucht, um sie zu finden. Man ist heutzutage angesichts immer leistungsfähigerer Computer nicht mehr willens, Stunden oder gar Tage auf die Komprimierung eines Bildes zu warten. Wir sind also daran interessiert, daß ein Kodierungsprogramm möglichst *schnell* arbeitet.

In diesem Kapitel untersuchen wir die Frage, wie schnell es prinzipiell möglich ist, ein Signal 'optimal' fraktal zu kodieren. Dazu formulieren wir im folgenden Abschnitt 4.2.1 das Problem, ein Signal möglichst gut fraktal zu kodieren, als Optimierungsproblem. In Abschnitt 4.2.2 erinnern wir an einige Begriffe aus der Komplexitätstheorie.

Mit dieser Vorarbeit können wir dann in Abschnitt 4.2.3 unsere Hauptaussage formulieren, nämlich daß das Finden eines optimalen fraktalen Codes **NP**-hart ist. Das heißt, daß es keinen polynomiellen Algorithmus für dieses Problem gibt, falls $\mathbf{P} \neq \mathbf{NP}$ gilt.

Der Beweis unserer Hauptaussage erfolgt in den Abschnitten 4.3 bis 4.5. Anschließend zeigen wir in Abschnitt 4.6, daß der von einem Collage Coder gelieferte Code beliebig weit vom optimalen Code entfernt sein kann. Und im letzten Abschnitt dieses Kapitels diskutieren wir die praktischen Konsequenzen der gewonnenen Resultate.

Dies ist übrigens das erste Mal, daß die Frage nach der Komplexität der fraktalen Kompression überhaupt gestellt (und beantwortet) wurde. Eine verkürzte Version dieses Kapitels findet sich in [RuHa97].

4.2 Optimale fraktale Kompression

4.2.1 Ein Modell der fraktalen Kompression

In diesem Kapitel wollen wir Aussagen über die Schwierigkeit von fraktaler Kompression machen. Dazu müssen wir zunächst klären, welche Art von fraktaler Kompression wir untersuchen wollen.[1]

Der Einfachheit halber betrachten wir die Kodierung von eindimensionalen Signalen. Das heißt, alle Signale in diesem Kapitel sind endliche Folgen von ganzen Zahlen, also Elemente von $\mathbb{Z}^k$ mit geeignetem k.

Des weiteren geben wir die Partitionierung fest vor, und zwar wählen wir die uniforme Partitionierung. Das heißt, die Signale werden in gleichgroße Ranges zerlegt, etwa das Signal $(x_1,\dots,x_{nm}) \in \mathbb{Z}^{nm}$ in die Ranges $R_1 = (x_1,\dots,x_m)$, $R_2 = (x_{m+1},\dots,x_{2m})$, $\dots$, $R_n = (x_{(n-1)m+1},\dots,x_{nm})$.

Die möglichen Domains[2] sind die Vereinigungen von Paaren aufeinanderfolgender Ranges, also $\mathcal{D}_1 = R_1R_2 = (x_1,\dots,x_{2m})$, $\mathcal{D}_2 = R_3R_4$, $\dots$, $\mathcal{D}_{\lfloor n/2\rfloor} = R_{2\lfloor n/2\rfloor-1}R_{2\lfloor n/2\rfloor}$. Beim Abbilden einer Domain auf eine Range wird die Domain durch Mittelwertbildung benachbarter Elemente auf Rangelänge geschrumpft.

Definition 4.1 (Fraktaler Code)
Sei $S \in \mathbb{Z}^{nm}$ ein Signal, partitioniert in n Ranges mit jeweils m Elementen. Ein fraktaler Code *C zu S ist ein $3n$-Tupel $C = (d_i, s_i, o_i)_{i=1,\dots,n}$.*

Dabei ist $d_i \in \{1,\dots,\lfloor\frac{n}{2}\rfloor\}$ der Index der Domain, die auf R_i abgebildet wird, und s_i, o_i sind die Parameter der zugehörigen Transformation. Für sie gilt $s_i \in [-s_{max}, s_{max}]$ mit einem $s_{max} \in]0,1[$ und $o_i \in \mathbb{Z}$. □

Definition 4.2 ($\mathcal{C}(n)$)
Die Menge aller fraktalen Codes zu einem in n gleichgroße Ranges partitionierten Signal bezeichnen wir mit $\mathcal{C}(n)$. □

Definition 4.3 (Ω_C)
Ein Code $C \in \mathcal{C}(n)$ induziert eine kontraktive Abbildung im $\mathbb{R}^{nm}$, wenn m die

---

[1] In Abschnitt 4.7 werden wir dann darauf eingehen, wie sich die Ergebnisse auf andere Varianten der fraktalen Kompression übertragen lassen.

[2] Man beachte den feinen Unterschied in der Notation. Die überhaupt *möglichen* Domains heißen $\mathcal{D}_i$. Demgegenüber ist $D_i = \mathcal{D}_{d_i}$ die der Range R_i zugeordnete Domain (siehe Kapitel 2).

Länge der Ranges ist.[3] *Den eindeutig bestimmten Fixpunkt bezeichnen wir (in Anlehnung an Ω_f) mit $\Omega_C \in \mathbb{R}^{nm}$.* □

Fraktale Bildkompression ist auf natürliche Weise ein Optimierungsproblem: Gegeben ist ein Signal $S \in \mathbb{Z}^{nm}$, partitioniert in n Ranges mit jeweils m Elementen. Gesucht ist der Code $C_{opt} \in \mathcal{C}(n)$, der folgenden Ausdruck erfüllt:[4]

$$||\Omega_{C_{opt}} - S|| = \min_{C \in \mathcal{C}(n)} ||\Omega_C - S||$$

Dabei bezeichnet $||\cdot||$ wie üblich die L_2-Metrik.

Für den Beweis der **NP**-Härte ist es jedoch praktischer, dies als Entscheidungsproblem zu definieren.

Definition 4.4 (FRACCOMP)
FRACCOMP *ist folgendes Entscheidungsproblem:*

Gegeben: *Signal $S \in \mathbb{Z}^{nm}$, partitioniert in n Ranges mit jeweils m Elementen, und eine Zahl $b \in \mathbb{N}$.*

Frage: *Existiert ein Code $C \in \mathcal{C}(n)$ mit $||\Omega_C - S||^2 \leq b$?* □

4.2.2 Etwas Komplexitätstheorie

Die Mittel zu entscheiden, ob ein Berechnungsproblem leicht oder schwer zu lösen ist, gibt uns die Komplexitätstheorie an die Hand. In diesem Abschnitt wollen wir kurz an einige Begriffe erinnern, die wir später brauchen werden. Für eine umfassende Einführung in dieses Thema vergleiche man etwa [Papa94].

Ziel der Komplexitätstheorie ist es vor allem, untere Schranken für die Laufzeit von Algorithmen zu bestimmen, die ein Problem lösen. Eine wichtige Klasse von Problemen ist **P**, die Probleme, die sich in polynomiell von der Eingabegröße abhängiger Zeit lösen lassen. Man sagt, daß ein Problem genau dann *effizient* bzw. *schnell* lösbar ist, wenn es in **P** liegt.

Eine weitere wichtige Klasse von Problemen ist **NP**. Das ist die Menge der Probleme, die in polynomieller Zeit von einem nichtdeterministischen Programm gelöst werden können. Vereinfacht gesagt, darf das Programm bei seiner Berechnung auf gewisse Weise raten. Für uns ist hier nur wichtig, daß einerseits **P** $\subset$ **NP** gilt, andererseits jedoch auch folgende Vermutung:

---

[3] Die Länge der Ranges wird immer aus dem Zusammenhang klar sein. Deswegen schreiben wir nicht $\Omega_{C,m}$, auch wenn das vielleicht korrekter wäre.

[4] Das Minimum auf der rechten Seite existiert immer. Denn $||\Omega_C - S||$ hängt für jede der endlich vielen Wahlmöglichkeiten der d_i stetig von den Parametern s_i und o_i ab. Die s_i bewegen sich nur auf einem kompakten Intervall. Und auch die o_i können als beschränkt angenommen werden, da mit sehr großen o_i auch der Fehler $||\Omega_C - S||$ sehr groß wird. Als stetige Funktion auf einem Kompaktum hat $||\Omega_C - S||$ somit ein Minimum.

Vermutung 4.5 (Cook 1971)
$\mathbf{P} \neq \mathbf{NP}$ □

Der Beweis dieser Vermutung ist wohl das bekannteste ungelöste Problem der (theoretischen) Informatik. Sie wurde zwar schon vor über 25 Jahren aufgestellt, hat sich bis heute aber standhaft jedem Versuch widersetzt, bewiesen oder widerlegt zu werden. Trotzdem hat sich allgemein die Meinung durchgesetzt, daß die Vermutung stimmt. Wir werden sie daher im folgenden stillschweigend voraussetzen.

Neben den Komplexitätsklassen brauchen wir den Begriff der Reduktion. Dabei identifizieren wir ein Berechnungsproblem mit der durch es induzierten Funktion (Eingabe $\rightarrow$ Ausgabe).

Definition 4.6 (P-reduzierbar)
Seien f und g berechenbare Funktionen. Dann heißt f **P**-reduzierbar *auf g, wenn es Verfahren $h_1, h_2 \in \mathbf{P}$ gibt, so daß gilt $f = h_2 \circ g \circ h_1$.* □

Wir können diese Definition so deuten: h_1 wandelt die Eingabe von f in polynomieller Zeit in eine Eingabe von g um. Die Ausgabe, die g daraufhin liefert, setzen wir in h_2 ein, was schließlich die Ausgabe von f auf die ursprüngliche Eingabe liefert.

Das wichtigste hier ist folgendes: Wenn f **P**-reduzierbar ist auf g, so ist g mindestens so schwer zu berechnen wie f. Schließlich läßt sich f ohne viel Mehrarbeit unter Verwendung des Algorithmus von g berechnen.

Alle Probleme, die in diesem Sinne mindestens so schwer wie alle Probleme in **NP** sind, nennt man **NP**-hart.

Definition 4.7 (NP-hart)
Ein Problem Π heißt **NP**-hart, *wenn alle Probleme aus* **NP** *auf Π* **P***-reduzierbar sind.* □

Ein Beispiel für ein **NP**-hartes Problem ist MAXCUT.

Definition 4.8 (MAXCUT)
MAXCUT *ist folgendes Entscheidungsproblem:*

Gegeben: *Ein (ungerichteter) Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$.*

Frage: *Gibt es eine Partitionierung $V = V_1 \dot{\cup} V_2$, so daß die Anzahl der Kanten, die von V_1 nach V_2 gehen, größer als k ist? In Formeln: $|E \cap (V_1 \times V_2)| \geq k$.*

Die Wahl der Teilmengen V_1, V_2 wird auch Cut *genannt, die Anzahl der Kanten von V_1 nach V_2 heißt* Größe *des Cuts.* □

Die **NP**-Härte von MAXCUT wurde 1972 von Richard Karp gezeigt [Karp72]. Eine umfassende Sammlung von **NP**-harten Probleme findet sich in [GaJo79, CrKa95].

4.2.3 Der Satz

Wir wollen nun folgenden Satz zeigen:

Satz 4.9
MAXCUT *ist* **P**-*reduzierbar auf* FRACCOMP.

Denn daraus folgt dann sofort die Hauptaussage dieses Kapitels.

Satz 4.10
FRACCOMP *ist* **NP**-*hart.*

4.3 Beweisüberblick

4.3.1 Das Entscheidungsgadget

Abbildung 4.1 zeigt ein Signal. Zur Vereinfachung ist das Signal nicht als Zahlenfolge, sondern als Treppenfunktion gezeichnet – wir werden das im folgenden immer tun, da das die Darstellung anschaulicher macht.

Das Signal besteht aus den Ranges $R_1, \ldots, R_7$ und den Domains $\mathcal{D}_1 = R_1R_2$, $\mathcal{D}_2 = R_3R_4$, $\mathcal{D}_3 = R_5R_6$. Wir wollen die Ranges R_5, R_6, R_7 fraktal kodieren. Dabei nehmen wir an, daß die Transformationen des fraktalen Codes nur wie folgt abbilden dürfen. Auf die Ranges R_5 und R_6 müssen die Domains R_1R_2 oder R_3R_4 abgebildet werden und für R_7 können wir nur die Domain R_5R_6 wählen. Außerdem setzen wir immer $s = 1, o = 0$.

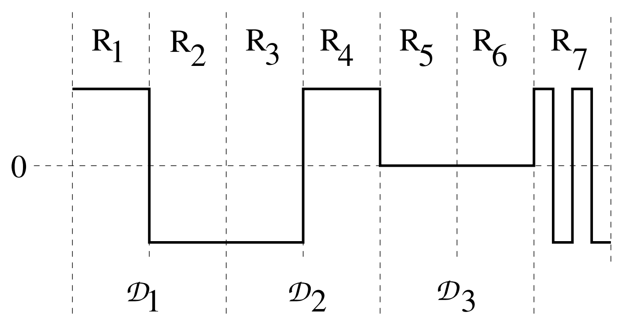


Abbildung 4.1: Das Entscheidungsgadget

Die einzige Wahl, die wir hier haben, ist die Entscheidung, welche Domain man auf R_5 und R_6 abbilden soll. Abbildung 4.2 zeigt die Attraktoren für die beiden Möglichkeiten, der Unterschied zum Originalsignal ist schraffiert dargestellt.

