Universidade Federal de Pernambuco

Graduação em Engenharia da Computação

Centro de Informática

2009.2

QUANTIZAÇÃO VETORIAL MAIS EFICIENTE NA COMPRESSÃO DE TEXTURAS COM IPACKMAN

**Trabalho de Graduação**

Recife, 30 de novembro de 2009

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

**RAFAEL BORBA COSTA DOS SANTOS**

“QUANTIZAÇÃO VETORIAL MAIS EFICIENTE NA COMPRESSÃO DE TEXTURAS COM IPACKMAN”

*Trabalho apresentado no curso de graduação em Engenharia da Computação do Centro de Informática - Universidade Federal de Pernambuco - como requisito parcial para obtenção do grau de Engenheiro da Computação, sob a orientação do professor* ***Marcelo Walter*** *e co-orientação do professor* ***Sílvio Melo****.*

*“Obstáculo é aquilo que você enxerga quando tira os olhos do seu objetivo”*

**Henry Ford**

**Resumo**

Este trabalho apresenta modificações na utilização do algoritmo *iPACKMAN* [1] que melhoram significativamente seu desempenho na compressão de imagens e texturas em dispositivos móveis. Tais modificações são fruto de uma pesquisa de otimização da quantização vetorial do algoritmo. *iPACKMAN* utiliza um único conjunto de tabelas (*codebook*) fixo para sintetizar o valor de cada pixel em qualquer imagem. Mostramos que é possível melhorar a qualidade da compressão particularizando algumas tabelas otimizadas para grupos de imagens distintas.

Numa abordagem anterior, apresentamos alterações no *iPACKMAN* para simplificar ainda mais sua implementação (reduzindo o custo computacional) sem perda de qualidade na compressão de texturas [2]. Neste trabalho, propomos uma otimização do processo de quantização vetorial, o que melhora a qualidade de compressão para todas as imagens. Os ganhos de qualidade com a nova proposta variam conforme a classe de imagens atingindo, em média, 0.14 dB. Para tanto, utilizamos um *codebook* para cada classe de imagens em detrimento de um *codebook* único e universal.

**Palavras-chave**: Compressão de Imagens, compressão de texturas, quantização vetorial, LBG, otimização, *PACKMAN*, *iPACKMAN*

**Sumário**

[Introdução 8](#_Toc247365707)

[1.1 Motivação 8](#_Toc247365708)

[1.2 Contexto 8](#_Toc247365709)

[1.3 Objetivo e metodologia 9](#_Toc247365710)

[1.4 Estrutura do trabalho 10](#_Toc247365711)

[Compressão de imagens e texturas 12](#_Toc247365712)

[2.1 Compressão de imagens versus compressão de texturas 12](#_Toc247365713)

[2.1 Esquemas de compressão de texturas 14](#_Toc247365714)

[2.1.1 Block Truncation Coding 14](#_Toc247365715)

[2.1.2 S3TC 15](#_Toc247365716)

[2.1.2 Vector Quantization (Quantização Vetorial) 16](#_Toc247365717)

[Ericsson Texture Compression (ETC) 18](#_Toc247365718)

[3.1 PACKMAN 18](#_Toc247365719)

[3.1.1 Quantização vetorial no PACKMAN 20](#_Toc247365720)

[3.2 iPACKMAN 22](#_Toc247365721)

[3.2.1 Descompressão direta com iPACKMAN 25](#_Toc247365722)

[Classificação de imagens 27](#_Toc247365723)

[4.1 As classes 28](#_Toc247365724)

[Experimentos 36](#_Toc247365725)

[5.1 Métricas de erro 36](#_Toc247365726)

[5.2 A quantização vetorial (LBG adaptado) 37](#_Toc247365727)

[5.2.1 Descrição em linguagem natural do quantizador 38](#_Toc247365728)

[5.3 Testes e resultados 40](#_Toc247365729)

[5.4 Otimizando *codebooks* para classes de imagens distintas 42](#_Toc247365730)

[5.5 Testando os novos *codebooks* 45](#_Toc247365731)

[Conclusões 52](#_Toc247365732)

[6.1 Validação do *codebook* ótimo 52](#_Toc247365733)

[6.2 Viabilidade da proposta de classificação das imagens 52](#_Toc247365734)

[6.3 Trabalhos futuros 53](#_Toc247365735)

[Referências bibliográficas 54](#_Toc247365736)

**Lista de figuras**

[Figura 2.1] *Variable Bit Length Coding*

[Figura 2.2] *Block Truncation Coding*

[Figura 2.3] *Vector Quantization*

[Figura 3.1] Separação das informações de cor em *iPACKMAN*

[Figura 3.2] *Benchmark* original - *iPACKMAN*

[Figura 3.3] Histograma - *iPACKMAN*

[Figura 3.3] Compressão - *iPACKMAN*

[Figura 3.4] Comparação visual - *PACKMAN* x *iPACKMAN*

[Figura 3.5] Descompressão simples do *iPACKMAN*

[Figura 4.1] Árvore de classificação de imagens

[Figura 4.2] *Benchmark* para a classe: “Paisagens” em “Naturais”

[Figura 4.3] *Benchmark* para a classe “Humanos” em “Naturais”.

[Figura 4.4] *Benchmark* para a classe “Outros” em “Naturais”.

[Figura 4.5] *Benchmark* para a classe “Textos” em “Artificiais”.

[Figura 4.6] *Benchmark* para a classe “Texturas” em “Artificiais”.

[Figura 4.7] *Benchmark* para a classe “Computação Gráfica” em “Artificiais”.

[Figura 5.1] *Root mean square error*

[Figura 5.2] *Peak signal to noise ratio*

[Figura 5.1] Teste com imagem *Lorikeet*

[Figura 5.2] Detalhe do teste com imagem *Lorikeet*

[Figura 5.3] Testando a imagem *Lena*

[Figura 5.4] Uma textura cujo erro foi minimizado com outro *codebook*

[Figura 5.5] Testando uma imagem da classe “Outras” em “Artificiais”

[Figura 5.6] Imagem da classe “Textos”

[Figura 5.7] Imagem da classe “Paisagens”

[Figura 5.8] Imagem de teste da classe “Computação Gráfica”

[Figura 5.9] Outra Imagem da classe “Computação Gráfica”

[Figura 5.10] Imagem da classe “Outras”

**Lista de tabelas**

[Tabela 3.1] *Codebook* utilizada pelo algoritmo *PACKMAN*.

[Tabela 3.2] *Codebook* do *iPACKMAN*

[Tabela 5.1] *Codebook* obtido pelo quantizador – primeira execução

[Tabela 5.2] *Codebook* obtido pelo quantizador – segunda execução

[Tabela 5.3] *Codebook* obtido pelo quantizador – terceira execução

[Tabela 5.4] Melhor *codebook* para o *benchmark* da classe “Paisagens”

[Tabela 5.5] Melhor *codebook* para o *benchmark* da classe “Humanos”

[Tabela 5.6] Melhor *codebook* para o *benchmark* da classe “Outras”

[Tabela 5.7] Melhor *codebook* para o *benchmark* da classe “Textos”

[Tabela 5.8] Melhor *codebook* para o *benchmark* da classe “Texturas”

[Tabela 5.9] Melhor *codebook* para o *benchmark* da classe “Computação gráfica”

CAPÍTULO 1

Introdução

# 1.1 Motivação

Aplicações de computação gráfica, jogos e interfaces gráficas demandam muitas imagens de alta resolução, tornando escassos os recursos de memória e largura de banda, limitando o desempenho dos sistemas [3]. Por esta razão, compressão de imagens e texturas sempre foi um problema recorrente em computação. Diversos algoritmos, cada vez mais complexos e eficientes, têm sido apresentados para esta finalidade, desde o pioneiro *Block Truncation Coding (BTC)* [4] ao complexo método *Joint Photographic Experts Group (JPEG)* [5].

Em contrapartida, os dispositivos móveis, sabidamente providos de limitados recursos de processamento e armazenamento de dados, necessitam de algoritmos mais simples e rápidos (menor custo computacional) sem que, contudo, seja prejudicada significativamente a qualidade das imagens.

# 1.2 Contexto

*PACKMAN* [6], algoritmo desenvolvido na empresa *Sony Ericsson*™, ganhou notoriedade por equilibrar convenientemente simplicidade e eficiência. *PACKMAN* é um algoritmo de compressão, com perda de informação (*lossy*), baseado na divisão da imagem em blocos e na separação das informações de crominância e intensidade de cada bloco. Adicionalmente, tal algoritmo pode ser implementado em *hardware* com baixo custo, tornando-o ainda mais vantajoso para dispositivos móveis.

*Improved PACKMAN* ou simplesmente *iPACKMAN* [1] é uma extensão do primeiro, proposta pelo mesmo autor, e que utiliza, de forma mais eficiente, o espaço alocado para cada bloco através de conclusões estatísticas sobre redundância de informação. *iPACKMAN* realiza uma quantização vetorial num conjunto de imagens de treinamento (consideradas “universais”) e utiliza o resultado (um conjunto de tabelas chamado *codebook*) para sintetizar o conteúdo de outras imagens. Tal quantização é realizada por outro algoritmo derivado do conhecido LBG [7].

