PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

PESQUISA, ESCOLHA E IMPLEMENTAÇÃO DE UM ALGORITMO DE COMPRESSÃO DE IMAGENS PARA CADRG EM C++

VIRGILIO MARCOS TORT PEIXOTO

Trabalho de Conclusão II apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Marcelo Cohen

Porto Alegre
2015

DEDICATÓRIA

Dedico este trabalho aos meus pais Raul Tort Peixoto e Lourdes Marcos Peixoto pelo seu apoio incondicional à minha formação.

“Share your knowledge. It’s a way to achieve immortality.”
(Dalai Lama)

AGRADECIMENTOS

Ao Bruno Schwab por ter dado a grande ideia de um tema relacionado ao trabalho!
Ao Dr. Marcelo Cohen pela paciência e por toda a orientação.

PESQUISA, ESCOLHA E IMPLEMENTAÇÃO DE UM ALGORITMO DE COMPRESSÃO DE IMAGENS PARA CADRG EM C++

RESUMO

Bases de dados de imagens georreferenciadas consomem muito espaço quando armazenadas em formato raw. Imagens comprimidas consomem uma capacidade de processamento valiosa ao serem lidas nos computadores de bordo das aeronaves e veículos civis e militares. Para resolver ambos problemas surgiu o padrão CADRG, que exige uma imagem comprimida por QV e palheta de cores. Este trabalho propõe uma implementação de código aberto para a compressão de uma imagem com os parâmetros especificados no CADRG.

Palavras-Chave: Quantização Vetorial, QV, CADRG, algoritmo LBG.

RESEARCH AND IMPLEMENTATION OF AN IMAGE COMPRESSION ALGORITHM FOR CADRG IN C++

ABSTRACT

Georeferenced image databases use a lot of space when stored in raw format. Compressed images require a valuable processing power when read in onboard computers in civilian and military vehicles and aircraft. CADRG standard was created to solve both problems and requires an image compressed using vectorial quantization and color palette. This work proposes an open source implementation to compress an image with the parameters specified in the CADRG standard.

Keywords: Vectorial Quantization, VQ, CADRG, LBG algorithm.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

CADRG – Compressed ADRG

ADRG – ARC Digitized Raster Graphics

NIMA – National Imagery and Mapping Agency

ARC – equal Arc-second Raster Chart/map

LBG – Linde-Buzo-Gray algorithm

QV – Quantização Vetorial

VQ – Vectorial Quantization

RGB – Red, Green and Blue storage format for images

SUMÁRIO

1. INTRODUÇÃO

O objetivo deste capítulo é expor a ideia geral deste trabalho, bem como clarificar sua finalidade e os motivos que levaram ao seu desenvolvimento.

1.1 Contextualização, Motivação e Objetivos

ADRG [NIM90] é um padrão militar aberto que define um formato de arquivos de imagens digitalizadas georreferenciadas. Ele foi concebido pela *National Imagery and Mapping Agency* (NIMA), uma agência de inteligência dos Estados Unidos. Arquivos ADRG são mapas e gráficos que são digitalizados para a utilização em software de visualização de dados georreferenciados. Seu principal objetivo é prover um padrão para armazenar, distribuir e classificar um conjunto de dados de mapa que represente o mundo todo. O georreferenciamento desses dados se dá pela utilização do sistema ARC (*equal arc-second raster chart/map*), detalhado no documento do padrão ADRG.

CADRG [NIM94a] é um padrão militar aberto concebido pela NIMA que sucedeu o ADRG. Ele propõe que os dados *raster* do ADRG sejam comprimidos e que sejam organizados segundo o formato descrito pelo padrão RPF [NIM94b].

O formato CADRG é largamente utilizado há muitos anos em aplicações militares e civis para armazenamento de dados rasterizados. São os motivos dessa adoção:

- Padrão bem difundido e conhecido;
- Tamanho pequeno e determinístico;
- Fácil decodificação.