Das Wesentliche, was man hier sieht, ist die Tatsache, daß die Entscheidung in R_5 und R_6 auch Auswirkungen in R_7 hat! Im Attraktor wird jede Entscheidung, die man trifft, weitergereicht. Und während sich die Abbildungsfehler in R_5 und R_6 (betragsmäßig) noch nicht unterscheiden, ist der Unterschied in R_7 beträchtlich: in einem Fall wird gar kein Fehler gemacht, im anderen Fall ist er sehr groß.

Diese Methode, den Coder zu einer Entscheidung zu zwingen und diese später zu verrechnen, wird der Kern unseres Beweises zu Satz 4.9 sein. Da dieser Trick so wichtig ist, bekommt er auch einen eigenen Namen: das *Entscheidungsgadget*.

4.3.2 Beweise

Zum Beweis von Satz 4.9 geben wir im folgenden eine polynomielle Reduktion von MAXCUT auf FRACCOMP an. Genauer gesagt, konstruieren wir in polynomieller Zeit zu einem Graphen G

- ein Signal $S_G \in \mathbb{Z}^{nm}$, partitioniert in n gleichgroße Ranges, wobei n, m geeignet gewählt sind mit $n = O(|V| + |E|)$, $m = O(|V|)$, sowie
- eine Funktion $F_G : \mathbb{N} \to \mathbb{R}^+$, die streng monoton fällt,

die folgende Eigenschaft haben.

Lemma 4.11
G besitzt einen Cut der Größe $\geq k \iff$ es gibt einen Code $C \in \mathcal{C}(n)$ mit $||\Omega_C - S_G||^2 \leq F_G(k)$.

Die Signalkonstruktion wird im nächsten Abschnitt beschrieben. Die Funktion F_G wird dann gerade so definiert, daß sich die Richtung '$\Longrightarrow$' von Lemma 4.11 von selbst erfüllt.

Die wesentlich schwierigere andere Richtung wird uns während Abschnitt 4.5 beschäftigen.

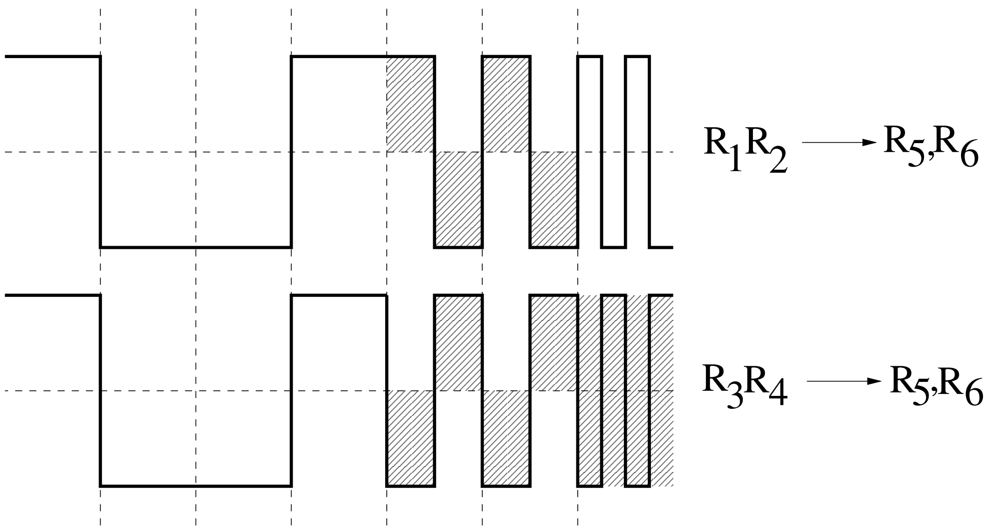


Abbildung 4.2: Attraktoren zum Entscheidungsgadget

4.4 Signalkonstruktion

4.4.1 L_1, L_2, L_3, L_4

Sei ein Graph $G = (V,E)$ gegeben. Das Signal S_G besteht im wesentlichen aus vier Teilen, *Level* genannt, die wir mit L_1, L_2, L_3, L_4 bezeichnen (siehe Abbildung 4.3). Die Zusammensetzung der Level sieht wie folgt aus:

L_1**:** Besteht aus $2|V|$ Domains, zwei für jeden Knoten des Graphen. In Abbildung 4.3 (ganz oben) sind diese beispielhaft für einen Knoten abgebildet. Die erste Range jeder Domain enthält ein besonderes Signalstück, die *ID* dieses speziellen Knotens. IDs von verschiedenen Knoten unterscheiden sich sehr stark.[5] Die zweiten Ranges enthalten komplementäre Signalstücke, *Flags* F_1 und F_2 genannt. Die Amplitude der ID beträgt a_1, die der Flags ist b_1.

L_2**:** Besteht aus $|V|$ Domains, eine für jeden Knoten. Die Hälfte der ersten Range enthält wieder die Knoten ID, auf die Hälfte ihrer ursprünglichen Breite in L_1 gestaucht (siehe Abbildung 4.3). Der Rest der Domain ist flach, d.h. gleich null. Die Amplitude der ID ist a_2.

L_3**:** Besteht aus $|E|$ Domains,[6] zwei für jede Kante des Graphen. In Abbildung 4.3 ist wieder ein Beispiel zu sehen, hier für die Kante (v,w). Das erste Viertel der ersten Range enthält die (entsprechend gestauchte) ID von v, das erste Viertel der zweiten Range enthält die ID von w. Der Rest der Domain ist gleich null.[7]

L_4**:** Besteht aus $2|E|$ Ranges, zwei für jede Kante (siehe Abbildung 4.3, ganz unten). Jede von ihnen sieht aus wie eine zusammengestauchte Kopie der Domain aus L_3 zu dieser Kante. Der einzige (aber wesentliche) Unterschied sind Kopien der Flags mit der Breite einer Achtel Range jeweils rechts neben den IDs. In einer der beiden Ranges sind dies die Flags F_1 und F_2, in der anderen F_2 und F_1 (jeweils in dieser Reihenfolge).

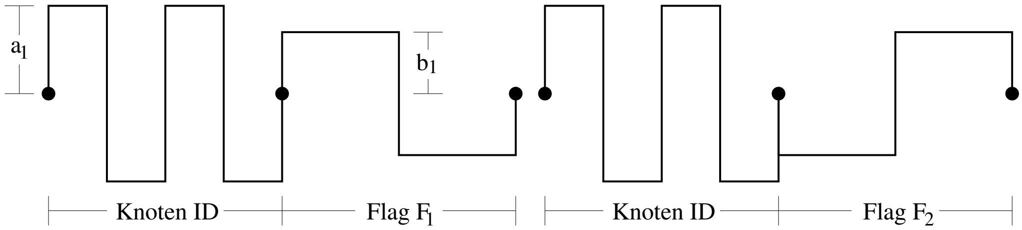
Die Amplituden a_i, b_i stehen zueinander in den Beziehungen $a_2 = \lambda_1 \cdot a_1$, $a_3 = \lambda_2 \cdot a_2$, $a_4 = \lambda_3 \cdot a_3$ und $b_i = a_i/\sqrt{2}$. Durch Wahl der Parameter λ_1, λ_2, λ_3, a_4 sind die Höhen also vollständig festgelegt. Die λ_i werden in Abschnitt 4.5 bestimmt, für uns ist hier nur wichtig, daß sie positiv und klein ($\ll 1$) sind, und a_4

[5] Genaueres zu deren Beschaffenheit in Abschnitt 4.4.3.

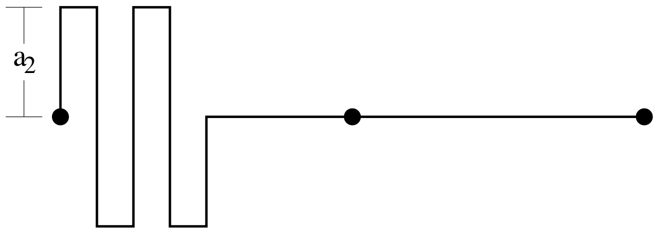
[6] $|E|$ ist die Anzahl der Kanten.

[7] Da G ungerichtet ist, muß man hier jeder Kante eine Richtung geben, um von erstem und zweiten Knoten sprechen zu können. Für den Beweis spielt es keine Rolle, wie man diese Wahl trifft. Wichtig ist nur, daß die Wahl für jede Kante konsequent durchgehalten wird, also insbesondere in L_4 wieder die gleiche Reihenfolge genommen wird.

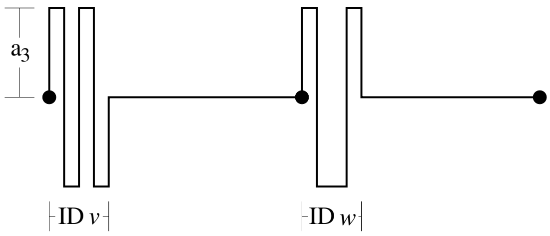
Level 1 (L_1): Zwei Domains / Knoten



Level 2 (L_2): Eine Domain / Knoten



Level 3 (L_3): Eine Domain / Kante



Level 4 (L_4): Zwei Ranges / Kante

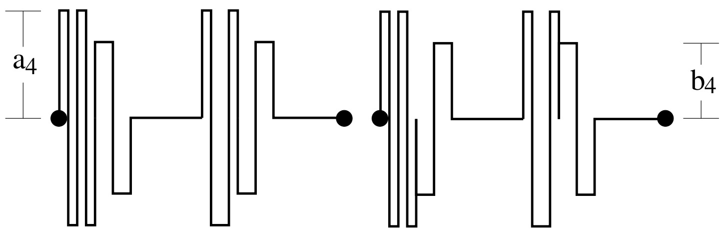


Abbildung 4.3: Bausteine des Signals S_G (Rangegrenzen sind mit schwarzen Punkten markiert)

ist ein Skalierungsparameter, von dem die gesamte Konstruktion linear abhängt. Denn man beachte, daß das soeben angegebene Signal nicht ganzzahlig ist, wie eigentlich gefordert. Dem kann man jedoch Abhilfe schaffen, indem man a_4 sehr groß wählt und dann alle Amplituden zu ganzen Zahlen rundet. Wenn a_4 nur groß genug ist, so ist der dabei entstehende Fehler für unsere Rechnungen vernachlässigbar.

4.4.2 Ein Beispiel

An einem Beispiel wollen wir jetzt die Idee hinter dieser Konstruktion erläutern. In Abbildung 4.4 ist ein Graph G und das zugehörige Signal S_G aufgezeichnet. Der Graph ist rechts unten zu sehen, er besteht nur aus zwei Knoten und einer Kante zwischen ihnen. Den beiden Knoten ist jeweils das neben ihnen gezeichnete Signalstück als ID zugeordnet. Die Flags sind in dieser Abbildung zur besseren Lesbarkeit farbig gezeichnet. Da wir hier nur etwas erklären, aber nichts beweisen wollen, ist das Signal etwas vereinfacht, und in L_4 ist nur eine Range (anstatt korrekterweise zwei) eingezeichnet.

Wie sieht der optimale Attraktor zu diesem Signal aus? Er ist in Abbildung 4.5 zu sehen. Der Fehler zum Originalsignal ist jeweils schraffiert. Der Code bildet von ‘oben’ nach ‘unten’ ab. Des weiteren bildet er IDs auf gleiche IDs ab. Im einzelnen:

Für die Ranges in L_2 eines jeden Knotens nimmt der Code eine der beiden Domains des Knotens aus L_1. Für den einen Knoten ist dies die mit dem (roten) Flag F_1, für den anderen Knoten diejenige mit (grünem) Flag F_2. Hier trifft der Attraktor sozusagen eine **Wahl**, die ganz ähnlich wie beim Entscheidungsgadget funktioniert.

Jede solche Wahl **induziert einen Cut**, wenn man die Knoten, denen das gleiche Flag zugeordnet wurde, zu einer Menge zusammenfaßt. In Abbildung 4.5 rechts unten ist er für unseren Beispielfall aufgezeichnet.

Jetzt zum zentralen Punkt dieser Konstruktion. In L_3 und L_4 wird weiter brav jede ID auf eine passende ID abgebildet, hier hat man dann gar keine Wahl mehr. Wir sehen, daß in unserem Beispiel die Range in L_4 ohne Fehler kodiert werden konnte. Erinnern wir uns, das es in L_4 für jede Kante eine Range gibt.[8] Diese Range wird ohne Fehler kodiert *genau dann, wenn* die beiden Knoten verschiedene Flags zugeordnet bekommen haben. Dies passiert *genau dann, wenn* die betreffende Kante im induzierten Cut liegt. Das liefert uns die Verbindung zwischen Attraktorfehler und Größe eines Cuts.

Das ist die Idee, die hinter dieser Konstruktion steckt. Der Rest sind technische Details, die man aber braucht, damit das ganze auch wirklich funktioniert. Wer

[8] In Wahrheit natürlich zwei, aber vergessen wir das für den Moment mal.