# 1.3 Objetivo e metodologia

Como veremos mais adiante, a quantização vetorial do *PACKMAN* e, posteriormente, do *iPACKMAN* foram baseadas em suposições generalizadoras que não foram demonstradas por seus idealizadores. O primeiro objetivo desta pesquisa foi investigar e validar tais suposições baseando-se em experimentos demonstráveis. Fizemos isto especificando o algoritmo de quantização vetorial do *iPACKMAN* de maneira sistemática. Este algoritmo é derivado do LBG, mas é adaptado para convergir mais rapidamente para a solução ótima neste caso em particular. Sua finalidade é obter um conjunto de dados (c*odebook*), a partir de um conjunto de imagens de treinamento (chamado de *benchmark*), capaz de aproximar o conteúdo de qualquer outra imagem estatisticamente.

Para validar a primeira hipótese de *iPACKMAN* (a de que o *codebook* proposto em [1] é ótimo para o dado *benchmark*) precisávamos ser capazes de encontrar o mesmo *codebook* a partir de nossa implementação do algoritmo de quantização vetorial e incapazes de encontrar um outro *codebook* melhor para a mesma finalidade.

Adicionalmente, a partir dos resultados obtidos, pudemos sugerir uma maneira de otimizar o processo de quantização vetorial e tornar a compressão ainda melhor. Propondo particularizar o tratamento das imagens em classes distintas, e obtendo um *codebook* diferente para cada classe, esperávamos conseguir resultados superiores àqueles obtidos pelo método tradicional, que utiliza apenas um *codebook* suposto ótimo e universal.

Para tanto, propomos uma forma de classificar as imagens em seis grupos distintos, formulamos um *benchmark* para cada grupo e executamos o quantizador com cada *benchmark*. Então, executamos testes para avaliar a qualidade da compressão com nossos novos *codebooks*. Era esperado que cada classe de imagens com um *codebook* particular pudesse aumentar a qualidade global da compressão (diminuindo o erro inerente) sem, contudo, aumentar significativamente seu custo computacional.

Por último, estendemos o *software* *ETCPack* (implementação livre do *iPACKMAN*) com as alterações propostas e executamos testes comprimindo imagens avulsas, de diferentes classes, com o objetivo de mensurar o ganho de qualidade alcançado e a viabilidade da proposta dentro de métricas de erro pré-estabelecidas.

# 1.4 Estrutura do trabalho

Este documento divide-se em quatro capítulos: neste capítulo é apresentada a motivação, o contexto e o objetivo do trabalho. O capítulo 2 faz uma revisão bibliográfica sobre os métodos de compressão de imagens e texturas mais relevantes para esta proposta e apresenta alguns conceitos úteis ao longo do trabalho. O Capítulo 3 discorre, principalmente, sobre o *Ericsson Texture Compression*: algoritmos de compressão de imagens para dispositivos móveis desenvolvidos e utilizados pela empresa *Sony Ericsson*™. Descrevemos nossos *insights* sobre o algoritmo e sobre como seria possível melhorar seu desempenho, preservando suas principais características, no capítulo 4. Os experimentos e seus resultados estão detalhados no Capítulo 5. Por último, o Capítulo 6 expõe nossas conclusões a respeito dos resultados e as considerações finais sobre o processo de otimização.

CAPÍTULO 2

Compressão de imagens e texturas

Este capítulo apresenta uma revisão sobre as principais técnicas de compressão de imagens e texturas. Na seção 2.1 explicamos algumas diferenças entre compressão de imagens e compressão de texturas, exibindo alguns desafios particulares na elaboração de esquemas de compressão para cada abordagem. O capítulo foi baseado no trabalho desenvolvido por F. Brayner [8].

# 2.1 Compressão de imagens versus compressão de texturas

O projeto de esquemas para compressão de texturas difere daqueles para compressão de imagens em alguns pontos importantes. Quando projetamos um esquema de compressão de texturas, não estamos preocupados apenas com armazenamento e transmissão como ocorre com a compressão de imagens. Ao lidar com texturas temos que levar em conta outros aspectos como a velocidade de decodificação do algoritmo e o acesso randômico a qualquer elemento da textura, por exemplo. Esses aspectos são divididos por A. Beers, em [9], da seguinte forma:

* Velocidade de decodificação: O tempo para acessar um pixel é um fator crítico, por isso é necessária uma descompressão rápida já que a textura será renderizada diretamente a partir do seu formato comprimido (economizando memória).
* Acesso randômico: Por não termos como saber antecipadamente como o sistema de renderização irá acessar a textura, temos que garantir que qualquer pixel possa ser acessado a qualquer momento sem que, por exemplo, seja necessário descompactar toda a textura.
* Taxa de compressão e qualidade visual: Alguns esquemas de compressão conseguem preservar completamente a qualidade da imagem (*lossless*), mas acabam chegando a taxas de compressão relativamente baixas quando comparados com os algoritmos *lossy*, onde ocorre alguma perda na qualidade da imagem. Aqui, fica evidente mais uma diferença entre compressão de imagens e texturas: em compressão de imagens a qualidade da imagem comprimida é mais importante, enquanto em compressão de texturas a qualidade da cena renderizada, não o mapa de texturas, tem maior importância.

A classificação elaborada por [9] deixa implícita outra diferença fundamental entre algoritmos de compressão de imagens e de texturas: a forma como a imagem é codificada. Em JPEG [5], a imagem é dividida e comprimida em blocos de 8x8 separadamente. A maioria dos algoritmos de compressão de imagens, como JPEG, é classificada como *Variable Bit Length Coding* (VBLC), ou seja, partes das imagens são codificadas com quantidades de bits diferentes. Algoritmos VBLC permitem situações onde um bloco que é difícil de codificar ocupa mais bits do que outro mais simples, por exemplo, bloco composto por uma única cor (figura 2.1).



Figura 2.1: *Variable Bit Length Coding* (VBLC).

Entretanto, taxa de bits variável significa que não se pode calcular o endereço de um pixel diretamente a partir dos bits em JPEG. Para saber o endereço de um determinado pixel, todo arquivo deve ser decodificado, o que não é viável em várias situações. Desta forma, a maioria dos codificadores de compressão de textura trabalha com uma taxa fixa de compressão, ou seja, cada bloco na imagem ocupa o mesmo número de bits e, desta forma, é possível calcular o endereço de um bloco particular a partir de uma seqüência de bits comprimida. Uma técnica de compressão de imagens como JPEG não é aplicável a compressão de texturas devido ao custo computacional de sua descompressão e a taxa de codificação variável feita de forma a dificultar o acesso randômico aos pixels.

Pelos motivos apresentados até aqui, é evidente que um esquema de compressão de texturas, por ser mais genérico, rápido, simples e de menor custo, é mais viável em aplicações móveis.

# 2.1 Esquemas de compressão de texturas

## 2.1.1 Block Truncation Coding

*Block Truncation Coding* (BTC), um esquema simples para compressão *lossy* de imagens, foi proposto por Delp em 1979 [4]. BTC comprime imagens dividindo-as em blocos de 4x4 pixels. Para cada bloco, duas cores em tons de cinza são armazenadas e cada pixel é aproximado por um desses dois valores. Ou seja, para cada bloco 4x4 duas cores base de 8 bits são selecionadas e o bloco passa a ser representado por um conjunto de índices (1 bit para cada pixel = 16 bits) para indexar qual das duas cores base o pixel será composto (Figura 2.2). Desta forma a taxa de compressão é de apenas de 2 bits por pixel (bpp). Este esquema influenciou vários algoritmos de compressão posteriores inclusive o *iPACKMAN*, objeto de estudo deste trabalho.

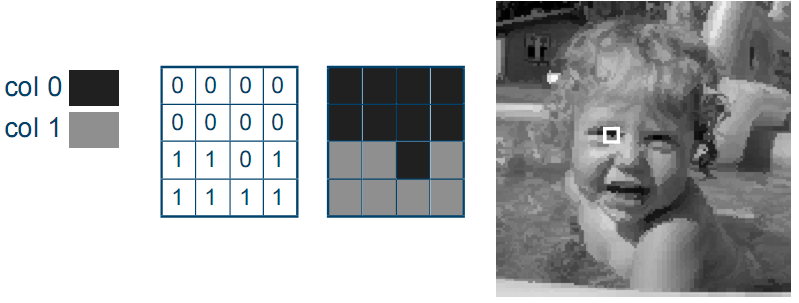


Figura 2.2: Com *Block Truncation Coding* - BTC, cada bloco de 4x4 pixels é transformado em 2 cores base de 8 bits e uma máscara de 16 bits que indexa essas duas cores base. Dessa forma, BTC comprime imagens em tons de cinza com 2 bpp.