Hoje em dia parece não haver motivação aparente para desenvolver um trabalho baseado em um formato antigo (20 anos) e especifico como o CADRG, mas os motivos acima citados se tornam mais importantes dadas as condições de uso do mesmo.

A aviônica comercial de hoje, apesar de contar com constantes avanços tecnológicos, como por exemplo a incorporação de *tablets* para ajudar na navegação e substituir manuais impressos[McK12], ainda tem suas raízes em padrões e sistemas legados. O hardware utilizado nas aeronaves muitas vezes se limita a arquiteturas embarcadas que já são certificadas e bem conhecidas. Isso se deve ao fato de que é muito cara e morosa a certificação civil de novos dispositivos, processadores, protocolos e enfim tecnologias. Portanto pode-se considerar o hardware embarcado nesses veículos como sendo consideravelmente defasado em relação ao que se tem hoje nos dispositivos móveis e mais ainda em relação aos computadores *desktops*, *notebooks*, etc.

Além do hardware e tecnologia limitadas, é possível enumerar outros fatores importantes a serem considerados, como por exemplo o armazenamento de grandes extensões de mapas que devem ser suportados por estas aeronaves. Um dos motivos dessa necessidade advém do problema logístico que se revela ao ser necessário atualizar o banco de dados de uma aeronave, esteja onde ela estiver, cada vez que for necessário navegar por uma área não coberta pelo mapa atual. Essa atualização é cara pois necessita de pessoal especializado, equipamento especial e certificado e, por vezes, dada a localização remota em que a mesma se encontra, se torna um feito impossível em tempo hábil. Portanto, considerando como caso o Brasil, um país de 8,5 milhões de quilômetros quadrados, repleto de áreas de difícil acesso, a situação de atualização se agrava.

Para atender a esses requisitos de performance, software existente certificado e espaço de armazenamento o CADRG sugeriu a utilização de um algoritmo de compressão *lossy,*[1]de fator de até 55:1, com custo computacional baixo para a descompressão. O fator de 55:1 foi obtido por três etapas de compressão:

1. A primeira etapa consistiu em converter os segmentos de mapas já digitalizados existentes, com densidade de pixels variável ao longo do globo (a depender da latitude), para 1536x1536 pixels cada e densidade de pixels constante, de 169 pixels por polegada. Percebeu-se que os mapas não sofreram distorções significativas para a aplicação desejada. Ao fim dessa conversão obteve-se uma redução de 2.25:1;

2. No segundo estágio, reduziu-se a palheta de cores de 24 bits para 8 bits, tendo então um fator de 3:1;

3. No terceiro estágio, reduziu-se a imagem pela quantização vetorial, tendo um fator de 8.1:1 (incluindo cabeçalhos do arquivo).

Portanto, temos um fator total de redução de $2.25 \times 3 \times 8.1 \cong 55$. Esse fator somado aos motivos já citados o mantém como padrão de mercado sendo utilizado até hoje em produtos novos sendo lançados, como por exemplo a modernização do avião Bandeirante C95 da Embraer (ano 2012), modernização do C-130H da Lockheed Martin (ano 2000), a produção do Osprey V-22 (ano 2005), entre outros.

O problema encontrado com esse padrão, em especifico com os algoritmos de compressão da imagem é que não existe muita documentação ou implementação disponível para quem queira aprender a utilizar essa tecnologia. A maioria dos fabricantes implementa o padrão desenvolvendo código proprietário. Portanto o objetivo desse trabalho é o de pesquisar, escolher e implementar um algoritmo de compressão que se encaixe na especificação do CADRG, mas que seja aberto e disponibilizado como tal para livre utilização em um portal de projetos de software livre, no caso o *SourceForge*.