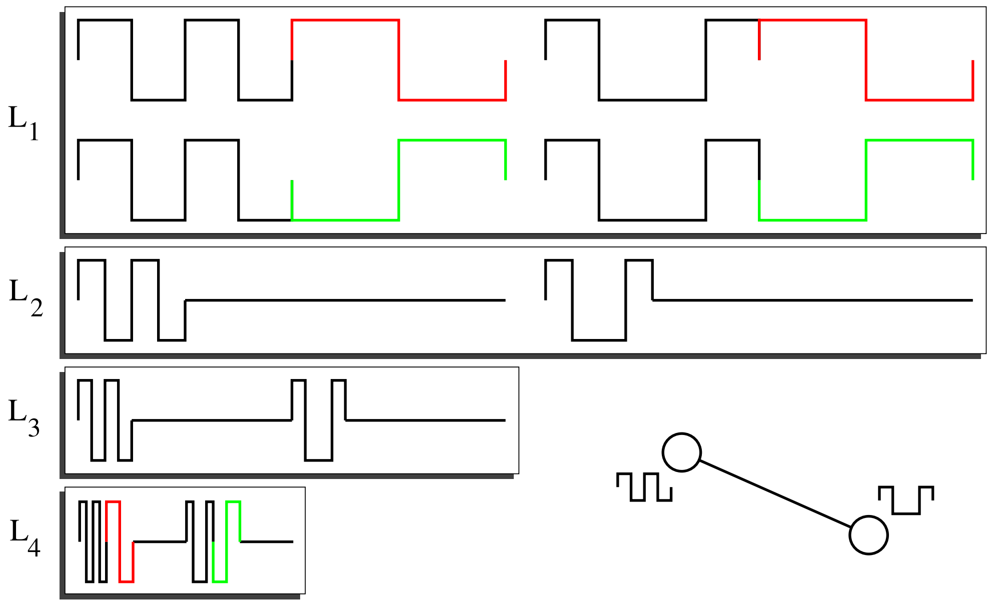


Abbildung 4.4: Ein einfacher Graph G (rechts unten) mit Signal S_G

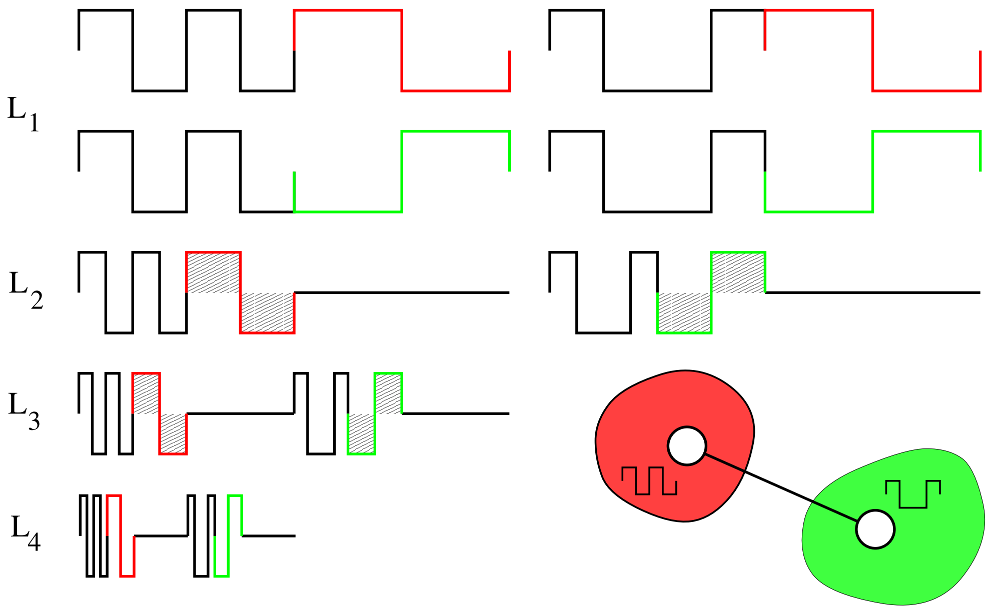


Abbildung 4.5: Der optimale Attraktor zum Signal aus Abbildung 4.4 mit induziertem Cut (rechts unten)

sich das ersparen will, kann jetzt gleich weiterblättern zu Abschnitt 4.6 auf Seite 61 (und erspart sich damit viel Arbeit). Ansonsten geht es jetzt weiter mit allen Details zum Beweis von Lemma 4.11.

4.4.3 Konstruktion der IDs

Wir wollen erreichen, daß in unserem Attraktor IDs immer nur auf gleiche IDs abgebildet werden. Dazu konstruieren wir IDs, die sehr unterschiedlich aussehen, und zwar mittels folgendem Lemma.

Lemma 4.12
Für alle $n \in \mathbb{N}$ gibt es einen binären Code aus n Codewörtern, jedes mit Länge $\ell = O(n)$, so daß für $i \neq j$ die Hammingabstände $d_H(c_i, c_j)$ und $d_H(c_i, \overline{c_j})$ gleich $\ell/2$ sind. ($\overline{c_j}$ ist das binäre Komplement von c_j.)

Beweis: Wir zeigen durch Induktion, daß das Lemma für $n = \ell = 2^m$ und alle $m \in \mathbb{N}_0$ gilt. Für alle anderen n wähle man einfach n der Codewörter, die für die Länge $2^{\lceil \log n \rceil}$ konstruiert wurden.

Zum Induktionsanfang ist $c_1 = 0$ ein solcher Code für $n = 2^0$, da er die Bedingungen trivialerweise erfüllt. Nehmen wir nun an, ein solcher Code $(c_i)_{i=1,\dots,2^m}$ wäre für $n = 2^m$ bereits konstruiert. Dann ist die Menge $\{c_ic_i, c_i\overline{c_i} | 1 \leq i \leq 2^m\}$ ein solcher Code für die Länge 2^{m+1}. Denn für $i \neq j$ gilt:

- c_ic_i und $c_i\overline{c_i}$ bzw. $\overline{c_i\overline{c_i}} = \overline{c_i}c_i$ haben trivialerweise den Hammingabstand 2^m.
- c_ic_i und c_jc_j bzw. $\overline{c_jc_j}$ unterscheiden sich in der Hälfte der Bits, da sie es nach Induktionsvoraussetzung auf beiden Hälften tun.
- Analog für c_ic_i und $c_j\overline{c_j}$ bzw. $\overline{c_j}c_j$. ■

Beispiel 4.13
Die Codes für die Längen 1,2,4,8 sehen wie folgt aus:

1	0
2	00 01
4	0000 0011 0101 0110
8	00000000 00001111 00110011 00111100 01010101 01011010 01100110 01101001

□

Sei $(c_i)_{i=1,\dots,|V|}$ ein gemäß Lemma 4.12 konstruierter Code der Länge $\ell = 2^{\lceil \log |V| \rceil}$. Dann ist $\mathfrak{C} := \{c_i\overline{c_i} | 1 \leq i \leq |V|\}$ ebenfalls ein Code mit $|V|$ Elementen, von denen sich je zwei in der Hälfte ihrer Bits unterscheiden. Außerdem $\mathfrak{C}$ erfüllt aber noch folgende 'Symmetrieeigenschaften':

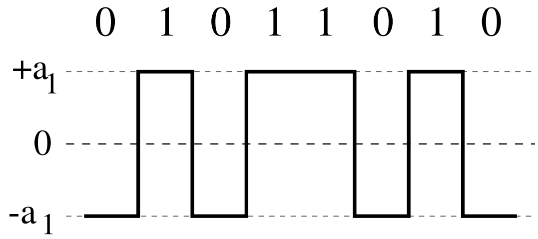


Abbildung 4.6: Umwandlung von Code in Signal.

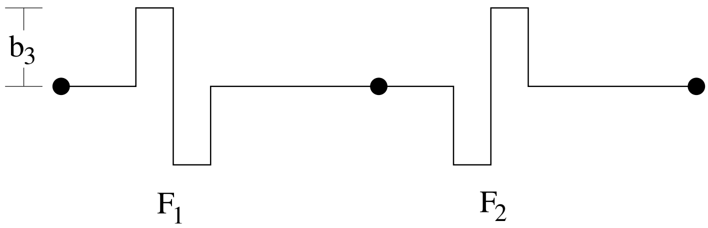


Abbildung 4.7: Extrablock in L_3

- Jedes Codewort besteht aus ℓ Nullen und ℓ Einsen.
- Für je zwei verschiedene Codewörter $p,q \in \mathfrak{C}$ gilt: für alle $i,j \in \{0,1\}$ gibt es genau $\ell/2$ Stellen, an denen p ein i-Bit und q ein j-Bit hat.

Der (einfache) Beweis sei dem Leser überlassen.

Zu den gewünschten IDs kommen wir, indem wir jedem Knoten ein Codewort aus $\mathfrak{C}$ zuordnen, und dann aus diesem ein Signalstück konstruieren. Die Nullen und Einsen werden dabei durch Amplituden $-a_1$ bzw. $+a_1$ interpretiert. Ein Beispiel ist in Abbildung 4.6 zu sehen.

Um das Signal S_G komplett zu machen, fügen wir noch Konstruktionselemente $L_{0,1},\dots,L_{0,\gamma}$ mit $\gamma = O(\log|V|)$ hinzu, um Level L_1 fehlerfrei im Attraktor erzeugen zu können. Wir setzen $L_0 := L_{0,1}\dots L_{0,\gamma}$.

Außerdem brauchen wir noch den in Abbildung 4.7 gezeigten speziellen Signalblock in Stufe L_3. Er enthält nur die Flags F_1 und F_2. Man beachte, daß man während einer Abbildung mittels eines negativen Skalierungsfaktors daraus leicht die Flags F_2 und F_1 machen kann. Wir wollen diesen Block im folgenden *Extrablock* nennen. Durch Hinzufügen entsprechender Konstruktionssegmente in den höheren Levels können wir annehmen, daß auch er fehlerfrei im Attraktor erzeugt werden kann.

Durch Zusammensetzen aller beschriebenen Signalstücke erhält man ein Signal S_G mit $O(|V|+|E|)$ Ranges aus jeweils $O(|V|)$ Elementen, und damit insgesamt ein Signal von polynomieller Länge.

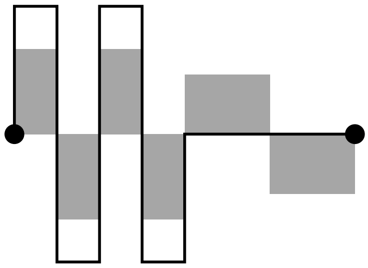


Abbildung 4.8: Signalstück von S_G (schwarze Linie) im Vergleich zum Attraktor (schraffiert)

4.4.4 Ein guter Attraktor

Kommen wir nun zum Beweis der ersten Richtung von Lemma 4.11. Für das soeben konstruierte Signal $S_G = L_0L_1L_2L_3L_4$ geben wir einen Attraktor an, den wir benutzen werden, um F_G zu definieren.

Habe $G = (V, E)$ einen Cut der Größe k, etwa mittels $V = V_1 \dot\cup V_2$.

Nach Konstruktion lassen sich L_0, L_1, und alle zu dem Extrablock führenden Teile von S_G ohne Fehler kodieren. Also tun wir dies auch. Für die restlichen Signalteile wählen wir folgende Transformationen:

L_2: Für jeden Knoten v kodieren wir die zugehörige Domain wie folgt.

Für die erste Range wählen wir eine der beiden Domains mit gleicher Knoten ID aus L_1. Und zwar, je nachdem, ob $v \in V_1$ oder $v \in V_2$ gilt, diejenige mit Flag F_1 bzw. F_2. Als Transformationsparameter wählen wir $s = \frac{2}{3}\lambda_1$ und $o = 0$. Die Höhe des Attraktorsignals ist hier also $\frac{2}{3}$ der Höhe des ursprünglichen Signals (siehe Abbildung 4.8).

Die zweite Range kodieren wir mittels $s = o = 0$.

Der Fehler,[9] den der Attraktor auf beiden Ranges zusammen macht, ist dann $\frac{1}{2}((a_2 - \frac{2}{3}a_2)^2 + (\frac{2}{3}b_2 - 0)^2) = \frac{1}{2}(\frac{1}{9}a_2^2 + \frac{4}{9}\frac{a_2^2}{2}) = \frac{1}{6}a_2^2$.

Der Gesamtfehler für den ganzen Level L_2 ist also $\frac{1}{6}|V|a_2^2$.

L_3: Hier wählen wir für jede Range die Domain aus L_2 mit passender ID. Als Skalierungsfaktor wählen wir $s = \lambda_2$ und als Offset $o = 0$. Wiederrum ist die Höhe des Attraktorsignals $\frac{2}{3}$ der Höhe des ursprünglichen Signals. Man beachte auch, daß das Flag weitergereicht wurde. Der Fehler beträgt also $\frac{1}{4}((a_3 - \frac{2}{3}a_3)^2 + (\frac{2}{3}b_3 - 0)^2) = \frac{1}{12}a_3^2$, für beide Ranges einer Kante $\frac{1}{6}a_3^2$.

Der Gesamtfehler in L_3 ist $\frac{1}{6}|E|a_3^2$.

[9] Zur Vereinfachung normieren wir die L_2-Metrik so, daß eine Range die Länge 1 hat.

L_4**:** Der Fehler in L_4 hängt von der Wahl des Cuts ab. Je nachdem, ob eine Kante im Cut liegt oder nicht, nehmen wir nämlich verschiedene Domains für sie:

- *Kante liegt im Cut.* In diesem Fall kann man eine der beiden zu der Kante gehörigen Ranges (die mit der richtigen Wahl der Flags) exakt, d.h. mit Fehler 0, kodieren. Und zwar mit der zur Kante gehörigen Domain aus L_3 (Parameter: $s = \frac{3}{2}\lambda_3, o = 0$). Die andere Range kodieren wir mit dem Extrablock ($s = \pm\lambda_3, o = 0$). Das führt zu einem Fehler von $\frac{1}{4}a_4^2$.
- *Kante liegt nicht im Cut.* In diesem Fall kodieren wir beide zur Kante gehörigen Ranges mit der zur Kante gehörigen Domain aus L_3 und den Parametern $s = \lambda_3, o = 0$. Das ergibt in jedem Fall einen Fehler von $\frac{1}{4}(a_4 - \frac{2}{3}a_4)^2 + \frac{1}{8}((b_4 - \frac{2}{3}b_4)^2 + (b_4 - (-\frac{2}{3}b_4))^2) = \frac{1}{36}a_4^2 + \frac{1}{8}b_4^2\frac{26}{9} = \frac{2}{72}a_4^2 + \frac{13}{72}a_4^2 = \frac{5}{24}a_4^2$, also für beide zusammen einen Fehler von $\frac{5}{12}a_4^2$.