## 2.1.2 S3TC

Uma técnica de compressão importante, que pode ser vista como mais uma adaptação dos métodos BTC/CCC, é o *S3 Texture Compression* (S3TC) [10], utilizado no *DirectX* [11], onde é conhecido como DXTn (onde n depende de como o canal alpha da textura é tratado. DXT1, DXT3 e DXT5 são os mais comumente utilizados). No DXT1, por exemplo, as texturas são comprimidas codificando cada bloco de pixels 4x4 em 64 bits. Duas cores base no formato RGB565 são armazenadas na primeira metade dos 64 bits. Na segunda metade dos 64 bits, um índice de dois bits é armazenado para cada pixel. Esse índice aponta para um conjunto de cores local que consiste das duas cores base e de duas cores adicionais entre as duas primeiras. Sendo assim, a tabela de *lookup* é consultada para determinar o valor da cor para cada pixel, com valor de 0 a 3 para cada cor Cx calculada para o bloco. Sendo C0 e C1 os valores RGB das cores base, as cores adicionais são calculadas da seguinte forma. Se C0 > C1:

* C2 = (2/3) C0 + (1/3) C1
* C3 = (1/3) C0 + (2/3) C1

Se não:

* C2 = (1/2) C0 + (1/2) C1
* C3 = uma variação da cor preta com alpha. C3 assumirá uma cor opaca se os bits de controle forem 0, 1 ou 2. Caso os bits indiquem o valor 3, C3 será transparente.

DXT3 e DXT5 armazenam 128 bits por cada bloco 4x4 de pixels (8 bpp) funcionando de forma similar ao DXT1, mas por utilizar mais bits utilizam um número maior de interpolações além do suporte ao canal alpha. A qualidade do S3TC é geralmente maior que a fornecida pelo CCC, mas os ganhos são alcançados com um custo de 4 bpp, no caso do DXT1. Uma desvantagem desse método é que apenas quatro cores podem ser utilizadas por bloco, logo, DXTC tem problemas com texturas com muitas cores com valores diferentes no mesmo bloco. Ivanov e Kuzmin [12] tentaram resolver esse problema utilizando cores dos blocos vizinhos, mas o aumento na utilização da banda da memória se tornou um problema. De qualquer forma, S3TC se tornou um dos métodos de compressão mais populares.

## 2.1.2 Vector Quantization (Quantização Vetorial)

Utilizado em várias áreas como compressão de áudio e imagem, *Vector Quantization* (VQ) também influenciou vários esquemas de compressão de texturas. A idéia básica por trás de VQ é tentar encontrar um número menor de vetores que aproximem uma dada distribuição vetorial mantendo uma taxa de erro pequena. Beers et al. [9] tratam a textura como um conjunto de blocos de pixels que utiliza VQ para comprimir texturas e alcançar taxas de compressão bastante altas. VQ tenta caracterizar esse conjunto de blocos por um grupo menor de blocos representativos chamado de *codebook*. Sendo assim, a textura é representada por uma tabela de índices (*index map*) e o *codebook*. Assumindo que as coordenadas de textura (s,t) já tenham sido convertidas para os inteiros (i,j), podemos definir esse algoritmo com os seguintes passos:

1. Determinar em qual bloco B o pixel (i,j) se encontra e determinar o *offset* do pixel (i,j) dentro do bloco.
2. Recuperar o índice associado com o bloco B para obter o *codeword* correspondente no *codebook*.
3. Recuperar o pixel (i,j) dentro do *codeword* indicado no passo anterior .

A Figura 2.3 mostra esse processo de decodificação de um *texel* (*texture element*, pixel de uma textura) através dos três passos. Segundo o próprio autor, a parte crítica do algoritmo é a definição do *codebook*, em outras palavras, a definição de qual é o menor conjunto de vetores que melhor representa o conjunto total. Várias técnicas foram utilizadas como o *Tree Structured VQ* e o *Full Search VQ* destacando-se, porém, o algoritmo LBG [7] do qual deriva a quantização vetorial utilizada em *iPACKMAN*. O esquema permitia decodificação rápida alcançando taxas de 1 a 2 bpp.

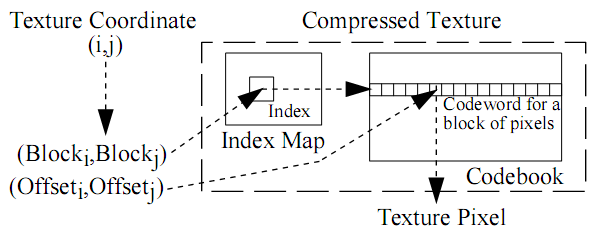


Figura 2.3: Acessando um pixel utilizando o esquema baseado em *Vector Quantization*: com as coordenadas de textura (Blocki, Blockj) consegue-se descobrir o codeword e com o deslocamento do pixel dentro do bloco, consegue-se obter a cor final do pixel dentro do *codeword*.

CAPÍTULO 3

Ericsson Texture Compression (ETC)

Este capítulo revisa os trabalhos que motivaram esta proposta. *Ericsson Texture Compression* é uma técnica de compressão de texturas, de baixa complexidade, desenvolvida pelo *Ericsson Research* e baseada, principalmente, em um algoritmo chamado de *iPACKMAN* (às vezes, os termos ETC e *iPACKMAN* são utilizados como sinônimos). Tal algoritmo é sucessor do *PACKMAN*, motivo pelo qual apresentamos este primeiro. Estes algoritmos foram desenvolvidos para serem implementados em *hardware*, embora também tenham sido utilizados em outras aplicações, atendendo aos requisitos discutidos nos capítulos 1 e 2.

Aqui, estaremos interessados em expor os conceitos destes algoritmos que são mais relevantes para a nossa proposta.

# 3.1 PACKMAN

*PACKMAN* é um algoritmo de compressão de imagens, com perda de informação (*lossy*), criado por J. Ström e T. Akenine-Möller [6], visando uma baixa complexidade para que se possa implementá-lo com facilidade e baixo custo em *hardware*. Este algoritmo foi inspirado no BTC embora utilize o conceito de quantização vetorial. A imagem é dividida em blocos de 2 x 4 pixels, onde cada bloco é representado por 32 bits. Para cada bloco é armazenada uma cor base média de 12 bits, com 4 bits para componente R, G e B (abreviado RGB444), e os outros 20 bits modulam a luminescência para cada pixel do bloco. A idéia básica da técnica é separar a informação de cor (crominância) da informação de intensidade (luminescência) dos pixels, como exemplificado na imagem 3.1. Desta forma, portanto, *PACKMAN* atinge uma taxa de compressão fixa de 4 bpp.

Figura 3.1: Separação das informações de cor (crominância), que é fixa para cada bloco de 2x4 pixels, e intensidade (luminescência), que varia para cada pixel.

Para modularmos a intensidade de um pixel, simplesmente escolhemos um valor constante, em um conjunto de tabelas, e essa constante é somada a cada componente RGB da cor base do bloco ao qual este pixel pertence. Este conjunto de tabelas, conhecido como *codebook*, é fixo, único e universal. Cada bloco deve escolher uma das possíveis tabelas (também conhecidas como *codewords)* e cada pixel, dentro deste bloco, deve selecionar uma constante da tabela para modular sua intensidade.

A Tabela 3.1 mostra o *codebook* universal utilizado no *PACKMAN*. Cada coluna representa uma tabela (*codeword*) e cada linha um índice desta tabela. Temos 16 tabelas e 4 índices por tabela.

Na descompressão, a cor base é expandida, de 12 para 24 bits (RGB888), e o valor constante selecionado na tabela é adicionado à cor base expandida. Os componentes de cor devem ser truncados ao intervalo [0, 255] (processo que chamamos de *clamping*).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| -8 | -12 | -31 | -34 | -50 | -47 | -80 | -127 | -16 | -24 | -62 | -68 | -100 | -94 | -160 | -254 |
| -2 | -4 | -6 | -12 | -8 | -19 | -28 | -42 | -4 | -8 | -12 | 24 | -16 | -38 | -56 | -84 |
| 2 | 4 | 6 | 12 | 8 | 19 | 80 | 42 | 4 | 8 | 12 | 24 | 16 | 38 | 56 | 84 |
| 8 | 12 | 31 | 34 | 50 | 47 | 28 | 127 | 16 | 24 | 62 | 68 | 100 | 94 | 160 | 254 |

Tabela 3.1: *Codebook* utilizada pelo algoritmo *PACKMAN*.

## 3.1.1 Quantização vetorial no PACKMAN

O *codebook* utilizado pelo *PACKMAN* não é aleatório. Ele é obtido a partir de uma variação do algoritmo de quantização vetorial LBG [7] e minimiza o erro cometido por cada pixel (diferença média entre os valores do pixel na imagem original e aquela obtida após a descompressão), maximizando a qualidade geral da textura, para um conjunto de imagens de treinamento (o *benchmark)*. Este conjunto de imagens foi arbitrado pelo idealizador do algoritmo a fim de cobrir a maior quantidade possível de tipos de imagens, para as mais variadas finalidades e assim alcançar, na média, um *codebook* com desempenho ótimo. Este *benchmark* é exibido na figura 3.2 abaixo.

Uma observação importante é que na quantização vetorial, as tabelas são forçadas a serem simétricas (por exemplo [-8, -2, 2, 8]). Isto agiliza o processo de convergência para a solução ideal e permite que apenas metade da tabela seja armazenada em *hardware*.

  
Figura 3.2: *Benchmark* utilizado na tentativa de gerar um *codebook* universal ótimo.