[1] *Lossy* neste contexto significa uma compressão com perdas, onde a imagem original verbatim não é mais recuperável, mas que as perdas de maneira geral são aceitáveis dado o tradeoff.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo introduz brevemente a teoria por trás das áreas de conhecimento que serão utilizadas para o trabalho. A quantização vetorial é o método de compressão de imagens que motivou este trabalho, pois é o que a especificação CADRG solicita. Linde-Buzo-Gray (LBG) e *Pairwise Nearest Neighbor* (PNN) são os dois algoritmos testados e comparados pelos artigos relacionados da época (ano 1994) e portanto serão os algoritmos estudados neste trabalho. Na primeira parte, os conceitos de quantização e quantização vetorial são introduzidos. Na sequência, temos uma generalização de como um método de compressão de imagens baseado em quantização vetorial opera. E finalmente temos a descrição dos algoritmos de geração de *codebook*[1].

2.1 Quantização Vetorial

No capítulo 5.1 de [GG92] temos que **quantização escalar** é o cerne da conversão analógico-digital. Na sua forma mais simples, um quantizador recebe um número e seleciona o valor mais próximo de um conjunto finito predefinido de números que o represente. Podemos dizer, então que, por exemplo, o truncamento de um número real para zero casas decimais depois da virgula é uma forma de quantização escalar.

Analogamente à quantização escalar, a **quantização vetorial** (QV) recebe blocos de pixels (vetores) de tamanho fixo e os substitui por um índice que corresponde a um bloco presente numa lista previamente computada, chamada de *codebook*. Essa redução, de um espaço vasto de possibilidades de blocos de imagens (todas as combinações de valores de pixel e posições dos valores) para um espaço limitado de índices de uma lista previamente computada, é o que caracteriza esse processo como uma compressão de imagem com perda. Isso porque ao reduzir as possibilidades de blocos a um espaço fixo de blocos, existe perda da informação original (por exemplo, de forma análoga à conversão de imagens RGB para palheta de cores). Este *codebook* pode ser usado para formar uma nova imagem, que será uma versão quantizada da imagem original, montada como se fosse um mosaico usando peças predefinidas do *codebook*. Podemos considerar esse processo de quantização de imagem como sendo uma compressão de imagens por quantização vetorial.

[1] Um *codebook*, no contexto da quantização vetorial é uma lista onde cada entrada é um segmento de imagem.

2.2 Compressão de imagens baseado em QV

Para entender a compressão de imagens usando QV temos que pensar na imagem como se ela fosse inicialmente um mosaico. Para uma imagem qualquer de 8x8 pixels, temos um mosaico de 64 peças, cada uma representada por um pixel.

No primeiro passo da quantização vetorial, aumenta-se o tamanho da peça, representando um numero maior de pixels, por exemplo 2. Na Figura 2.1 vemos este passo exemplificado, ao qual chamamos de Vetorização. Após a vetorização temos uma imagem 32 peças, que se generalizadas para qualquer imagem, podem ser 32 peças totalmente diferentes. Podemos notar que essas 32 peças de tamanho 2 acabam ocupando o mesmo espaço que as 64 peças iniciais de um único pixel.