Der Gesamtfehler in L_4 ist somit $k \cdot \frac{1}{4}a_4^2 + (|E| - k) \cdot \frac{5}{12}a_4^2 = (\frac{5}{12}|E| - \frac{1}{6}k)a_4^2$.

Wir definieren $F_G(k)$ als den Fehler, den der Attraktor im Falle eines Cuts von Größe k macht:

$$F_G(k) := \frac{1}{6}|V|a_2^2 + \frac{1}{6}|E|a_3^2 + (\frac{5}{12}|E| - \frac{1}{6}k)a_4^2$$

In Abhängigkeit von unseren Parametern λ_1, λ_2, λ_3, a_4 ergibt sich:

$$F_G(k) = \frac{1}{12}(\frac{2|V|}{\lambda_2^2\lambda_3^2} + \frac{2|E|}{\lambda_3^2} + (5|E| - 2k))a_4^2$$

Damit ist gleichzeitig auch die eine Richtung von Lemma 4.11 ‘bewiesen’.

4.5 Attraktoren

Die andere Richtung von Lemma 4.11 folgt aus diesem Lemma.

Lemma 4.14
Der maximale Cut von G hat Größe k $\Longrightarrow$

$$\nexists\, C \in \mathcal{C}(n):\ ||\Omega_C - S_G||^2 \leq F_G(k+1).$$

Beweis: Sei G ein Graph mit einem maximalen Cut der Größe k. Und nehmen wir einmal an, ein solcher Attraktor Ω_C mit $||\Omega_C - S_G||^2 \leq F_G(k+1)$ würde doch existieren.

Nach Abschnitt 4.4 wissen wir schon, daß es einen Attraktor Ω mit Fehler $F_G(k)$ gibt. Offensichtlich muß Ω_C das Signal besser approximieren als Ω, und zwar um mindestens $F_G(k) - F_G(k+1) = \frac{1}{6}a_4^2$ besser. Da Ω auf L_0 und L_1 keinen Fehler macht, muß dieser Gewinn auf den Segmenten L_2, L_3, L_4 erfolgen. Durch einen Schubfachschluß sehen wir, daß es zumindest auf einer dieser Stufen eine Verbesserung um wenigstens $\frac{1}{18}a_4^2$ geben muß.

Wir werden nun, abhängig allein vom Graphen G (also nicht von k!) die Parameter $\lambda_1, \lambda_2, \lambda_3$ so wählen, daß dies nicht passieren kann.

Für die folgende Diskussion nehmen wir vorübergehend an, daß die Ranges von L_2 durch Domains von L_1, die Ranges von L_3 durch Domains von L_2 und die Ranges von L_4 durch Domains von L_3 kodiert werden müssen. Am Ende werden wir dann erklären, wie man diese Einschränkung aufheben kann.

Fall 1: Ω_C *ist auf* L_2 *um* $\frac{1}{18}a_4^2$ *besser als* Ω

Nehmen wir zunächst an, daß Ω_C auf L_1 mit S_G identisch ist. Dann gibt es in C für jede Range zwei Möglichkeiten, eine Domain zu wählen:

1. Wenn wir eine Domain mit *passender ID* wählen, so ist der Attraktorfehler bei der Range, abhängig von den Transformationsparametern s und o gleich

$$E(s,o) = \frac{1}{4}\Big((a_2 - (a_1 s + o))^2 + (-a_2 - (-a_1 s + o))^2 + (b_1 s + o)^2 + (-b_1 s + o)^2\Big)$$

Die optimalen Parameter für diese quadratische Form erhält man, wenn man die partiellen Ableitung gleich null setzt:

$$\frac{\partial E}{\partial o}(s,o) = \frac{1}{2}\Big((a_2 - (a_1 s + o)) + (-a_2 - (-a_1 s + o)) + (b_1 s + o) + (-b_1 s + o)\Big) \stackrel{!}{=} 0 \implies 2o = 0 \implies o = 0$$

$$\frac{\partial E}{\partial s}(s,0) = \frac{1}{4}\Big(4(-a_2 + a_1 s)a_1 + 2(b_1 s)b_1 - 2(-b_1 s)b_1\Big) \stackrel{!}{=} 0$$

$$\implies s(2a_1^2 + b_1^2 + b_1^2) = 2a_1 a_2 \implies s(2a_1^2 + a_1^2) = 2a_1^2\lambda_1 \implies s = \frac{2}{3}\lambda_1$$

Die optimalen Werte sind also $s = \frac{2}{3}\lambda_1$ und $o = 0$, was einen Fehler von $E(\frac{2}{3}\lambda_1, 0) = \frac{1}{6}a_2^2$ ergibt.

2. Falls man eine Domain mit einer falschen ID nimmt, sieht die Fehlerfunktion wie folgt aus:

$$E(s,o) = \frac{1}{8}\Big((a_2-(a_1s+o))^2+(-a_2-(-a_1s+o))^2+(a_2-(-a_1s+o))^2+ \\ (-a_2-(a_1s+o))^2+2\cdot(b_1s+o)^2+2\cdot(-b_1s+o)^2\Big)$$

Die optimalen Parameter, die man genauso wie oben berechnen kann, sind $s=o=0$. Dies führt zu einem Fehler von $E(0,0)=\frac{1}{2}a_2^2$, dreimal so viel wie im Fall der korrekten IDs.

Der Fehler von Ω_C in L_2 ist also mindestens $(|V|+2l)\cdot\frac{1}{6}a_2^2$, wobei l die Anzahl der falsch zugeordneten IDs ist. Wir wählen λ_2 so klein, daß schon der Fehler durch eine falsch zugeordnete ID größer als der Gesamtfehler von Ω in L_3 und L_4 ist:

$$\begin{aligned} & 2\cdot\frac{1}{6}a_2^2 \overset{!}{>} \frac{1}{12}(\frac{2|E|}{\lambda_3^2}+(5|E|-2\cdot 0))a_4^2 \\ \iff \quad & \frac{1}{3}a_2^2 > \frac{1}{12}|E|(\frac{2}{\lambda_3^2}+5)a_2^2\lambda_2^2\lambda_3^2 \\ \iff \quad & 1 > \frac{1}{4}|E|(2+5\lambda_3^2)\lambda_2^2 \\ \iff \quad & \frac{2}{\sqrt{|E|(2+5\lambda_3^2)}} > \lambda_2 \end{aligned}$$

Wenn diese Bedingung erfüllt ist, muß l gleich null sein, da andernfalls der Fehler von Ω_C in L_2 allein schon größer als der Gesamtfehler von Ω wäre, im Widerspruch zur Annahme.

Nun zur Hilfsannahme, daß Ω_C auf L_1 wie S_G aussieht. Nach Voraussetzung muß der Unterschied der beiden Signale auf L_1 kleiner als $F_G(k)$ sein. Man beachte, daß $F_G(k)$ unabhängig von λ_1 ist. Indem wir λ_1 ausreichend klein wählen, können wir sicherstellen, daß der Fehler $F_G(k)$ sehr klein gegenüber a_1 ist. Dieser relative Fehler wird unsere Rechnung dann ein wenig ändern. Aber durch die Wahl von λ_1 können wir diesen Rechnungsfehler beliebig klein machen, auf jeden Fall kleiner als $\frac{1}{18}a_4^2$. Somit kann dieser erste Fall nicht eintreten.

Fall 2: Ω_C *ist auf* L_3 *um* $\frac{1}{18}a_4^2$ *besser als* Ω

Zunächst können wir wieder annehmen, daß Ω_C auf L_2 wie Ω aussieht.

Denn wie wir schon gesehen haben, bedeutet jeder wesentliche Unterschied von Ω_C und Ω auf L_2 einen zusätzlichen Fehler für Ω_C. Da Ω_C insgesamt besser als Ω sein soll, darf dieser Fehler nicht sehr groß sein. Indem wir λ_2 genügend klein wählen, spielt ein vielleicht vorhandener Unterschied nach der Transformation auf L_3 auch keine Rolle mehr für unsere Rechnungen.

Wie in Fall 1 kann man nun die verschiedenen Abbildungsvarianten (richtige ID oder falsche ID) durchspielen, mit dem gleichen Ergebnis wie dort: Im Falle der richtigen ID sind in Ω schon die optimalen Transformationsparameter gewählt, also ist keine Verbesserung möglich. Im Falle einer falschen ID macht man einen größeren Fehler, der durch geeignete Wahl von λ_3 größer als der Gesamtfehler von Ω in L_4 ist. Also kann auch Fall 2 nicht eintreten.

Fall 3: *Ω_C ist auf L_4 um $\frac{1}{18}a_4^2$ besser als Ω*

Wir können wiederrum annehmen, daß Ω und Ω_C auf L_3 ausreichend 'ähnlich aussehen', so daß der Fehler in unseren Rechnungen vernachlässigbar ist.

Wiederum untersuchen wir den Fehler, den verschiedene Domain-Range-Paarungen verursachen. Dabei unterscheiden wir zwei Fälle: eine Kante gehört zum Cut, d.h. die beiden Knoten haben verschiedene Flags, oder sie gehört nicht zum Cut.

1. *Kante gehört zum Cut.* In diesem Fall kann eine der beiden zugehörigen Ranges in L_4 ohne Fehler mit der Domain aus L_3 abgedeckt werden ($s = \frac{3}{2}\lambda_3, o = 0$). Für die andere gibt es nur die folgenden fehlerbehafteten Möglichkeiten:

   (a) *Die zugehörige Domain aus L_3.* Hier sind die IDs richtig, aber beide Flags falsch.

   (b) *Domain mit einer oder zwei falschen IDs und keinem, einem oder zwei falschen Flags.*

   (c) *Der Extrablock.*

   Durch erschöpfende Rechnung erhält man, daß die Wahl des Extrablocks ($s = \pm\lambda_3, o = 0$) den geringsten Fehler mit $\frac{1}{4}a_4^2$ macht. Das ist wieder die Wahl, die schon in Ω getroffen wurde.

2. *Kante gehört nicht zum Cut.*

   Auch hier gibt es zahlreiche Möglichkeiten, eine Domain aus L_3 zu wählen. Und viel Rechnung beweist es: die optimale Wahl ist genau die, die schon in Ω getroffen wurde und einen Fehler von $\frac{5}{12}a_4^2$ für beide Ranges zusammen verursacht.

Zusammenfassend sehen wir also, daß auch auf L_4 keine wesentliche Verbesserung möglich ist, der dritte Fall somit auch nicht eintreten kann.

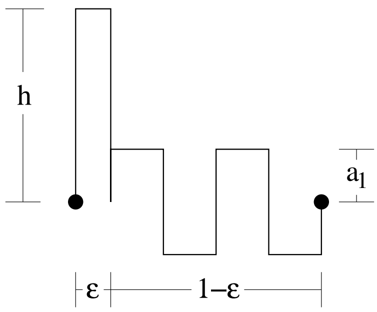


Abbildung 4.9: Signalspitze mit Höhe $h \gg a_1$ und Breite ε

Es bleibt zu zeigen, wie man erzwingen kann, daß Ranges in L_i nur durch Domains aus L_{i-1} (für $i = 2, 3, 4$) kodiert werden können. Die einfachste Möglichkeit, das zu tun, ist es, das Signal leicht abzuändern. Alle Rechnungen und Abschätzungen müssen dann leicht geändert werden, bleiben aber sinngemäß die gleichen.

Wir ändern die Knoten IDs, indem wir an den Anfang jeder ID eine Signalspitze mit Höhe $h \gg a_1$ und Breite ε setzen (siehe Abbildung 4.9). Der Rest der ID wird entsprechend auf Breite $1-\varepsilon$ gestaucht. Das führt dazu, daß wir am linken Rand der Domains in L_2, L_3, L_4 nun eine Signalspitze mit Breite $\frac{\varepsilon}{2}$, $\frac{\varepsilon}{4}$, $\frac{\varepsilon}{8}$ haben. Wenn wir h nur groß genug wählen, haben wir unser Ziel erreicht: Wenn der Code z.B. eine Domain aus L_3 auf eine Range aus L_2 abbildet, so wird eine Signalspitze der Breite $\frac{\varepsilon}{8}$ (man bedenke das Halbieren der Domainbreite) auf eine der Breite $\frac{\varepsilon}{2}$ abgebildet, was einen riesigen Fehler verursacht, ganz egal, wie der Rest von Range und Domain aussehen.

Mit dieser Überlegung ist Lemma 4.14 und damit auch Lemma 4.11 bewiesen.
■

4.6 Approximation und Collage Coder

Fraktale Bildkompression ist **NP**-hart. Aller Wahrscheinlichkeit nach gibt es also kein Programm, das ‘schnell’ einen optimalen Code für ein Signal findet. Es ist jedoch nicht ausgeschlossen, daß es Verfahren gibt, die beliebig gut an die optimale Lösung herankommen, oder sie sogar fast immer finden. Für die Praxis kann dies vollkommen ausreichend sein. In diesem Abschnitt werden wir jedoch sehen, daß die heutzutage verwendeten Programme dieses Ziel nicht erreichen. Sie können vom optimalen Ergebnis beliebig weit entfernt sein.