# 3.2 iPACKMAN

Em imagens onde a crominância não varia suavemente de um bloco para outro, o *PACKMAN* não apresenta um resultado muito satisfatório, pois mesmo a menor variação de crominância pode significar um intervalo muito grande, quando se usa uma representação de 12 bits. Por outro lado, nos casos onde a crominância varia suavemente de um bloco para outro, podemos observar uma redundância significativa de informação na imagem comprimida, pois a cor base de cada bloco é repetida, com pouca ou nenhuma variação, nos blocos adjacentes.

O *iPACKMAN* (*improved PACKMAN*) surgiu na tentativa de resolver estes problemas. Ao invés de codificar uma cor base para cada bloco de 2 x 4, são agrupados 2 blocos, formando um grande bloco de 4 x 4, e a cor base desses blocos é obtida da seguinte forma:

1. Armazena-se a cor base do primeiro bloco em 15 bits (5 bits para cada componente de cor RGB).
2. Armazena-se apenas a diferença da cor base do segundo bloco, em relação ao primeiro, em 9 bits (3 bits para cada componente de cor RGB).

Para justificar esta modificação, foram realizados testes com um conjunto de 20 imagens (as mesmas do *benchmark*), que mostraram que a grande maioria, cerca de 88%, dos blocos podem ser codificados diferencialmente usando-se apenas 3 bits por componente de cor (gráfico da figura 3.3). Para isso, é calculada a cor média para os blocos 2 x 4 adjacentes e ambas são quantizadas para RGB555, melhorando a representação da crominância.

Como em alguns casos os blocos não podem ser codificados dessa forma, foi inserido um bit para decidir se vai ser usada a codificação diferencial ou não. Este bit será chamado de *diffbit*. Este bit foi extraído do *codebook*, que antes tinha 4 bits (16 *codewords*), que passa a ter 3 bits (8 *codewords*). Dessa forma teremos um bit sobrando, que será utilizado para decidir se os blocos serão 2 x 4 ou 4 x 2, chamaremos esse bit de *flipbit*. O *flipbit* é decidido, em tempo de compressão, por tentativa e erro. Ou seja, o ETC comprime o bloco 4x4 separando-o em blocos 2x4 e, em seguida, separando-o em blocos 4x2. A alternativa que apresentar melhor qualidade (menor erro médio) é escolhida. As métricas de erro utilizadas em todas as etapas deste algoritmo serão minuciosamente explicadas no próximo capítulo.



Figura 3.3: De acordo com o histograma 88% dos blocos estão entre -4 e 3 e, portanto, nestes casos o *iPACKMAN* é utilizado.

O *codebook* do *iPACKMAN* é mostrado na Tabela 3.2 e foi obtido de maneira similar àquela utilizada no *PACKMAN*.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| -8 | -17 | -29 | -42 | -60 | -80 | -106 | -183 |
| -2 | -5 | -9 | -13 | -18 | -24 | -33 | -47 |
| 2 | 5 | 9 | 13 | 18 | 24 | 33 | 47 |
| 8 | 17 | 29 | 42 | 60 | 80 | 106 | 183 |

Tabela 3.2: *Codebook* do *iPACKMAN*. Tem apenas metade do número de tabelas do *PACKMAN*.

*iPACKMAN* consegue maior qualidade de compressão em relação ao *PACKMAN* mantendo, contudo, a taxa de compressão em 4 bpp. A representação de um bloco 4x4 no *iPACKMAN* é dada como mostrado na figura 3.4.

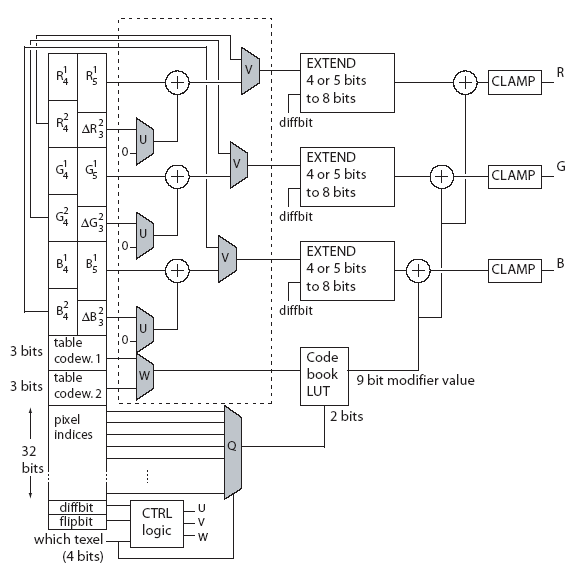
Figura 3.3: Uso dos bits para representar um bloco 4x4. O primeiro bloco tem sua cor base inteiramente representada. A cor base do segundo bloco é representada diferencialmente em relação a do primeiro. 3 bits, para cada bloco, indexam as tabelas utilizadas. Os outros 32 bits, em amarelo, indexam os índices das tabelas escolhidos por cada pixel.

A descompressão funciona de maneira análoga ao *PACKMAN*, porém são levados em consideração o *diffbit* e o *flipbit*. Na figura 3.4 podemos ver um comparativo visual que mostra a superioridade do *iPACKMAN*.

Figura 3.4: Comparação visual entre *PACKMAN* e *iPACKMAN* na compressão de uma imagem onde a cor base varia suavemente ao longo dos blocos.

## 3.2.1 Descompressão direta com iPACKMAN

A maior vantagem de *iPACKMAN* em relação aos outros algoritmos de compressão está na simplicidade e baixo custo de implementação em hardware, tornando-o ideal em aplicações embarcadas. O modelo esquemático do módulo de descompressão do algoritmo é mostrado a seguir na figura 3.5. Este modelo utiliza apenas seis somadores e alguns multiplexadores.

  
Figura 3.5. Esquema de descompressão simples do *iPACKMAN*.

CAPÍTULO 4

Classificação de imagens

Os objetivos deste trabalho deixaram-nos claro de que havia duas hipóteses distintas cuja veracidade precisávamos investigar:

1. O *codebook* proposto pelo autor de *iPACKMAN* é de fato um conjunto de dados ótimo para ser utilizado por este algoritmo na compressão das imagens do *benchmark.*
2. Este *codebook,* além de ótimo, é universal, ou seja, pode ser utilizado com mesmo desempenho em qualquer imagem que não esteja no *benchmark*.

Para obtermos uma resposta para o primeiro questionamento a abordagem foi resgatar as imagens do benchmark, implementar sistematicamente o algoritmo de quantização vetorial descrito empiricamente pelo autor de *iPACKMAN,* e executar este algoritmo com o *benchmark* como parâmetro. Então, pudemos comparar os resultados com os do *codebook* original. Também tentamos de maneira exploratória encontrar outros *codebooks* e compará-los ao original.

Para investigar a segunda hipótese resolvemos utilizar a seguinte abordagem: propusemos uma classificação das imagens candidatas (a serem comprimidas) de acordo com uma taxonomia preestabelecida. Fizemos isto porque quisemos testar se imagens de uma mesma classe (por exemplo: as naturais em detrimento das sintéticas) possuíam padrões semelhantes entre si e que, portanto, faria sentido agrupá-las e fazê-las utilizar um mesmo *codebook*. Ao passo que imagens de outras classes seriam comprimidas de forma mais eficiente com outros *codebooks* por possuírem outros padrões, e assim sucessivamente. Nesta nova abordagem o *Ericsson Texture Compression* deveria decidir a que classe de imagens uma determinada imagem pertence antes de proceder com a compressão: atuando, portanto, como um classificador. Esta tarefa poderia ser feita de forma análoga aos passos de busca exaustiva do próprio *iPACKMAN*, comprimindo a imagem simultaneamente com os *codebooks* de cada classe e escolhendo aquele em que o erro foi minimizado. Desta forma a imagem seria classificada automaticamente pelo ETC em tempo de compressão.

Caso os resultados obtidos fossem próximos aos que se obtinha anteriormente, não justificando os recursos computacionais extras alocados para testar os novos *codebooks*, isto provaria a universalidade do *codebook* original e validaria por completo os resultados publicados em *iPACKMAN*. Por outro lado, caso conseguíssemos ganhos significativos separando as imagens em classes e mostrando sua viabilidade de implementação, teríamos contribuído de maneira expressiva para o aperfeiçoamento do algoritmo.

# 4.1 As classes

Baseando-nos nas próprias imagens propostas no *benchmark* original da figura 3.1, elaboramos a seguinte árvore de classificação de imagens (figura 4.1) que é capaz de separar imagens de acordo com seu significado (ou padrão de conteúdo).

Figura 4.1: Árvore de classificação de imagens. Cada imagem será, em tempo de compressão, classificada como “paisagem”, “humanos”, “outras imagens naturais”, “textos”, “texturas” ou “computação gráfica”.

Primeiramente classificamos as imagens entre “Naturais” ou “Artificiais”. Dentre as “Naturais” podemos classificá-las como “Paisagens”, “Humanos” ou “Outras imagens naturais”. Já dentre as “Artificiais”, as classificamos como “Textos”, “Texturas” ou “Computação Gráfica”.