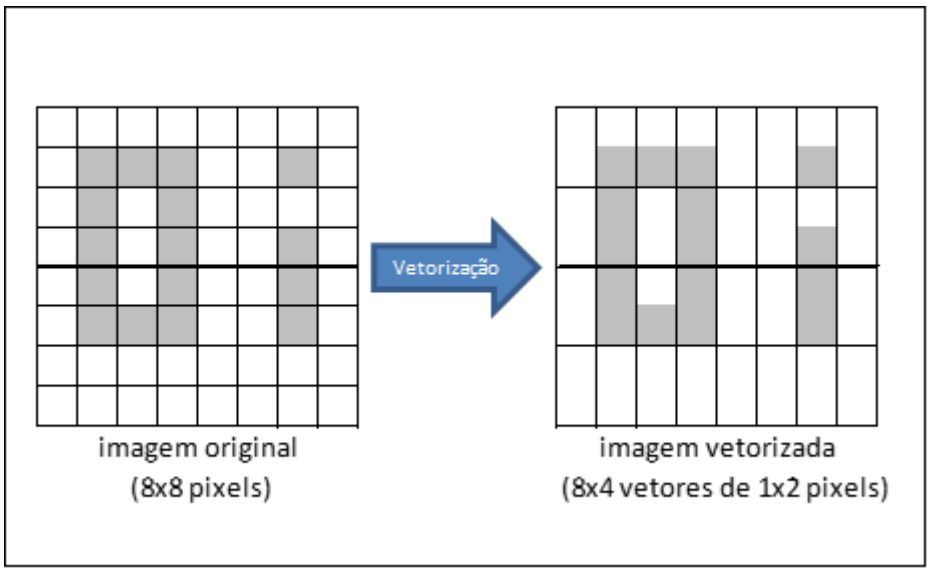


Figura 2.1 – Vetorização

No segundo passo da QV vemos a ideia principal dessa técnica ser aplicada: essencialmente, varremos a imagem a procura de similaridade entre as peças buscando produzir um conjunto reduzido de possibilidades. Vimos que após a vetorização podemos ter 32 peças distintas, uma para cada posição da imagem. Mas podemos reduzir essas possibilidades na imagem usada como exemplo a apenas 3, apenas verificando por similaridade entre as peças, como pode ser visto na Figura 2.2.

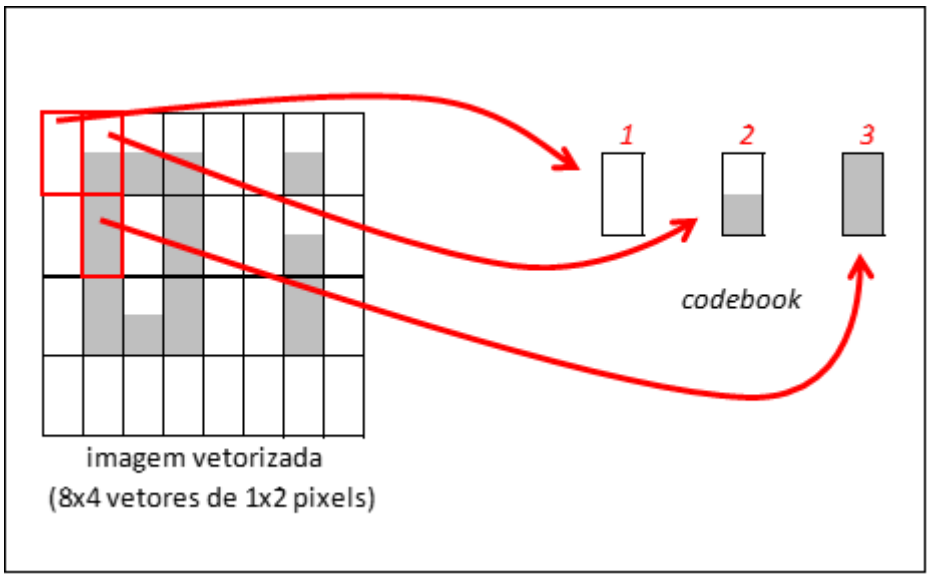


Figura 2.2 – Geração de *codebook*

No terceiro passo da QV temos que produzir uma matriz de índices, que informará a posição de cada peça nesse mosaico (analogamente ao gabarito de um quebra-cabeça). Em cada posição temos o índice que corresponde ao conjunto reduzido de possibilidades (que são apenas 3) calculado no segundo passo. Podemos ver essa matriz na Figura 2.3.