FRACCOMP ist ein Minimierungsproblem – es wird versucht, den Fehler möglichst klein zu machen. Man sagt, daß ein Algorithmus ein solches Problem *approximierend* löst, wenn die gelieferte Lösung höchstens um einen konstanten Faktor über der optimalen Lösung liegt. Mit dieser Notation können wir folgende Aussage formulieren.

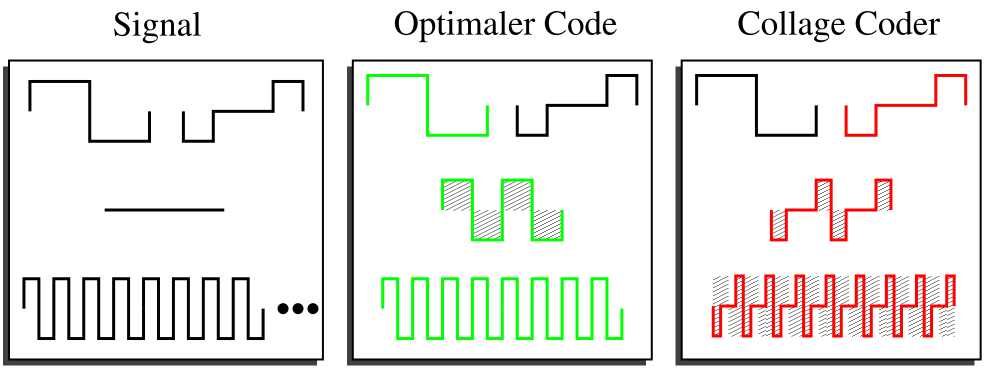


Abbildung 4.10: Signal S_Θ (links), optimaler Attraktor dafür (Mitte) und vom Collage Coder gelieferter Attraktor (rechts)

Satz 4.15
Collage Coding ist kein approximierender Algorithmus für FRACCOMP. *Denn für jedes* $\Theta > 0$ *gibt es ein Signal* S_Θ*, so daß für den optimalen Attraktor* Ω *und den vom Collage Coding gelieferten Attraktor* Ω' *gilt:*

$$||S_\Theta - \Omega'|| > \Theta \cdot ||S_\Theta - \Omega||$$

Beweis: Der Beweis benutzt letztlich wieder die Idee der Entscheidungsgadgets. Das Signal S_Θ sieht so aus wie in Abbildung 4.10 links gezeigt. Genauer erhält man es durch Zusammensetzen der beiden Domains in der obersten Zeile (Level 1), der flachen Domain in der Mitte (Level 2) und vielen Kopien der 'Schlangenlinien' ganz unten (Level 3).

Wie im Beweis zu Lemma 4.14 gesehen, können wir durch eine kleine Änderung des Signals erzwingen, daß alle Abbildungen nur von oben nach unten stattfinden können.

Es gibt demnach zwei grundsätzlich verschiedene Codes für S_Θ, je nachdem welche der beiden oberen Domains man auf die mittlere abbildet (siehe Abbildung 4.10).

- Optimal ist es, die linke Domain zu wählen. Das verursacht auf der mittleren Domain zwar einen Fehler (in Abbildung 4.10 schraffiert dargestellt), dafür kann aber der gesamte untere Level ohne Fehler kodiert werden.

- Der Collage Coder wird jedoch die rechte Domain wählen. Und zwar aus dem einfachen Grund, weil der Fehler auf dem mittleren Segment geringer ist, wie Abbildung 4.10 zeigt. Die Quittung kommt jedoch prompt, der Fehler auf dem untersten Segment ist beträchtlich. Und diesen Fehler kann man beliebig groß machen, indem man dieses untere Segment durch zahllose Wiederholungen der Schlangenlinie verlängert.

Das beweist auch schon unseren Satz. ∎

4.7 Diskussion

Was sind die Auswirkungen der in diesem Kapitel gezeigten Ergebnisse?

Zuerst sei festgestellt, daß sich die Resultate dieses Kapitels leicht auch auf höherdimensionale Signale, insbesondere also auf (zweidimensionale) Bilder, übertragen lassen.

Bedeutet der Beweis der **NP**-Härte aber nun, daß es keinen Sinn mehr macht, weiter an Verfahren zur fraktalen Bildkompression zu arbeiten, da es ja schließlich keine 'schnellen' Verfahren für optimale Kompression gibt?

Nein, natürlich nicht. Es gibt mehrere Gründe, warum fraktale Kompression weiterhin für die Praxis sinnvoll ist. Wir wollen sie kurz diskutieren.

1. *„Sie funktioniert!"* Das ist das Verblüffende. In Kapitel 3 wurde dies anhand eines Programms überzeugend dargelegt. Ein Praktiker könnte diese Diskussion damit für beendet erklären. Als Theoretiker fragen wir uns allerdings:

   Warum funktioniert fraktale Kompression in der Praxis? Schließlich haben wir eben gesehen, daß Collage Coding beliebig schlechte Codes liefern kann (siehe Abschnitt 4.6). Die Antwort könnte mit dem nächsten Punkt zusammenhängen.

2. *„Meine Bilder sehen nicht so aus!"* Wir konnten zeigen, daß es zu speziellen Signalen, eben den S_G's, sehr schwierig ist, den optimalen Code zu finden. Aber kommen solche Signale überhaupt in der Praxis vor? Wir hatten sie schließlich extra für unseren Beweis konstruiert. Es kann sein, daß reale Bilder viel einfacher zu kodieren sind als unsere künstlichen. In Abwesenheit eines guten mathematischen Modells von 'natürlichen Bildern' ist diese Aussage aber leider kaum zu präzisieren. Alle existierenden Verfahren schließen solche Signale wie die S_G's jedoch nicht explizit aus.

3. *„Wen interessiert schon ein optimaler Code?"*. Für praktische Belange ist es oft gar nicht so wichtig, *den* optimalen Code zu erzeugen. Es reicht meist, 'nicht zu weit' von Optimum entfernt zu sein. Die Frage ist also: gibt es Approximationsalgorithmen für FRACCOMP? Unser Beweis der **NP**-Härte läßt diese Möglichkeit offen. Wir wissen nur, daß Collage Coding kein solcher Algorithmus ist. Die Untersuchung der Approximationseigenschaften von fraktaler Kompression könnte also ein lohnenswertes Forschungsziel sein.

4. *„Solche Partitionierungen benutzt doch niemand!"*. Dieser und der nächste Punkt beschäftigen sich mit der Frage, ob unser Modell der fraktalen Bildkompression, das wir zur Definition von FRACCOMP herangezogen haben, vernünftig gewählt wurde. In unserem Problem ist die Partitionierung fest

vorgegeben. In der Realität ist die Wahl der Partitionierung jedoch meist auch Teil des Kodierens (siehe Kapitel 3). Das so entstehende Problem könnte theoretisch leichter sein als das von uns gestellte. Realistisch ist das jedoch nicht – das Problem wird dadurch vermutlich eher noch schwieriger.

5. *„Parameter sind in der Praxis diskret!“* Die von uns verwendeten Skalierungsfaktoren s waren kontinuierlich. Wenn man sie jedoch abspeichern oder überhaupt mit einem Computer verarbeiten will, müssen sie diskret sein. Auch die im Prinzip unbeschränkt großen Werte für o sind in dieser Hinsicht unrealistisch. Wird das Problem einfacher, wenn man nur eine feste Anzahl von Werten für s und o zuläßt?

   Nein. Bei genauer Betrachtung des Beweises fällt auf, daß lediglich fünf Werte für s verwendet wurden und daß zumindest im interessanten Teil des Signals o immer gleich 0 war. Wenn wir nur genau diese Werte zugelassen hätten wäre das Problem genau gleich schwer geblieben.

   Unschön ist dann jedoch, daß diese Werte von der Größe der Graphen, und damit von der Signallänge abhängen. Es wäre schön, die **NP**-Härte z.B. auch für den Fall zeigen zu können, bei dem s konstant gleich $\frac{3}{4}$ ist. Denn wichtige Coder, wie der von Barnsleys Firma Iterated Systems, arbeiten mit solchen festen s-Werten [Lu96].

Wo stehen wir also? Fraktale Kompression funktioniert ganz gut, und daran ändert sich auch nichts dadurch, daß sie im optimalen Fall **NP**-hart ist. Jedoch wirft dieses Resultat eine Reihe sehr interessanter und grundlegender Fragen auf. Und zeigt damit, wie wenig die theoretischen Grundlagen der fraktalen Kompression eigentlich verstanden sind.

Aber die wichtigste Lehre ist: Versucht nicht, einen fraktalen Coder zu bauen, der immer den optimalen Code findet! Lieber einen schnellen Coder, der nie weit vom Optimum entfernt ist.

Kapitel 5

PIFS und Berechenbarkeit

5.1 Einleitung

Wie ausdrucksstark sind PIFS? Wie kompliziert können durch sie kodierte Attraktoren aussehen? In diesem Kapitel wollen wir auf diese Frage eine (teilweise) Antwort geben. Im Gegensatz zur restlichen Arbeit beschäftigen wir uns dabei mit kontinuierlichen Signalen, die Trägermenge T ist eine Teilmenge des $\mathbb{R}^2$.

Als wichtigstes Resultat zeigen wir, daß PIFS so ausdrucksstark sind, daß die durch sie beschriebenen Fixpunkte im allgemeinen nicht mehr berechenbar sind.

Wie immer zuerst ein kurzer Überblick über den Aufbau des Kapitels. In Abschnitt 5.2 werden die nötigen Begriffe und Definitionen der Berechenbarkeitstheorie bereitgestellt. Diese erlauben uns, in Abschnitt 5.3 die Hauptaussage des Kapitels zu präzisieren und zu beweisen. Die abschließende Diskussion stellt die Resultate in den Zusammenhang der restlichen Arbeit.

5.2 Berechenbarkeitstheorie

Die Turingmaschine, 1936 vom britischen Mathematiker Alan Turing erfunden [Turing36], ist ein *universelles Berechnungsmodell*. Das heißt, alle Funktionen, die im intuitiven Sinne berechenbar sind, sind auch von einer Turingmaschine berechenbar.

Definition 5.1 (Turingmaschine)
Eine Turingmaschine *ist ein 4-Tupel* $M = (Z, \alpha, \omega, f)$*. Dabei sind*

- Z *eine endliche, nicht leere Menge* $\{z_1, \ldots, z_k\}$ *von* Zuständen
- α *ein Element von* Z*, der* Startzustand
- ω *ein Element von* Z*, der* Haltezustand

- *f eine Funktion von* $(Z - \{\omega\}) \times \{0,1\}$ *nach* $Z \times \{\mathbf{0}, \mathbf{1}, \mathbf{L}, \mathbf{R}\}$. □

Die Funktionsweise der Turingmaschine ist wie folgt. Die Maschine operiert mit einem Schreib-/Lesekopf auf einem in beiden Richtungen unendlich langen Arbeitsband, das mit Nullen und Einsen beschrieben werden kann. Am Beginn der Berechnung steht der Kopf am Anfang des auf das Band geschriebenen Eingabeworts[1]. Der Rest des Bandes ist mit Nullen gefüllt. Die Maschine startet im Zustand α. Wir nennen dies die *Startkonfiguration*. Allgemein ist die *Konfiguration* der Turingmaschine die Einheit von Zustand, Kopfposition und Bandinhalt – durch diese Daten wird das zukünftige Verhalten der Maschine vollständig beschrieben.

In jedem Schritt geht die Maschine abhängig vom aktuellen Zustand und dem Zeichen unter dem Lesekopf mittels f in einen neuen Zustand über (die erste Komponente des Wertes von f) und führt eine Aktion aus (die zweite Komponente des Wertes von f). Die Aktion kann sein:

- **0** = schreibe eine Null an die aktuelle Kopfposition
- **1** = schreibe eine Eins an die aktuelle Kopfposition
- **L** = bewege den Kopf eine Position nach links
- **R** = bewege den Kopf eine Position nach rechts

Falls die Turingmaschine in den Zustand ω übergeht, so endet die Berechnung, man sagt auch: die Maschine hält. Das Ergebnis der Berechnung ist dann das Zeichen (0 oder 1) unter der aktuellen Kopfposition.

Folgende Sätze aus der Berechenbarkeitstheorie werden wir verwenden. Zum Beweis vergleiche man [Schöning92].

Satz 5.2 (Halteproblem)
Es ist nicht entscheidbar (d.h. berechenbar), ob eine Turingmaschine, angesetzt auf das leere Band (überall Nullen) irgendwann hält. □

Satz 5.3 (Äquivalenzproblem)
Es ist nicht entscheidbar, ob eine Turingmaschine immer (d.h. für jeden Bandinhalt) mit Ergebnis 0 stoppt. □

Das Äquivalenzproblem ist sogar *schwerer* als das Halteproblem, da sich das Halteproblem auf das Äquivalenzproblem reduzieren läßt, jedoch nicht umgekehrt.

[1] Die Eingabe kann dabei nicht einfach binär kodiert sein, da die Maschine sonst nicht wissen würde, wo das Eingabewort aufhört. Das kann man umgehen, indem man etwa die Null als `01` und die Eins als `11` kodiert.