Esta árvore de classificação é um exemplo suficientemente complexo para executarmos nossos testes. Diversos outros métodos de classificação poderiam ter sido utilizados. De maneira geral, quanto mais folhas tivermos em nossa árvore de classificação, melhor espera-se que seja a qualidade das imagens, pois teríamos um maior número de *codebooks* à disposição do algoritmo de compressão. Por outro lado, quanto mais *codebooks*, mais hardware é necessário para armazenar as tabelas e para executar a compressão com cada um deles e escolher aquele mais adequado à imagem e, portanto, mais cara se torna sua implementação. A escolha da árvore de classificação e de seu tamanho dependeria dos parâmetros e requisitos da aplicação.

A idéia central da proposta foi elaborarmos um conjunto de imagens de treinamento (*benchmarks*) para cada classe e executarmos o algoritmo de quantização vetorial com cada um deles e, desta forma, obtermos **um *codebook* otimizado para cada classe de imagens**. Os *benchmarks* elaborados são apresentados na figuras 4.2, 4.3, 4.4, 4.5, 4.6 e 4.7. As imagens utilizadas são todas *true color*, o que significa que não sofreram compressão anteriormente e preservam suas cores reais em valores RGB. Cada *benchmark*, assim como no original, possui 20 imagens de 128 x 128 pixels compondo lado a lado uma única imagem que será usada para se obter os *codebooks* otimizados. As imagens foram cortadas e reduzidas com o aplicativo Gimp™.

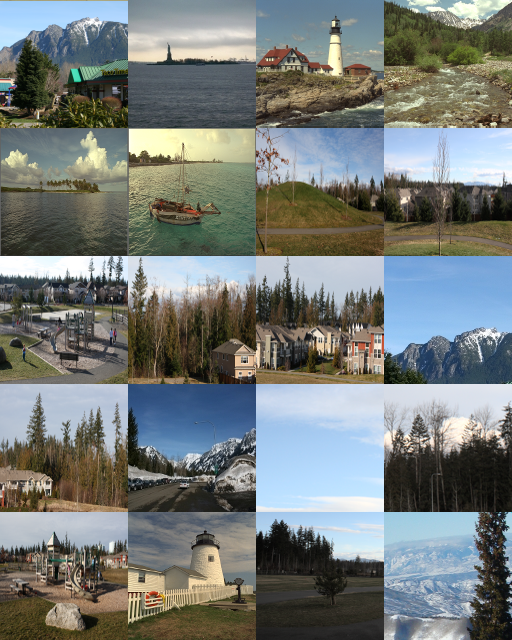
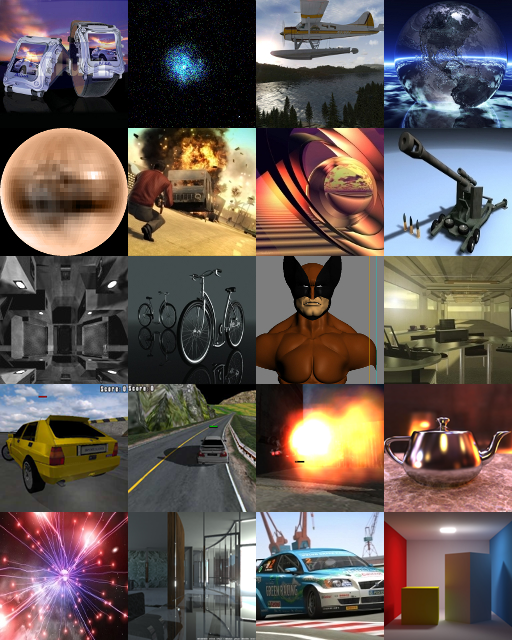
  
Figura 4.2. *Benchmark* para a classe: “Paisagens” em “Naturais”.

  
Figura 4.3. *Benchmark* para a classe “Humanos” em “Naturais”.

  
Figura 4.4. *Benchmark* para a classe “Outros” em “Naturais”.

  
Figura 4.5. *Benchmark* para a classe “Textos” em “Artificiais”.

  
Figura 4.6. *Benchmark* para a classe “Texturas” em “Artificiais”.

  
Figura 4.7. *Benchmark* para a classe “Computação Gráfica” em “Artificiais”.

CAPÍTULO 5

Experimentos

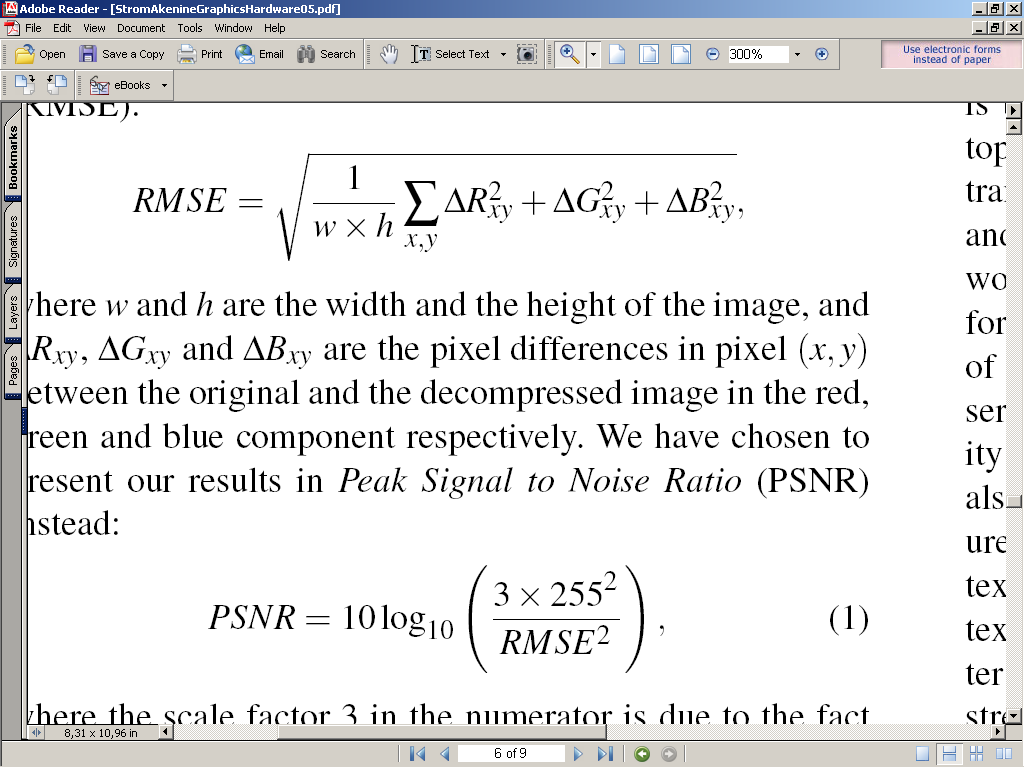
Este capítulo contém os ensaios realizados e os resultados obtidos neste trabalho.

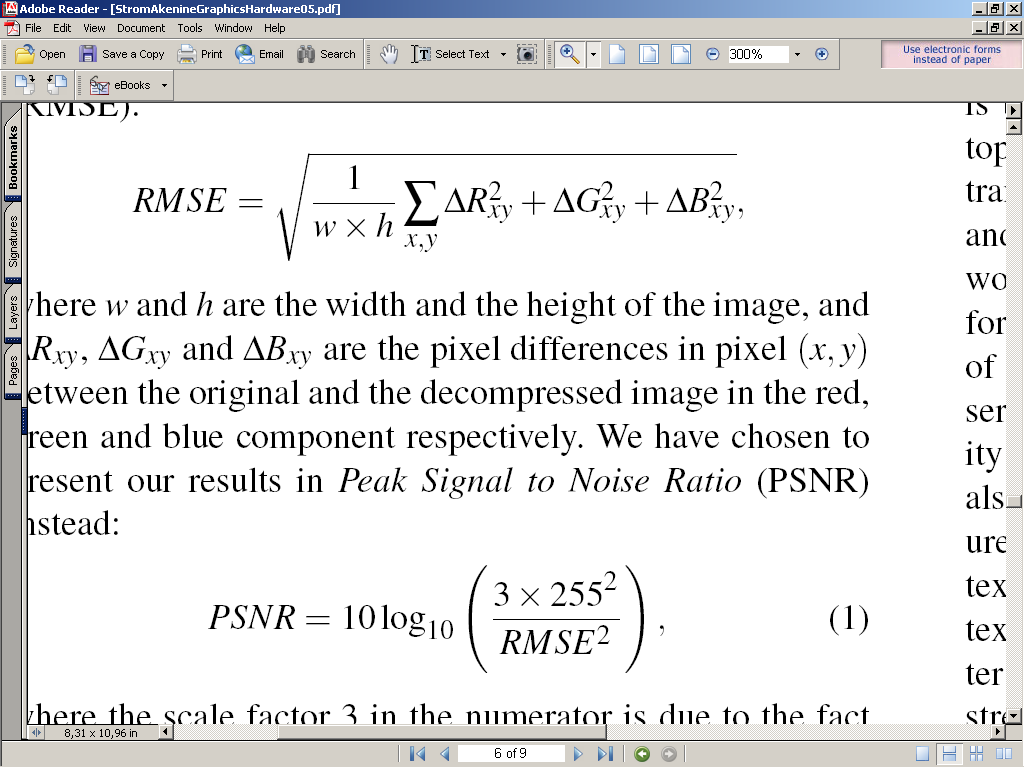
# 5.1 Métricas de erro

Adotamos formas convencionais de mensuração de erro entre imagens [1]. Utilizamos o RMSE (*root mean square error* ou erro médio quadrático) para comparar duas imagens. Usualmente estávamos interessados em comparar uma imagem original com aquela obtida, após ser comprimida e descomprimida com um determinado *codebook*. RMSE (figura 5.1) é a métrica usada por Fenney [13] para expressar resultados de compressão em relação à imagem original.