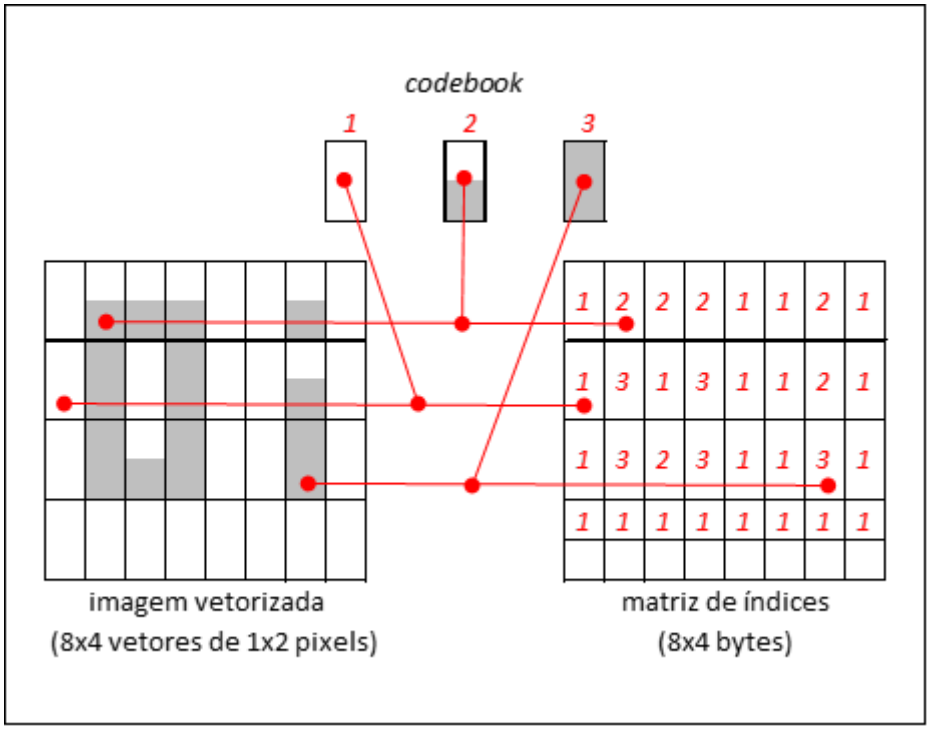


Figura 2.3 – Geração da matriz de índices

Assim ao fim do terceiro passo temos uma imagem comprimida com QV. A cada parte da imagem chamamos simplesmente de vetor, pois essa parte é composta por um grupo de pixels. A cada peça do conjunto reduzido de possibilidades chamamos de vetor

de treino, ou centroide. Esse nome se deve ao fato de que cada peça representa diversas partes do mosaico e é muito similar, senão igual, as partes que ela representa. A palavra "treino" vem do fato que precisamos treinar(modificar) os pixels desses vetores do conjunto reduzido para deixá-los mais parecidos as partes a que representam. Isso fica mais evidente nos sub-capítulos que se seguem, onde veremos os algoritmos de geração desses vetores de treino em detalhes. Ao conjunto de vetores de treino chamamos de *codebook*. Isso porque esse conjunto é de fato um "livro" de códigos (vetores de treino) ao qual a matriz de índices produzida no terceiro passo vai referenciar para cada parte da imagem.

De acordo com [BK97], diferentemente de métodos usados em formatos comuns como o JPEG, cujo processamento para codificar e decodificar uma imagem é similar, numa imagem resultante de Quantização Vetorial (QV) o esforço para decodificar é muito menor. Para uma imagem resultante de QV, basta montar a imagem usando o *codebook* de vetores de treino e a matriz de índices. Cada índice da matriz é substituído pelo bloco predefinido indexado e tem-se a imagem final.

Um algoritmo de codificação baseada em QV consiste nas seguintes etapas:

1. vetorização da imagem de entrada (Figura 2.1);
2. geração, a priori, de um bom[2] *codebook* baseado na imagem de entrada (Figura 2.2);
3. algoritmo que gera a matriz de índices, fazendo casamento de cada segmento da imagem com uma entrada do *codebook* (Figura 2.3).

Os dois últimos passos são executados conjugadamente em um algoritmo definido por uma técnica de geração de *codebooks*. Nas seções 2.2.1 e 2.2.2 serão tratadas duas conhecidas técnicas citadas nos trabalhos relacionados(Capítulo Chapter 3).