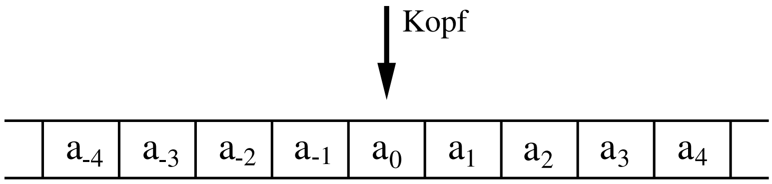


Abbildung 5.1: Der Bandinhalt der Turingmaschine wird mit a_i $(i \in \mathbb{Z})$ bezeichnet. a_0 ist immer das Zeichen, welches der Kopf gerade liest.

5.3 Fixpunkte

Das Ziel dieses Abschnitts ist der Beweis von folgendem Satz.

Satz 5.4
Gegeben sei eine Turingmaschine M. Dann können wir auf effektive Weise ein konvergentes PIFS und eine Position p seines Trägers angeben, so daß gilt: Der Wert des Fixpunktes des PIFS an Position p ist ungleich 0 genau dann, wenn die Turingmaschine angesetzt auf ein leeres Band hält.

Daraus folgt natürlich sofort durch Reduktion auf das unentscheidbare Halteproblem:

Korollar 5.5
Die Frage, welchen Wert der Fixpunkt eines PIFS an einer gewissen Position annimmt, ist im allgemeinen nicht entscheidbar. □

Beweis (Satz 5.4):
Im folgenden wollen wir den Bandinhalt einer Turingmaschine durch zwei reelle Zahlen wiedergeben. Der Bandinhalt sei etwa $I = (a_i)_{i \in \mathbb{Z}}$, $a_i \in \{0, 1\}$, wobei a_0 das Zeichen unter der aktuellen Kopfposition ist, und negative bzw. positive Indizes Zeichen links bzw. rechts des Kopfes bedeuten (siehe Abbildung 5.1). Dann setzen wir (siehe Abbildung 5.2)

- $x(I) := \sum_{i=1}^{\infty} a_{-i} \cdot 2^{-i}$ und
- $y(I) := \sum_{i=0}^{\infty} a_i \cdot 2^{-i-1}$.

$x(I)$ gibt also den Bandinhalt links der Kopfposition, $y(I)$ den Bandinhalt an der Kopfposition und rechts davon wieder. Da diese Zuordnung eineindeutig[2] ist, können wir auch einfach vom Bandinhalt (x, y) und von einer Konfiguration (x, y, z), wobei z ein Zustand ist, sprechen.

[2] Da das Band immer nur endlich viele Einsen enthalten kann, ist die Interpretation von Zahlen wie $x = \frac{1}{2} = 0.10000\ldots = 0.011111\ldots$ eindeutig.

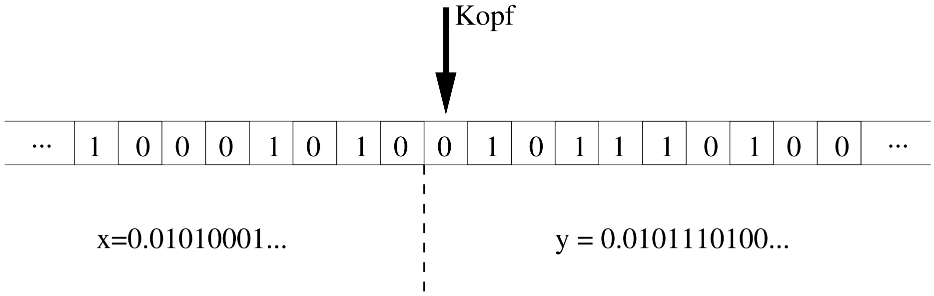


Abbildung 5.2: Bestimmung von $x(I)$ und $y(I)$ (binär)

Man beachte auch, daß die Werte von x und y beide in $[0,1[$ liegen. Denn sie können nie gleich 1 sein, da beim Start der Maschine nur endlich viele Elemente des Bandes eine 1 enthalten, und dies natürlich auch nach endlich vielen Schritten so bleibt.

Bemerkung 5.6
Zum besseren Verständnis einige triviale Aussagen über diese Kodierung. Dabei schreiben wir kurz x und y für $x(I)$ bzw. $y(I)$.

- Das höchstwertige Bit von y gibt immer das Zeichen unter der aktuellen Kopfposition wieder. Ist es Null, so gilt also $y < \frac{1}{2}$. Wenn es Eins ist gilt $y \geq \frac{1}{2}$. Der Wert des Zeichens ist somit immer gleich $\lfloor 2y \rfloor$.
- Das Schreiben einer Null an die Kopfposition entspricht also dem Übergang $y \to y \mod \frac{1}{2}$. Das Schreiben einer Eins entspricht $y \to (y \mod \frac{1}{2}) + \frac{1}{2}$.
- Was passiert mit x und y, wenn sich der Kopf eine Position nach links bewegt? Das höchstwertige Bit von x wird zum höchstwertigsten Bit vom neuen y, die restlichen Positionen rücken entsprechend nach. Formal bedeutet dies $(x,y) \to (2x \mod 1, \frac{1}{2}y + \frac{1}{2}\lfloor 2x \rfloor)$.
- Im Falle einer Bewegung nach rechts sieht es genau umgekehrt aus (man vertausche x und y). □

Zusammenfassend haben die auf dem Bandinhalt operierenden Aktionen **0, 1, L, R** also folgende Entsprechung auf dem Zahlenpaar $(x(I), y(I))$:

- **0**: $(x,y) \mapsto (x, y \mod \frac{1}{2})$
- **1**: $(x,y) \mapsto (x, (y \mod \frac{1}{2}) + \frac{1}{2})$
- **L**: $(x,y) \mapsto (2x \mod 1, \frac{1}{2}(y + \lfloor 2x \rfloor))$
- **R**: $(x,y) \mapsto (\frac{1}{2}(x + \lfloor 2y \rfloor), 2y \mod 1)$

Die Operationen können geometrisch als Strecken $(\cdot 2)$, Stauchen $(\cdot \frac{1}{2})$ und Abschneiden (mod 1 bzw. mod $\frac{1}{2}$) aufgefaßt werden. Da sich diese Operationen leicht mittels eines PIFS simulieren lassen, werden wir nun die Funktionsweise der Turingmaschine $M = (Z, \alpha, \omega, f)$ in ein PIFS abbilden.

Die Trägermenge T des PIFS bestehe aus $k := |Z|$ Kopien des Einheitsquadrates $[0, 1[^2$, die wir $Q_{z_1}, \ldots, Q_{z_k}$ nennen. Jedes dieser Quadrate besteht aus einer, zwei, drei oder vier Ranges, abhängig von f. Wir beschreiben nun die Rangeaufteilung und die zugehörigen Domains. Im Gegensatz zu den anderen Kapiteln haben die Domains diesmal *nicht* zwangsläufig die doppelte Größe der Ranges!

Die Konstruktion erfolgt so, daß ein Punkt $(x, y) \in Q_z$ im Attraktor genau dann ungleich null ist, wenn die Turingmaschine mit Bandinhalt (x, y) und in Zustand z irgendwann stoppt.

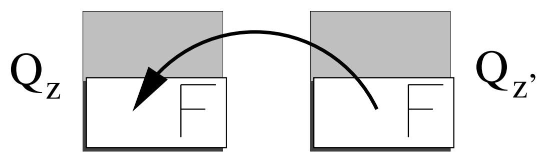
Dies kann man schon anhand der Konstruktion verfolgen, wird aber anschließend nochmal formal mittels eines Induktionschlusses gezeigt.

Die Rangeaufteilung der Q_z $(z \in Z)$ sieht wie folgt aus:

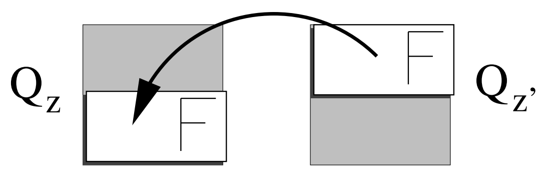
$z = \omega$: Das ist der einfachste Fall, ganz Q_ω ist eine Range. Man nehme eine beliebige Domain und Transformationsparameter $s = 0, o = 1$. Bei der Iteration des PIFS sind die Werte in Q_ω somit immer konstant gleich 1.

$z \neq \omega$: Wir geben die Rangeaufteilung und zugehörige Domains für die 'untere' Hälfte von Q_z an. Diese hängt vom Funktionswert $f(z, 0)$ ab. Die zugehörigen Transformationsparameter sind dabei immer $s = \frac{1}{2}, o = 0$.

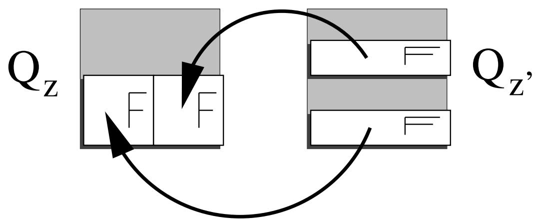
$f(z, 0) = (z', \mathbf{0})$: In diesem Fall ist die gesamte untere Hälfte von Q_z eine Range. Die zugehörige Domain ist die untere Hälfte von $Q_{z'}$, die Abbildung ist größen- und orientierungserhaltend.



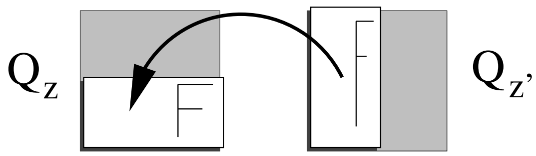
$f(z, 0) = (z', \mathbf{1})$: In diesem Fall ist wieder die ganze untere Hälfte von Q_z eine Range. Die zugehörige Domain ist diesmal die obere Hälfte von $Q_{z'}$, die Abbildung ist größen- und orientierungserhaltend.



$f(z,0) = (z', \mathbf{L})$**:** Diesmal besteht die untere Hälfte von Q_z aus zwei Ranges der Größe $[0, \frac{1}{2}[^2$. Die Domain zur linken Range ist das untere Viertel von $Q_{z'}$, in x-Richtung wird gestaucht (Faktor 2) und in y-Richtung gestreckt (gleicher Faktor). Die Domain zur rechten Range ist der Ausschnitt $[0,1[\times[\frac{1}{2}, \frac{3}{4}[$ aus $Q_{z'}$, die Abbildung ist die gleiche wie bei der anderen Range.



$f(z,0) = (z', \mathbf{R})$**:** Hier ist wieder die gesamte untere Hälfte eine Range. Die zugehörige Domain ist die linke Hälfte von $Q_{z'}$. Die Domain wird in x-Richtung mit dem Faktor 2 gestreckt und in y-Richtung mit dem gleichen Faktor gestaucht.



Die Aufteilung der oberen Hälfte von Q_z erfolgt ähnlich anhand $f(z,1)$. Die Details seien dem Leser überlassen.

Den Fixpunkt dieses PIFS erhalten wir, indem wir mit dem leeren Bild (überall gleich null) beginnen, und die Abbildungen iterieren. Hierfür gilt folgendes Lemma:

Lemma 5.7
In der n-ten Iteration I_n des PIFS hat ein Punkt $(x,y) \in Q_z$ genau dann einen Wert ungleich 0, wenn die Turingmaschine in der Konfiguration (x,y,z) in weniger als n Schritten hält.

Beweis:
Für $n = 1$ ist dies trivialerweise richtig. Denn das gesamte Q_ω ist wegen $o = 1$ konstant gleich 1. Der Wert von I_1 auf dem restlichen Träger ist wegen $o = 0$ gleich 0.

Sei also nun $n \geq 2$ und die Aussage für I_{n-1} bereits gezeigt. Sei weiterhin (x,y,z) eine beliebige Maschinenkonfiguration, bei der die Maschine in weniger als n Schritten stoppt. Wir wollen zeigen, daß I_n an der Stelle (x,y) in Q_z einen Wert ungleich null annimmt.

Im Falle $z = \omega$ ist das wieder unproblematisch, da I_n auf ganz Q_ω den Wert 1 annimmt.

Gilt $z \neq \omega$, so geht die Maschine im nächsten Schritt in einen neuen Zustand (x', y', z') über, bei dem sie in $< n-1$ Schritten stoppt. Also wissen wir nach Induktionsvoraussetzung, daß der Punkt (x', y') von $Q_{z'}$ in I_{n-1} ungleich null ist. Wir zeigen nun einfach, daß es genau dieser Punkt ist, der durch das PIFS auf (x, y) in Q_z abgebildet wird, dieser also in I_n ungleich null ist.

Dazu machen wir eine kleine Fallunterscheidung, wobei wir uns auf den Fall $y < \frac{1}{2}$ beschränken.

$f(z, 0) = (z', \mathbf{0})$**:** Die nächste Konfiguration der Turingmaschine ist also (x, y, z'). Nach Definition des PIFS wird der Punkt (x, y) von $Q_{z'}$ auf Punkt (x, y) in Q_z abgebildet, was zu zeigen war.

$f(z, 0) = (z', \mathbf{1})$**:** Wieder der gleiche Schluß. Die nächste Konfiguration ist $(x, y + \frac{1}{2}, z')$, und die Position $(x, y + \frac{1}{2})$ aus $Q_{z'}$ wird auf (x, y) in Q_z abgebildet.

$f(z, 0) = (z', \mathbf{L})$**:** Unterscheiden wir die Fälle $x < \frac{1}{2}$ und $x \geq \frac{1}{2}$, was den Ranges links unten und rechts unten in Q_z entspricht.