PSNR (*peak signal to noise ratio*) é uma função logarítmica do RMSE (figura 5.2) e o utilizamos por ser uma forma mais prática para comparar uma mesma imagem comprimida com diferentes *codebooks*. Em compressão de imagens, quanto maior o PSNR, maior é a qualidade da imagem, pois menor é o RMSE e, portanto, a imagem se parece mais com a original.

Neste capítulo, quando mencionado que um novo *codebook* melhorou a qualidade de compressão de uma imagem, isto significa que, com este novo *codebook*, aumentamos o PSNR da imagem comprimida em relação ao PSNR que era obtido com o *codebook* antigo. Porém, a efeito de comparação entre *codebooks*,não faz sentido compararmos o PSNR obtido após a compressão de duas imagens distintas.

  
Figura 5.1: *Root mean square error*. Medida adimensional tradicional de erro entre vetores. Onde “w” e “h” são a largura e altura da imagem, respectivamente, e “∆R”, “∆G”, “∆B” são as diferenças das componentes de cor R, G e B (de 0 a 255), para o pixel na coordenada (x,y), entre as duas imagens.

  
Figura 5.2: *Peak signal to noise ratio*. Função logarítmica do RMSE, medida em decibel (dB).

# 5.2 A quantização vetorial (LBG adaptado)

Como já mencionado, o algoritmo de quantização vetorial utilizado no *ETC* é uma versão modificada do LBG [7], que foi desenvolvido de forma empírica e não foi publicado por seus idealizadores. A primeira contribuição significativa deste trabalho foi, então, especificá-lo sistematicamente (seção 5.2.1) e comprovar sua eficácia.

O objetivo deste algoritmo é encontrar um *codebook* ótimo para um dado *benchmark*, ou seja, aquele que minimiza o erro inerente quando utilizado pelo *iPACKMAN* para comprimir as imagens. Portanto, nossa implementação do algoritmo deveria ser capaz de convergir para um conjunto de 8 tabelas, com 4 coeficientes simétricos cada, similar ao exibido na tabela 3.2.

## 5.2.1 Descrição em linguagem natural do quantizador

PASSO A:

Primeiramente, inicializamos as oito tabelas de maneira completamente aleatória, com números em formato RGB888 (de 0 a 255), respeitando, contudo, a condição de simetria. Ex: [-8, -2, 2, 8] e não, por exemplo, [-9, -3, 0, 7].

PASSO B:

A imagem de treinamento é comprimida usando as tabelas e estas são ajustadas da seguinte maneira: encontramos os blocos que utilizaram cada tabela (blocos cujo valor de cor base excedem 255 (*clamping*) são ignorados). Depois, identificamos o coeficiente que cada pixel escolheu dentro de cada bloco. Então, substituímos o valor de cada coeficiente da tabela pelo valor médio que cada pixel gostaria de ter (aquele que reduziria seu erro médio a zero). Ex: Digamos que alguns blocos escolheram a tabela [-6, -3, 3, 6] e, dentro destes blocos, alguns pixels escolheram -6 e 6 (porque foram os valores disponíveis que mais se aproximavam do ideal para estes pixels). Nós então calculamos o valor de correção que estes pixels realmente gostariam de ter: por exemplo, -7, -7, -9, -5, -5 para os pixel que escolheram -6 e 8, 8, 9, 6 para os pixels que escolheram 6. Tiramos então o valor médio de [7, 7, 9, 5, 5, 8, 8, 9, 6], o que dá 7.111 (arredondado para 7). Nós poderíamos, por exemplo, encontrar 4.25 para os pixels que escolheram -3 ou 3 e, assim, a nova tabela [-7, -4, 4, 7] substituiria a anterior no *codebook*.

Isto é feito para cada tabela e então nós comprimimos a imagem de treinamento novamente, fazemos tudo de novo e assim sucessivamente até que nenhuma tabela possa mais ser corrigida.

PASSO C:

Agora, normalmente, algumas tabelas são muito boas e outras são tão ruins que, às vezes, nem um único bloco a escolhe. Isto faz com que o PASSO B não consiga mais progredir em sua tarefa de otimização. Portanto, neste passo, precisamos substituir a tabela menos útil, ou seja, aquela que menos reduz o erro médio. Comprimimos a imagem novamente, mas desta vez guardamos o quanto cada tabela reduz o RMSE (ou, equivalentemente, aumenta o PSNR). Então, removemos a tabela menos útil do *codebook*. É importante notar que isto não é o mesmo que simplesmente remover a tabela menos usada: algumas vezes uma tabela é muito pouco usada, mas reduz sensivelmente o erro global. Então, nós geramos uma tabela aleatória e a colocamos no *codebook*. Comprimimos a imagem novamente. Caso, ainda assim, o erro não tenha diminuindo, repetimos o processo até que o erro diminua.

PASSO D:

Executamos a quantização descrita no PASSO B novamente.

PASSO E:

Desta vez colocamos um “ruído” aleatório em cada coeficiente de cada tabela (mantendo-a, porem, simétrica e com coeficientes entre 0 e 255). Por ex: uma tabela pode ser [-8, -2, 2, 8] e o ruído gerado, -2. Então a nova tabela será: [-10, 0, 0, 10]. Isto acontece para que possamos sair de um possível ponto de mínimo local e que, desta forma, o algoritmo possa convergir para a solução ótima. Voltamos ao PASSO A. A cada iteração do algoritmo decrementamos o tamanho possível (*range*) do ruído, até que ele se torne nulo e então o algoritmo pára e inicia a execução do seu ultimo passo.

PASSO F (FINAL):

Temos, neste ponto, uma tabela provavelmente muito boa. Realizamos, finalmente, uma “busca coordenada” na vizinhança da tabela:

1. Comprima a imagem, guarde o erro.
2. Incremente o primeiro coeficiente da primeira tabela.
3. Comprima a imagem novamente. Caso o erro tenha diminuído, guarde este *codebook* e reinicie o processo. Caso contrário, tente o caminho inverso (decremente o mesmo coeficiente).
4. Se for possível melhorar, repita o procedimento. Caso não seja mais possível melhorar em nenhuma direção, passe para o próximo coeficiente.
5. Faça a mesma coisa para cada coeficiente de cada tabela seqüencialmente. Quando nenhum coeficiente puder mais ser corrigido o algoritmo finalmente para.

Fazemos todo este procedimento (passos A até F), algumas centenas de vezes (arbitrariamente) e escolhemos a melhor tabela-resultado.

# 5.3 Testes e resultados

O algoritmo de quantização vetorial envolve vários passos aleatórios. Duas execuções consecutivas do algoritmo com os mesmo parâmetros costumam resultar em *codebooks* diferentes (com desempenhos muito parecidos, porém). Além disso, por causa da complexidade computacional, não é possível demonstrar que um determinado resultado alcançado pelo algoritmo, executando por um determinado período de tempo, seja ótimo. Isto acontece porque existe uma combinação muito grande de tabelas possíveis e testá-las uma por uma seria inviável (cada uma das 8 tabelas possuem 4 coeficientes que podem variar de 0 a 255).

Por este motivo, após a implementação deste algoritmo, era necessário especificar um critério de parada padrão. Este critério poderia ser baseado no tempo de execução, na diminuição do erro ou na diminuição do ruído. Optamos pela diminuição do erro. Parávamos o algoritmo quando, após uma iteração completa (passos de A até o final), o PSNR não variava ou variava menos que 0.000001 dB (*threshold*). Com este critério de parada, o tempo médio de execução do algoritmo foi de 4 horas.

De acordo com nossa metodologia e com o objetivo do projeto, deveríamos primeiramente reproduzir o experimento dos autores do *iPACKMAN* e encontrar o *codebook* da tabela 3.2 utilizando o conjunto de imagens da figura 3.2 (o *benchmark* original). Após as três primeiras execuções do programa obtivemos as tabela mostradas a seguir:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| -8 | -17 | -29 | -42 | -59 | -108 | -80 | -202 |
| -2 | -5 | -8 | -13 | -16 | -24 | -27 | -39 |
| 2 | 5 | 8 | 13 | 16 | 24 | 27 | 39 |
| 8 | 17 | 29 | 42 | 59 | 108 | 80 | 202 |

Tabela 5.1: *Codebook* obtido na primeira execução do quantizador utilizando o *benchmark* original como parâmetro. PSNR = 32.209353 dB.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| -8 | -16 | -28 | -42 | -60 | -80 | -108 | -199 |
| -2 | -5 | -8 | -13 | -17 | -23 | -27 | -39 |
| 2 | 5 | 8 | 13 | 17 | 23 | 27 | 39 |
| 8 | 16 | 28 | 42 | 60 | 80 | 108 | 199 |

Tabela 5.2: *Codebook* obtido na segunda execução do quantizador utilizando o *benchmark* original como parâmetro. PSNR = 32.206213 dB.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| -9 | -15 | -26 | -40 | -57 | -76 | -105 | -182 |
| -3 | -2 | -7 | -13 | -16 | -23 | -30 | -54 |
| 3 | 2 | 7 | 13 | 16 | 23 | 30 | 54 |
| 9 | 15 | 26 | 40 | 57 | 76 | 105 | 182 |

Tabela 5.3: *Codebook* obtido na terceira execução do quantizador utilizando o *benchmark* original como parâmetro. PSNR = 32.170604 dB.