Nos algoritmos a seguir, serão explicadas duas maneiras de executar o segundo passo da QV. Como discutido anteriormente, esse passo consistem em verificar a similaridade do vetor de treino em relação a todas as partes da imagem, visando assim reduzir o espaço de possibilidades de vetores de treino diferentes. O algoritmo LBG vai reduzir esse espaço de possibilidades criando novos vetores de treino a cada iteração, de forma que um vetor é produzido (treinado) com base na média dos vetores mais similares entre si. O algoritmo PNN também cria novos vetores de treino a cada iteração, mas o faz "aglutinando" (agrupando, clusterizando) os vetores mais similares e levando em consideração o tamanho do grupo/*cluster* já existente.

[2]Bom significa representativo da imagem de entrada

2.2.1 Algoritmo LBG

A técnica definida por [LCCBG80] é baseada no algoritmo de Lloyd, explicado no capítulo 6.4 de [GG92]. O algoritmo de Lloyd consiste em encontrar conjuntos de pontos igualmente espaçados em sub-conjuntos de espaços euclidianos e partições desses sub-conjuntos conforme elucidado em [Wik14a]. Essas partições de sub-conjuntos são melhor descritas como diagramas de Voronoi, vide [Wik14b] ou [Wei14].

Supondo um espaço de coordenadas bidimensional, de limites 0 a 255 nos dois eixos, onde podemos considerar que cada dimensão representa a intensidade de cor de um pixel *grayscale*. Na Figura 2.4 vemos no *step1* o primeiro ponto que representa o primeiro vetor de treino (127,127), que é o centroide[3]. Este primeiro vetor tem todo o espaço de coordenadas dentro de seu grupo.

Partindo daí, podemos ordenar todos estes vetores (representados por pontos) em uma lista segundo a distância deles ao vetor de treino (ponto vermelho no centro da figura). Na Figura 2.5podemos ver um exemplo de como calcular distancias entre vetores. Nela, temos a mesma imagem vetorizada da Figura 2.1 mas agora com a intensidade dos pixels dentro de cada pixel. Assim, temos o vetor de treino, no caso (127,127) e calculamos a sua distancia em relação a alguns outros vetores.

Calculadas as distancias entre o vetor de treino e todos os outros vetores, temos a lista ordenada mencionada anteriormente. Então, dividimos esta lista em dois pedaços e são calculados os dois vetores de treino correspondentes, que são as respectivas médias aritméticas de cada um dos dois pedaços. Daí chegamos ao *step2* da Figura 2.4. Nele temos agora uma divisão entre dois grupos. O grupo1 dos pontos mais próximos ao vetor de treino 1 e o grupo2 dos pontos mais próximos ao segundo vetor de treino. Então ordenamos os vetores do grupo1 e do grupo2 e separamos ambos ao meio, como feito na ultima separação e por fim chegamos ao *step3*, onde temos 4 vetores de treino e quatro grupos. Entendido o processo, repetimos ele recursivamente até termos o número de vetores de treino desejado e ao fim, no *stepN*, cada vetor de treino vai ser uma entrada do *codebook*.

Essas regiões delimitadas pelas linhas que separam os pontos de um determinado grupo dos outros são partições definidas pela distancia dos vetores que as compõem até o seu vetor de treino. Cada vetor de uma partição está mais próximo do seu vetor de treino do que dos outros vetores de treino. No fim do processo, os pontos dentro de uma determinada região vão ser substituídos pela entrada do *codebook* que corresponde ao seu respectivo vetor de treino para finalmente montarmos a matriz de índices conforme ilustrado pela Figura 2.3.

[3]centroide é um ponto que representa a média aritmética de todos os outros pontos.