Für $x < \frac{1}{2}$ (links unten) ist die nächste Konfiguration gleich $(2x, \frac{1}{2}y, z')$. Und das entspricht auch dem Punkt, der im PIFS auf $(x, y) \in Q_z$ abgebildet wird.

Für $x \geq \frac{1}{2}$ (rechts unten) ist die nächste Konfiguration gleich $(2(x - \frac{1}{2}), \frac{1}{2}y + \frac{1}{2}, z')$. Wiederum ist das der Punkt, der auf $(x, y) \in Q_z$ abgebildet wird.

$f(z, 0) = (z', \mathbf{R})$**:** Die nächste Konfiguration ist $(\frac{1}{2}x, 2y, z')$. Man überzeuge sich, daß auch in diesem Fall das Gewünschte passiert, eben $(\frac{1}{2}x, 2y)$ von $Q_{z'}$ auf (x, y) in Q_z abgebildet wird.

Zusammengenommen zeigt das die Behauptung für $y < \frac{1}{2}$. Der Fall $y \geq \frac{1}{2}$ verläuft analog. ■

Da die Fixpunktbildung monoton wachsend ist ($0 \equiv I_0 \leq I_1 \leq I_2 \leq \ldots$), existiert der Fixpunkt des PIFS und ist gleich $\sup_n I_n$. Und nach Konstruktion (und Lemma 5.7) ist er an einer Position (x, y) in Q_z genau dann ungleich null, wenn die Maschine in Konfiguration (x, y, z) irgendwann stoppt. Folglich gibt der Wert des Fixpunktes an der Stelle $(0, 0)$ in Q_α an, ob die Turingmaschine angesetzt auf ein leeres Band hält. Damit ist unser Satz bewiesen. ■

Wir wissen also, daß es im allgemeinen unmöglich ist, festzustellen, ob ein bestimmter Punkt des Attraktors eines PIFS ungleich null ist. Nun könnte man

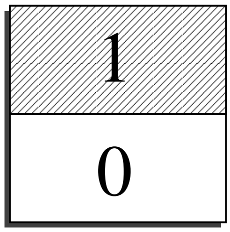


Abbildung 5.3: Q_ω für Satz 5.8

meinen, daß das gar nicht so wichtig ist. Wen interessieren schließlich schon einzelne Punkte? Viel interessanter könnte die Frage sein, ob sich in einem *Gebiet* (etwa einem abgeschlossenen Intervall) des Trägers ein Punkt befindet, der im Attraktor einen Wert ungleich null annimmt. Verblüffenderweise ist diese Frage jedoch noch schwerer zu beantworten:

Satz 5.8
Gegeben ein PIFS und eine kompakte Teilmenge T_0 seines Trägers[3]*, ist es im allgemeinen unentscheidbar, ob es einen Punkt in T_0 gibt, für den der Attraktor einen Wert ungleich 0 annimmt. Dieses Problem ist mindestens so schwer wie das Äquivalenzproblem.*

Beweis: Das ergibt sich ganz leicht aus dem Beweis zu Satz 5.4. Denn das Äquivalenzproblem ist gleichschwer wie die Frage, ob eine Maschine, falls sie hält, immer mit Ausgabe 0 hält. Wir wollen dies das ‘modifizierte Äquivalenzproblem’ nennen.

Zum Beweis ändern wir die obige Konstruktion so, daß Q_ω wie in Abbildung 5.3 aussieht: die obere Hälfte ist konstant 1, die untere konstant 0. Die restlichen Abbildungen bleiben gleich.

Das führt dazu, daß im Attraktor genau die Punkte einen Wert $\neq 0$ haben, bei denen die Maschine irgendwann mit Ausgabe 1 hält. Um das modifizierte Äquivalenzproblem zu beantworten, muß somit nur geprüft werden, ob irgendein Punkt in Q_α ungleich 0 ist. Diese Frage ist demnach mindestens so schwer wie das Äquivalenzproblem. Mit der Wahl $T_0 = Q_\alpha$ ist die Behauptung also gezeigt. ■

5.4 Diskussion

Dies sollte nur eine Andeutung von grundlegenden berechenbarkeitstheoretischen Fragen sein, die man im Zusammenhang mit fraktaler Kompression stellen kann.

[3]Kompaktheit ist hier bezüglich des Trägers gemeint.

Sie bedeuten insofern kein Problem für die Praxis, als man dort ja mit diskreten Trägern operiert, und für diese immer ein Berechnen des Fixpunktes in polynomieller Zeit möglich ist.

Der gezeigte Satz gibt aber einen Einblick in die Beschaffenheit der Fixpunkte im kontinuierlichen Fall. Das kann man als Anregung nehmen, die Beschaffenheit der Fixpunkte im diskreten Fall genauer zu betrachten. Dies scheint sinnvoll zu sein, um fraktale Kompression besser zu verstehen und Aussagen wie die im letzten Kapitel gezeigte **NP**-Härte zu präzisieren oder zu verschärfen.

Anhang

Anhang A

Manual Pages

A.1 Kodierer FRAP

NAME

frap - encode an image using **fr**actal compression with **a**daptive **p**artitionings

SYNOPSIS

frap [-f input file name] [-c output file name] [-r compression rate] [-t tolerance criterion] [-s bits for scaling parameter] [-o bits for offset parameter] [-a atomic block size] [-e epsilon for nn-search] [-hv]

DESCRIPTION

Frap compresses images using a fractal coding technique with highly adaptive range partitionings, as described in [1]. Initially, the image is partitioned into small atomic blocks of equal size, for which a fractal code is computed. Then, iteratively, selected neighboring range pairs are merged. This yields a sequence of partitions with a decreasing number of ranges. The iteration is stopped when either the desired compression ratio has been reached, or the image quality has fallen below a given threshold, whichever occurs first. The resulting code can be decompressed using the accompanying program *defrap*.

Input images must be stored in PPM format, and can be grayscale or color.

Various parameters modify the compression algorithm and control the desired compression ratio and/or image fidelity. The -f and -c flag determine input and output file names. The -r and -t flag can be used to select the desired compression rate, and image fidelity, respectively.

The flags -s,-o,-a,-e control the behavior of the compression algorithm, and can be left at default settings for most practical applications.

OPTIONS

-f *input file name*. **Required.** Selects the name of the input file, which must be stored in raw PPM-format, and must be either grayscale or color.

-c *output file name*. Defaults to `frap.out`. Selects the filename to which the resulting code is written.

-r *compression rate*. Defaults to `50`. Selects the compression rate, at which the encoder stops, and outputs the code. If a value ≤ 0 is supplied, this criterion is turned off. If `-t` is supplied also, the coder stops when the first of the two criteria is met. If none of them is turned on, the encoder stops when there less than 100 ranges left.

-t *tolerance criterion*. Defaults to `-1`. Selects the collage error (measured in PSNR), at which the encoder stops and outputs the code. If a value ≤ 0 is supplied, this criterion is turned off. If `-r` is supplied also, the coder stops when the first of the two criteria is met. If none of them is turned on, the encoder stops when there less than 100 ranges left.

-s *bits for scaling parameter*. Defaults to `5`. Selects the number of bits used for quantizing the scaling parameters in the affine transformations between domains and ranges.

-o *bits for offset parameter*. Defaults to `7`. Selects the number of bits used for quantizing the offset parameters in the affine transformations between domains and ranges.

-a *atomic block size*. Defaults to `4`. Selects the size of the atomic blocks, into which the image is partitioned initially.

-e *epsilon for nn-search*. Defaults to `10`. Selects the epsilon for the nearest neighbor search during the initialization phase (see [1]). A value < 0 turns off nn-search and uses full search, which makes the program run much slower.

-h Output help information.

-v Output program version.

FILES

Default output file is **frap.out**. The output files can be decoded with *defrap*.

REFERENCES

[1] Matthias Ruhl, Fraktale Bildkompression: Adaptive Partitionierungen und Komplexität, Diploma Thesis, Universität Freiburg, April 1997.

BUGS

Still too slow.

A.2 Dekodierer DEFRAP

NAME

defrap - **de**code an image encoded by fractal compressor ***frap*** and output it in PPM-format

SYNOPSIS

defrap [-f input file name] [-o output file name] [-i iterations] [-hv]

DESCRIPTION

Defrap decodes an image compressed with the fractal coder *frap*. This is done by repeatedly applying a contractive transformation as described by the code to an initially empty image.

The resulting image is output in PPM-format as either a grayscale or color image, depending on the format of the compressed image.

The -f and -c flag determine input and output file names. The -i flag can be used to set the number of iterations during the decoding process.

OPTIONS

-f *input file name*. Defaults to `frap.out`. Selects the name of the input file, which must be produced by *frap*.

-o *output file name*. Defaults to `frap.ppm`. Selects the filename to which the decoded image is written to.

-i *iterations*. Defaults to `10`. Select how often the contractive operation given by the fractal code is applied to the empty image to finally yield the decoded image. This normally does not need to be changed. Setting it too low may result in incomplete image reconstruction. The decoder issues a warning if that is the case.

-h Output help information.

-v Output program version.

FILES

Default input file is **frap.out**, default output file is **frap.ppm**. The program *frap* is used to encode images to be decoded by *defrap*.

BUGS

None known.

Anhang B

Ergebnisse

In diesem Kapitel sind einige Testdaten zum in Kapitel 3 beschriebenen Programm zusammengefaßt. Soweit entsprechende Daten vorhanden waren, sind jeweils die Ergebnisse folgender Verfahren verglichen worden:

Frap: Unser Programm, gestartet mit den Standardeinstellungen. Insbesondere war die Größe der atomaren Blöcke also 4×4. Durch Variieren dieser Größe lassen sich die Ergebnisse noch verbessern (Abbildung 3.13). Die Laufzeiten lagen zwischen 2 und 5 Minuten.

Quadtree: Fishers Programm `enc` aus [Fisher94b]. Dieses ist als Sourcecode verfügbar[1]. Die gewählten Parameter waren `-M 7 -h 512 -f -D 2 -d 4` und variierender Parameter `-t`. Die Laufzeit lag zwischen 5 und 60 Minuten.

HV: Die HV-Implementierung aus [FiMe94]. Dieses Programm ist leider nicht erhältlich, also wurden die in [Fisher94a], Anhang D, gezeigten Ergebnisse als Vergleich herangezogen. Die Laufzeit betrug Stunden oder gar Tage.

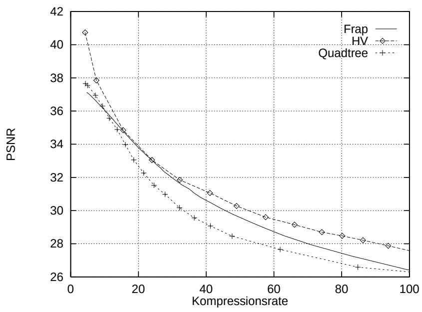
Alle verwendeten Bilder haben das Format 512×512 und stammen von einem Server der Universität von Waterloo[2]. Die einzelnen Dateinamen dort sind:

Lenna (Grauwert)	`lena2.pgm`
Boat	`boat.pgm`
Mandrill	`mandrill.pgm`
Barbara	`barb.pgm`
Lenna (Farbe)	`lena3.ppm`
Peppers (Farbe)	`peppers3.ppm`