As tabelas 4.1, 4.2 e 4.3 mostram resultados para execuções diferentes do quantizador. Os *codebooks* são ligeiramente diferentes por causa do fator aleatório inerente ao algoritmo, porém o PSNR em dois dos três casos apresentados revelou resultados surpreendentemente superiores, mesmo que por muito pouco, àqueles obtidos pelo idealizador do algoritmo (PSNR = 32.204845 dB). Isto pode ser explicado pelo fato de que, provavelmente, utilizamos um critério de parada mais rigoroso, dando ao quantizador mais tempo para convergir a um resultado ligeiramente melhor. De acordo com o próprio trabalho original do *PACKMAN* [6] apenas uma diferença de, pelo menos, 0.25 dB pode ser percebida visualmente em uma imagem.

Os experimentos realizados e seus resultados mostraram que é praticamente impossível encontrar uma solução pontual para o problema. Porém, existe uma região de interesse que é rapidamente atingida pelo quantizador. Contudo, mesmo após centenas de execuções, o quantizador não foi capaz de encontrar uma única tabela que obtivesse resultado superior em, pelo menos, 0.25 dB ao resultado original. Isto reforça o resultado obtido empiricamente por J. Ström de que este seria um *codebook* ótimo para este conjunto de imagens. Faltava-nos investigar a real universalidade destes resultados, ou seja, precisávamos descobrir se este c*odebook* pode ser utilizado para qualquer imagem, ou conjunto de imagens, de maneira igualmente satisfatória.

# 5.4 Otimizando *codebooks* para classes de imagens distintas

Para responder ao questionamento deixado em aberto no item 5.3, utilizamo-nos da abordagem de classificação das imagens apresentada no capítulo 4. Executamos o quantizador para cada um dos seis *benchmarks* elaborados a fim de encontrar os *codebooks* ótimos para cada classe. Utilizamos, para esta finalidade, exatamente os mesmos critérios do item 5.2. Os *codebooks* obtidos são apresentados nas tabelas 5.4, 5.5, 5.6, 5.7, 5.8 e 5.9.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  | -7 | -14 | -21 | -36 | -29 | -45 | -59 | -80 |
|  | -2 | -4 | -6 | -8 | -11 | -14 | -19 | -26 |
|  | 2 | 4 | 6 | 8 | 11 | 14 | 19 | 26 |
|  | 7 | 14 | 21 | 36 | 29 | 45 | 59 | 80 |

Tabela 5.4: Melhor *codebook* encontrado para o *benchmark* da classe “**Paisagens**” em “Naturais”. PSNR = 34.209077 dB.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| -8 | -16 | -26 | -60 | -38 | -52 | -77 | -108 |
| -2 | -5 | -8 | -11 | -12 | -18 | -23 | -33 |
| 2 | 5 | 8 | 11 | 12 | 18 | 23 | 33 |
| 8 | 16 | 26 | 60 | 38 | 52 | 77 | 108 |

Tabela 5.5: Melhor *codebook* encontrado para o *benchmark* da classe “**Humanos**” em “Naturais”. PSNR = 33.305892 dB.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| -8 | -14 | -21 | -31 | -43 | -59 | -79 | -110 |
| -2 | -4 | -6 | -9 | -13 | -17 | -23 | -32 |
| 2 | 4 | 6 | 9 | 13 | 17 | 23 | 32 |
| 8 | 14 | 21 | 31 | 43 | 59 | 79 | 110 |

Tabela 5.6: Melhor *codebook* encontrado para o *benchmark* da classe “**Outras**” em “Naturais”. PSNR = 31.965379 dB.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| -194 | -46 | -73 | -132 | -103 | -220 | -149 | -175 |
| -3 | -11 | -18 | -22 | -32 | -35 | -57 | -91 |
| 3 | 11 | 18 | 22 | 32 | 35 | 57 | 91 |
| 194 | 46 | 73 | 132 | 103 | 220 | 149 | 175 |

Tabela 5.7: Melhor *codebook* encontrado para o *benchmark* da classe “**Textos**” em “Artificiais”. PSNR = 29.947997 dB.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| -11 | -20 | -30 | -39 | -59 | -82 | -105 | -155 |
| -3 | -6 | -8 | -13 | -14 | -25 | -33 | -31 |
| 3 | 6 | 8 | 13 | 14 | 25 | 33 | 31 |
| 11 | 20 | 30 | 39 | 59 | 82 | 105 | 155 |

Tabela 5.8: Melhor *codebook* encontrado para o *benchmark* da classe “**Texturas**” em “Artificiais”. PSNR = 27.132904 dB.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| -7 | -17 | -27 | -40 | -58 | -81 | -105 | -145 |
| -2 | -5 | -9 | -11 | -16 | -23 | -33 | -45 |
| 2 | 5 | 9 | 11 | 16 | 23 | 33 | 45 |
| 7 | 17 | 27 | 40 | 58 | 81 | 105 | 145 |

Tabela 5.9: Melhor *codebook* encontrado para o *benchmark* da classe “**Computação gráfica**” em “Artificiais”. PSNR = 33.662547 dB.

Como podemos perceber, os codebooks obtidos para as diferentes classe de imagens diferem bastante do *codebook* universal do *iPACKMAN*. Isto nos evidencia que os resultados comparativos entre as duas técnicas serão também distintos. Reforçando esta evidencia, é possível perceber algumas explicações para a diferença entre os *codebooks*. Por exemplo, o *codebook* obtido para a classe “Textos” apresenta mais tabelas com valores muito discrepantes em relação aos outros *codebooks*. Isto pode ser explicado pelo fato de que, neste tipo de imagem, a cor costuma variar bruscamente de um pixel a outro dentro de um mesmo bloco (o preto no branco das imagens de texto).

Os resultados mostram, inclusive, que o algoritmo *iPACKMAN* em si possui desempenho melhor para imagens naturais como paisagens e pessoas do que para textos por exemplo (maiores valores de PSNR). Isto era de se esperar uma vez que nestas imagens as cores variam de maneira mais suave e *iPACKMAN* explora justamente a redundância de cor nas regiões de proximidade dos pixels (blocos). Portanto, pode ser ainda mais vantajoso possuir *codebooks* especiais para classes de imagens de baixo desempenho.

Resultados numéricos e visuais são apresentados a seguir.

# 5.5 Testando os novos *codebooks*

Quando J. Strom propôs *iPACKMAN*, tomou a precaução de incluir na representação binária de cada bloco o *diffbit*. Com este bit, cada bloco poderia escolher se usaria a nova representação diferencial de cor proposta ou se continuaria a usar a representação antiga (do *PACKMAN*). Isto foi feito para garantir que nenhuma imagem teria qualidade inferior com o novo algoritmo: na pior das hipóteses continuaria igual. Da mesma forma em nossa nova proposta adicionamos o codebook original do *iPACKMAN* (tabela 3.2) aos nossos seis codebooks encontrados. Assim garantimos antecipadamente que qualquer imagem terá desempenho melhor ou igual, em relação ao tradicional, com nossa nova abordagem.

Alteramos o algoritmo de compressão do ETC para que, a partir de agora, ao se comprimir uma imagem avulsa, cada codebook seja testado. Então, selecionamos o *codebook* que apresentou melhor resultado (maior PSNR) e escrevemos esta informação no cabeçalho da imagem comprimida. Quando tentamos descomprimir a imagem o ETC simplesmente lê no cabeçalho da imagem qual codebook utilizar. É válido lembrar que isto se trata apenas de uma simulação. Em *hardware* a imagem seria descomprimida paralelamente com cada um dos sete *codebooks* disponíveis e apenas a saída correta seria escolhida por um multiplexador.

A figura 5.1 mostra o primeiro teste da nova abordagem do *iPACKMAN* com uma pequena imagem conhecida em Computação Gráfica como *Lorikeet* (128 x 128 pixels). A imagem comprimida com o *codebook* “original” apresentou PSNR igual a 29.889702 dB enquanto que com o *codebook* “outras” em “Naturais” (escolhido automaticamente pelo programa) o PSNR foi 29.926978. Na figura 5.2 mostramos um detalhe das imagens para que se perceba com mais clareza a sutil diferença visual entre as duas. Percebemos traços mais macios e transições de cor mais suaves.

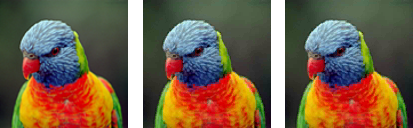


Figura 5.1. Teste com imagem *Lorikeet*. Da esquerda para a direita: imagem original, comprimida com codebook “original” e comprimida com codebook “outras”. Um ganho de 0.03727 dB.