[1] `http://inls3.ucsd.edu/y/Fractals/enc.c`

[2] `http://links.uwaterloo.ca/BragZone/Collected/PGM/`

(a) Original Lenna



(b) PSNR-Kurven zu Lenna

(c) PSNR 35,1 db, Kompr. 12,84:1

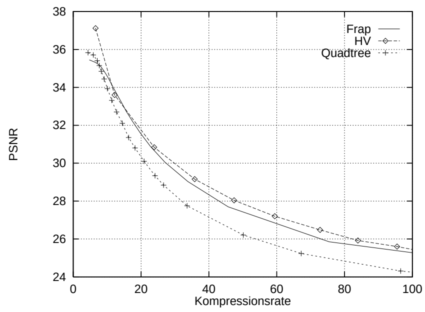
(d) PSNR 32,6 db, Kompr. 26,57:1

(e) PSNR 30,1 db, Kompr. 44,25:1

(f) PSNR 27,9 db, Kompr. 70,98:1

Abbildung B.1: Ergebnisse für Lenna

(a) Original Boat



(b) PSNR-Kurven zu Boat

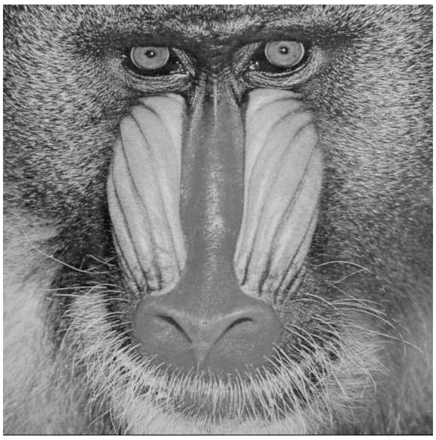
(c) PSNR 33,6 db, Kompr. 13,00:1

(d) PSNR 30,0 db, Kompr. 27,05:1

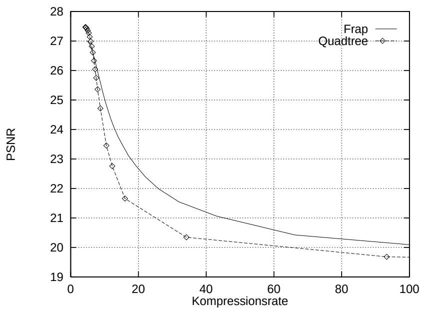
(e) PSNR 27,7 db, Kompr. 45,70:1

(f) PSNR 25,9 db, Kompr. 75,46:1

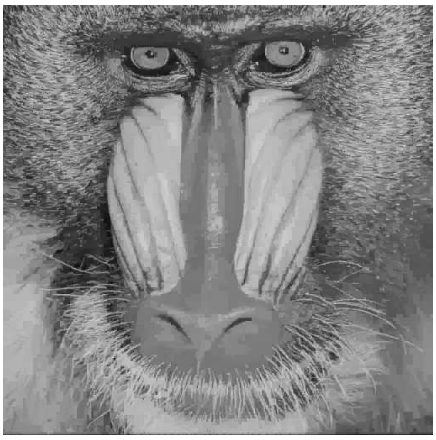
Abbildung B.2: Ergebnisse für Boat



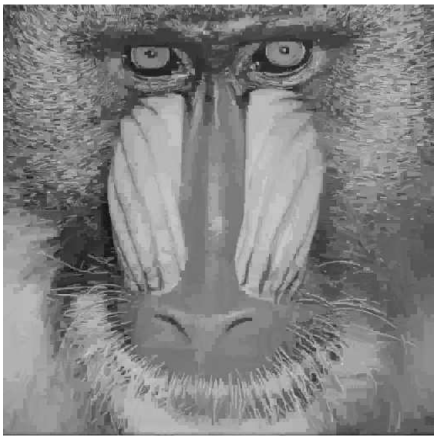
(a) Original Mandrill



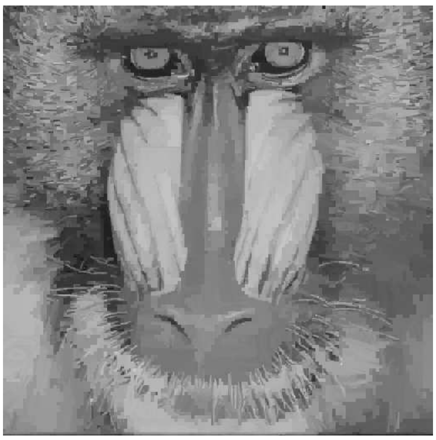
(b) PSNR-Kurven zu Mandrill



(c) PSNR 24,1 db, Kompr. 12,82:1



(d) PSNR 22,0 db, Kompr. 25,91:1

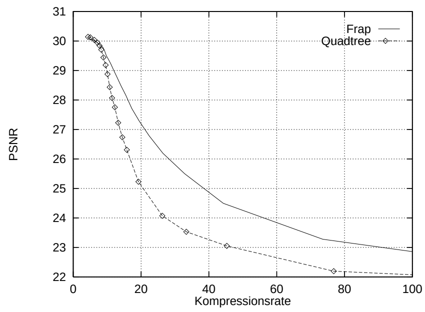


(e) PSNR 21,1 db, Kompr. 43,10:1

(f) PSNR 20,4 db, Kompr. 66,18:1

Abbildung B.3: Ergebnisse für Mandrill

(a) Original Barbara



(b) PSNR-Kurven zu Barbara

(c) PSNR 28,8 db, Kompr. 12,90:1

(d) PSNR 26,2 db, Kompr. 26,45:1

(e) PSNR 24,5 db, Kompr. 44,24:1

(f) PSNR 23,3 db, Kompr. 73,55:1

Abbildung B.4: Ergebnisse für Barbara

(a) PSNR 33,1 db, Kompr. 23,77:1

(b) PSNR 33,1 db, Kompr. 18,61:1

(c) PSNR 30,1 db, Kompr. 44,25:1

(d) PSNR 30,2 db, Kompr. 32,14:1

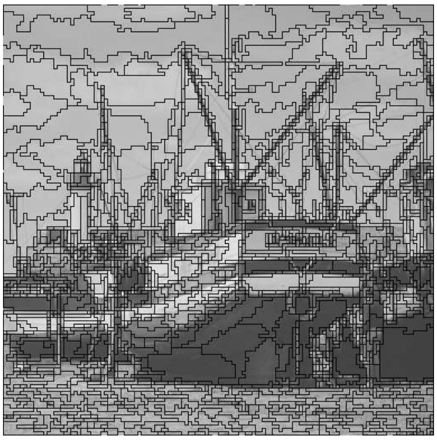
(e) PSNR 28,5 db, Kompr. 63,03:1

(f) PSNR 28,5 db, Kompr. 47,66:1

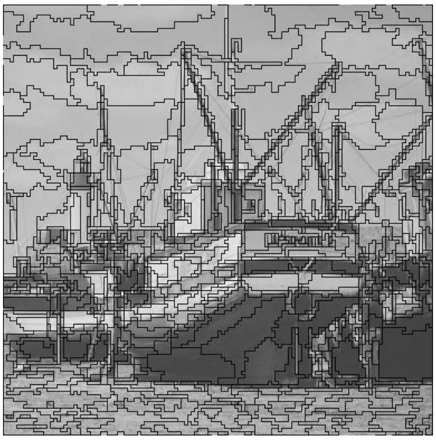
Abbildung B.5: Vergleich visueller Eindruck. Links: unser Verfahren, rechts: Quadtree

(a) Lenna, 2000 Ranges

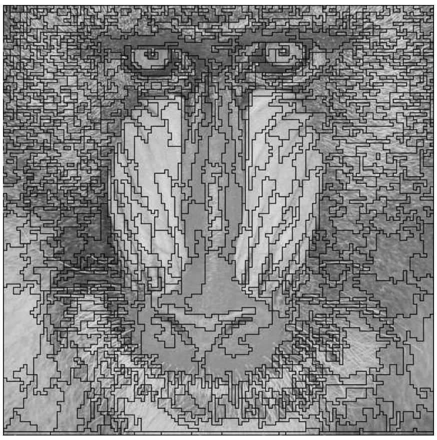
(b) Lenna, 1000 Ranges



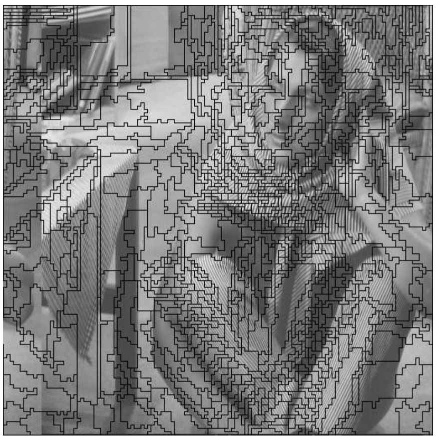
(c) Boat, 2000 Ranges



(d) Boat, 1000 Ranges



(e) Mandrill, 1000 Ranges



(f) Barbara, 1000 Ranges

Abbildung B.6: Beispiele für Partitionierungen

(a) Original Lenna

(b) Original Peppers

(c) 3000 Ranges, Kompr. 42,63:1

(d) 3000 Ranges, Kompr. 42,11:1

(e) 1000 Ranges, Kompr. 103,56:1

(f) 1000 Ranges, Kompr. 104,01:1

Abbildung B.7: Ergebnisse für Farbbilder

Literaturverzeichnis

[AMNSW94] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Wu, *An Optimal Algorithm for Approximate Nearest Neighbor Searching*, Proceedings 5th Annual ACM-SIAM Symposium on Discrete Algorithms, 573–582, 1994.

[BaVoNo93] K.-U. Barthel, T. Voyé, P. Noll, *Improved Fractal Image Coding*, Proceedings Picture Coding Symposium, März 1993.

[BaSl87] M. F. Barnsley, A. D. Sloan, *Chaotic Compression*, Computer Graphics World, November 1987.

[BaSl88] M. F. Barnsley, A. D. Sloan, *A better way to compress images*, BYTE Magazine, Januar 1988.

[BaSl90] M. F. Barnsley, A. D. Sloan, *Methods and apparatus for image compression by iterated function system*, U.S. Patent No. 4.941.193, Juli 1990.

[BeDeKe92] T. Bedford, F. M. Dekking, M. S. Keane, *Fractal image coding techniques and contraction operators*, Nieuw Arch. Wisk. (4) 10,3, 185–218, 1992.

[CrKa95] P. Crescenzi, V. Kann, *A compendium of NP optimization problems*, Technical report SI/RR-95/02, Universita di Roma "La Sapienza", 1995.

[DACB96] F. Davoine, M. Antonini, J.-M. Chassery, M. Barlaud, *Fractal image compression based on Delaunay triangulation and vector quantization*, IEEE Transactions on Image Processing 5,2, 338–346, 1996.

[DaHa97] I. M. Danciu, J. C. Hart, *Fractal Color Compression in the $L^*a^*b^*$ Uniform Color Space*, Abstract in: Proceedings DCC'97 Data Compression Conference, James A. Storer, Martin Cohn (Hrsg.), IEEE Computer Society Press, März 1997.

[EdKo85] M. Eden, M. Kocher, *On the performance of contour coding algorithms in the context of image coding, Part 1: Contour segment coding*, EURASIP, Signal Processing, 8, 381–386, 1985.

[FiMe94] Y. Fisher, S. Menlove, *Fractal Encoding with HV Partitions*, in [Fisher94a].

[Fisher94a] Y. Fisher (Hrsg.), *Fractal Image Compression: Theory and Application*, Springer, New York, 1994.

[Fisher94b] Y. Fisher, *Fractal Image Compression with Quadtrees*, in [Fisher94a].

[Freeman61] H. Freeman, *On the coding of arbitrary geometric configurations*, IRE Trans. Electronic Comp., EC(10), 260–268, Juni 1961.

[GaJo79] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[HaSaBa97] H. Hartenstein, D. Saupe, K.-U. Barthel, *VQ-Encoding of luminance parameters in fractal coding schemes*, Proceedings ICASSP'97, München, April 1997.

[JaFiBo92] E. W. Jacobs, Y. Fisher, R. D. Boss, *Image compression: A study of the iterated transform method*, Signal Processing 29, 251–263, 1992.

[Jaquin89] A. E. Jaquin, *A fractal theory of iterated Markov operators with applications to digital image coding*, Dissertation, Georgia Tech, Atlanta, 1989.

[Karp72] R. M. Karp, *Reducibility among combinatorial problems*, in: R. E. Miller, J. W. Thatcher (Hrsg.), *Complexity of Computer Computations*, 85–103, 1972.

[Leonardi87] R. Leonardi, *Segmentation adaptive pour le codage d'images*, Dissertation, EPFL, Lausanne, 1987.

[LeZi77] J. Ziv, A. Lempel, *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, Vol. 23, No. 3, 337–343, 1977.

[Lu96] N. Lu, *Fractal Image Compression*, in: Lectures on Fractal Techniques in Image Compression, ImageTech, Atlanta, März 1996.

[Nova93] M. Novak, *Attractor coding of images*, Dissertation, Department of Electrical Engineering, Linköping Universität, Mai 1993.

[Papa94] C. H. Papadimitriou, *Computational Complexity*, Addison Wesley, 1994.

[Reus94] E. Reusens, *Partitioning complexity issue for iterated function systems based image coding*, in: Proceedings of the VIIth European Signal Processing Conference EUSIPCO'94, Edinburgh, September 1994.

[RuHa97] M. Ruhl, H. Hartenstein, *Optimal Fractal Coding is NP-Hard*, Proceedings DCC'97 Data Compression Conference, James A. Storer, Martin Cohn (Hrsg.), IEEE Computer Society Press, März 1997.

[RuHaSa97] M. Ruhl, H. Hartenstein, D. Saupe, *Adaptive partitionings in fractal image compression*, Manuskript, 1997.

[SaHaHa96] D. Saupe, R. Hamzaoui, H. Hartenstein, *Fractal image compression: an introductory overview*, in: Dietmar Saupe, John Hart (Hrsg.), Fractal Models for Image Synthesis, Encoding and Analysis, SIGGRAPH'96 Course Notes XX, New Orleans, 1996.

[SaPe96] A. Said, W. A. Pearlman, *A new, fast, and efficient image codec based on set partitioning in hierarchical trees*, IEEE Transactions on Circuits and Systems for Video Technology, 6(3), 243–250, Juni 1996.

[SaRu96] D. Saupe, M. Ruhl, *Evolutionary fractal image compression*, ICIP Proceedings, September 1996.

[Saupe95] D. Saupe, *Acceleration fractal image compression by multi-dimensional nearest neighbor search*, Proceedings DCC'95 Data Compression Conference, James A. Storer, Martin Cohn (Hrsg.), IEEE Computer Society Press, März 1995.

[Saupe96] D. Saupe, *Lean domain pools for fractal image compression*, Proceedings IS&T/SPIE 1996 Symposium on Electronic Imaging: Science & Technology - Still Image Compression II, Vol. 2669, Januar 1996.

[Schöning92] U. Schöning, *Theoretische Informatik – Kurz gefaßt*, BI Wissenschaftsverlag, 1992.

[Shapiro93] J. M. Shapiro, *Embedded image coding using zerotrees of wavelet coefficients*, IEEE Transactions on Signal Processing, 41(12), 3445–3462, Dezember 1993.

[ThDe95] L. Thomas, F. Deravi, *Region-Based fractal image Compression using heuristic search*, IEEE Transactions on Image Processing, 4,6, 832–838, 1995.

[Turing36] A. M. Turing, *On computable numbers, with an application to the Entscheidungsproblem*, Proc. London Math. Society Ser. 2, Vol 42, 230–265, Vol 43, 544–546, 1936.