Figura 5.2. Mesmo teste com imagem *Lorikeet*. Exibindo detalhe da imagem para mostrar o pequeno ganho local de qualidade na imagem.

A figura 5.3 exibe o mesmo teste aplicado a outra imagem conhecida em Computação Gráfica: *Lena* (128 x 128 pixels). Desta vez, pequenos ganhos de qualidade já podem ser percebidos na imagem em tamanho natural.



Figura 5.3. Testando a imagem *Lena*: imagem original, comprimida com *codebook* “original” (32.919285 dB) e comprimida com *codebook* “humanos” (33.044304 dB). Ganho de 0.125019 dB.

Algumas vezes a imagem a ser comprimida, embora teoricamente pertença a uma determinada classe, escolhe um *codebook* diferente. Na figura 5.4, aplicamos uma textura ao compressor. Porém, o melhor resultado foi obtido com o *codebook* otimizado para paisagens. O mais importante aqui é que este novo *codebook* foi capaz de diminuir o erro para esta imagem. O mesmo acontece para a imagem da figura 5.5.



Figura 5.4. Uma textura cujo erro foi minimizado com o *codebook* otimizado para paisagens. O PSNR aumentou de 42.108746 dB para 42.325431 dB. Ganho de 0.216685 dB.



Figura 5.5. Testando uma imagem da classe “Outras” em “Artificiais”: imagem original, comprimida com *codebook* “original” (36.383313 dB) e comprimida com *codebook* “Computação Gráfica” (melhor resultado, PSNR = 36.464738 dB). Ganho de 0.081425 dB.

Um teste com uma imagem da classe “Textos” é apresentado a seguir na figura 5.6. Esta classe costuma ser mais afetada pelo *codebook* genérico do *iPACKMAN*.

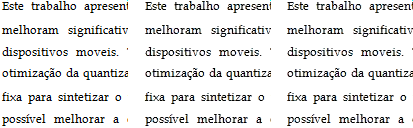


Figura 5.6. Imagem da classe “Textos”. Erro minimizado com o *codebook* da mesma classe aumentando o PSNR em 0.216685 dB, de 42.108746 dB para 42.325431 dB.

Na figura 5.7 é possível observar uma melhora significativa nas transições de cor no céu da paisagem, com o PSNR subindo de 31.849627 dB para 31.939323 dB.



Figura 5.7. Imagem da classe “Paisagens”. Erro minimizado com o codebook da classe “Outras” aumentando o PSNR de 31.849627 dB para 31.939323 dB. Diferença de 0.089696 dB.

Mais alguns testes são mostrados nas figuras 5.8, 5.9 e 5.10. As primeiras conclusões que pudemos tirar destes testes foram:

1. Nem sempre as imagens escolhem os *codebooks* de suas próprias classes;
2. Embora nenhuma imagem tenha escolhido o *codebook* original do *iPACKMAN* os resultados foram bastante próximos aos resultados originais.



Figura 5.8. Imagem de teste da classe “Computação Gráfica”. Erro minimizado com o *codebook* da mesma classe aumentando o PSNR de 37.463852 dB para 37.557753 dB. Aumento de 0.093901 dB de qualidade na imagem.

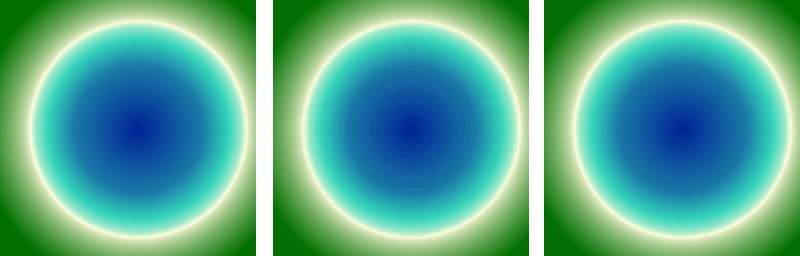


Figura 5.9. Outra Imagem da classe “Computação Gráfica”. Erro minimizado com o *codebook* da classe “Paisagens” aumentando o PSNR em 0.206129 dB, de 39.879183 dB para 40.085312 dB.



Figura 5.10. Imagem da classe “Outras”. Erro minimizado com o *codebook* da classe “Paisagens” aumentando o PSNR de 30.897773 dB para 30.950392 dB.

CAPÍTULO 6

Conclusões

# 6.1 Validação do *codebook* ótimo

O *codebook* da tabela 3.2 proposto por J. Ström [1] pode ser considerado ótimo, para o *benchmark* proposto pelo mesmo autor, para maioria dos fins práticos porque, mesmo executando durante vários minutos e centenas de vezes, o quantizador não foi capaz encontrar outro *codebook* muito melhor que o primeiro. A maioria não diferia muito em termos de valores de PSNR (nunca mais que 0.25 dB) em relação ao original. Por outro lado alguns codebooks ligeiramente melhores podem ser obtidos e o limite ótimo parece ser um dilema. Fica a cargo dos requisitos de qualidade da aplicação que utilizará *iPACKMAN* escolher qual *codebook* utilizar (até que ponto tentar otimizar).

# 6.2 Viabilidade da proposta de classificação das imagens

Tratando-se de quantização vetorial, classificar as imagens de treinamento em grupos de padrões semelhantes entre si parecia fazer sentido. Era razoável que o genérico não pudesse ter o mesmo desempenho que o particular em compressão de imagens. Porém, nossos experimentos demonstraram no capítulo 6 que, embora as tabelas obtidas para cada imagem sejam diferentes entre si e que utilizando classificação os valores de PSNR aumentem um pouco para imagens de propósito geral (as imagens de teste), nem sempre o ganho de qualidade é perceptível visualmente ou justifica o custo de replicar tabelas no *hardware*.

Novamente, a “universalidade prática” do *codebook* originalmente proposto poderá ou não ser assumida dependendo dos requisitos da aplicação.

# 6.3 Trabalhos futuros

Percebemos que, na compressão por classificação de imagens utilizando *iPACKMAN*, as imagens de uma determinada classe mostraram certa tendência em escolher o *codebook* da mesma classe, para o qual foi otimizado.

Existe um problema real em computação gráfica em se desenvolver classificadores de imagens: programas que sejam capazes de discernir a que grupo uma determinada imagem pertence [14]. Diversos modelos para este problema têm sido propostos, como em [15]. Uma possível e interessante experiência seria testar a nova abordagem do *iPACKMAN* para este inusitado objetivo e avaliar seu desempenho, ou seja, sua capacidade em separar imagens em classes arbitrárias de acordo com a escolha do *codebook* ótimo.

# Referências bibliográficas

[1] J. Ström, T. Akenine-Möller: 'iPACKMAN: High-Quality, Low Complexity Texture Compression for Mobile Phones', Graphics Hardware (2005), ACM Press, pp. 63-70.

[2] B. Marques, R. Santos, M. Walter, M. Campos: 'Um Estudo de Aplicação do iPACKMAN para Compressão de Texturas', Encontro Regional de Matemática Aplicada e Computacional (2007), VII ERMAC UFPE, Anais digitais.

[3] T. Aila, V. Miettinen, P. Nordlund: 'Delay streams for graphics hardware', ACM Transactions on Graphics 22 (2003) 792–800.

[4] M. Delp: 'Image compression using block truncation coding', IEEE Transactions on Communications (1979) 1335–1342.

[5] O. Hu, L. Raunheitte: 'Padrão JPEG de compactação de imagens'. Revista Mackenzie de Engenharia e Computação (2000). Ano 1 - Número 1. São Paulo.

[6] J. Ström , T. Akenine-Möller: 'PACKMAN: texture compression for mobile phones', ACM SIGGRAPH (2004), Sketches, August 08-12, Los Angeles, California.

[7] Y. Linde, A. Buzo, R. Gray: 'An Algorithm for Vector Quantizer Design', IEEE Transactions on Communications (1980), vol. 28, n.1, pp. 84-94.

[8] F. Brayner: 'Compressão de Texturas Utilizando Síntese de Texturas', Dissertação de Mestrado, Pós graduação em Ciência da Computação - UFPE (2009).

[9] A. Beers, M. Agrawala, N. Chaddha: 'Rendering from compressed textures'. In Proc. SIGGRAPH (1996), ACM press, pp 373–378.

[10] O. Konstantine, N. Krishna, H. Zhou: 'System and method for fixed-rate block-based image compression with inferred pixel values', us patent 5956431 (1999).

[11] Microsoft Directx: Advanced graphics on windows, 2008 http://msdn.microsoft.com/en-us/directx/default.aspx.

[12] D. Ivanov, Y. Kuzmin: 'Color distribution - a new approach to texture compression. Comput. Graph. Forum', 19(3):283–290(8), (2000).

[13] S. Fenney: 'Texture Compression using Low-Frequency Signal Modulation', Graphics Hardware (2003), ACM Press, pp. 84-91.

[14] T. Hermes, A. Miene, O.Moehrke: 'Automatic texture classification by visual properties', Classification and Information Processing at the Turn of the Millennium (1999), Springer Berlin.

[15] F. Wang, M. Kan: 'NPIC: Hierarchical Synthetic Image Classification Using Image Search and Generic Features', Lecture Notes in Computer Science (2006), Springer Berlin, ISBN 978-3-540-36018-6.