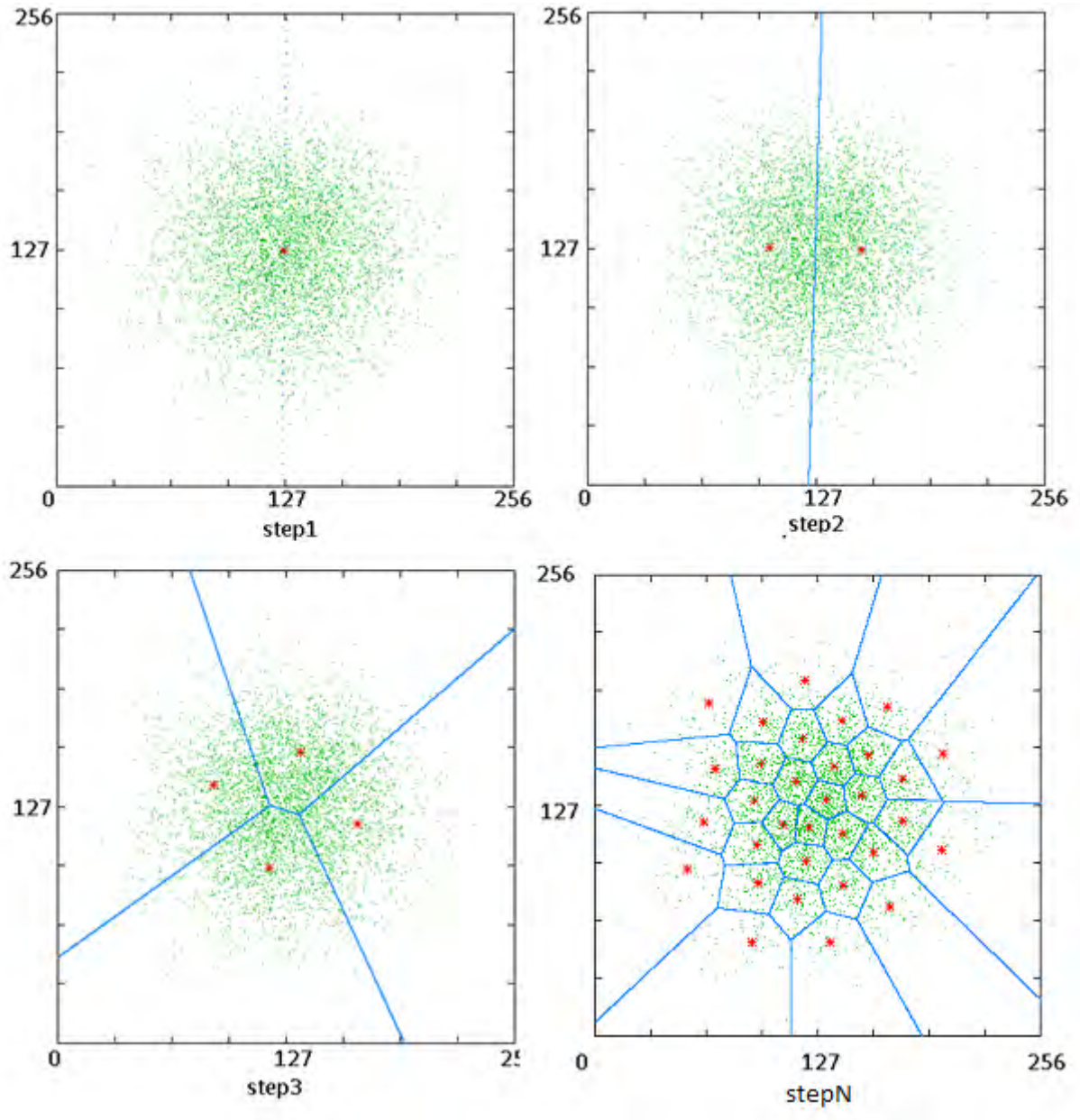


Figura 2.4 – Diagrama de Voronoi dos *steps* do algoritmo LBG (adaptado de [Dat14])

2.2.2 Algoritmo PNN

A técnica definida por [Equ89] expande os conceitos do LBG e propõe um método mais inteligente de agrupamento (clusterização), que pode ser visto na figura que o ilustra no próprio trabalho [4]:

Na Figura 2.6 vemos a esquerda uma série de pontos de entrada. Na figura da direita vemos que dois dos 6 pontos iniciais foram clusterizados[5], resultando em 5 pontos.

[4] as figuras do artigo original foram refeitas pois no artigo original não estavam em boa resolução

[5] clusterizar significa unir dois ou mais pontos de modo a representá-los a partir de então usando somente um único ponto

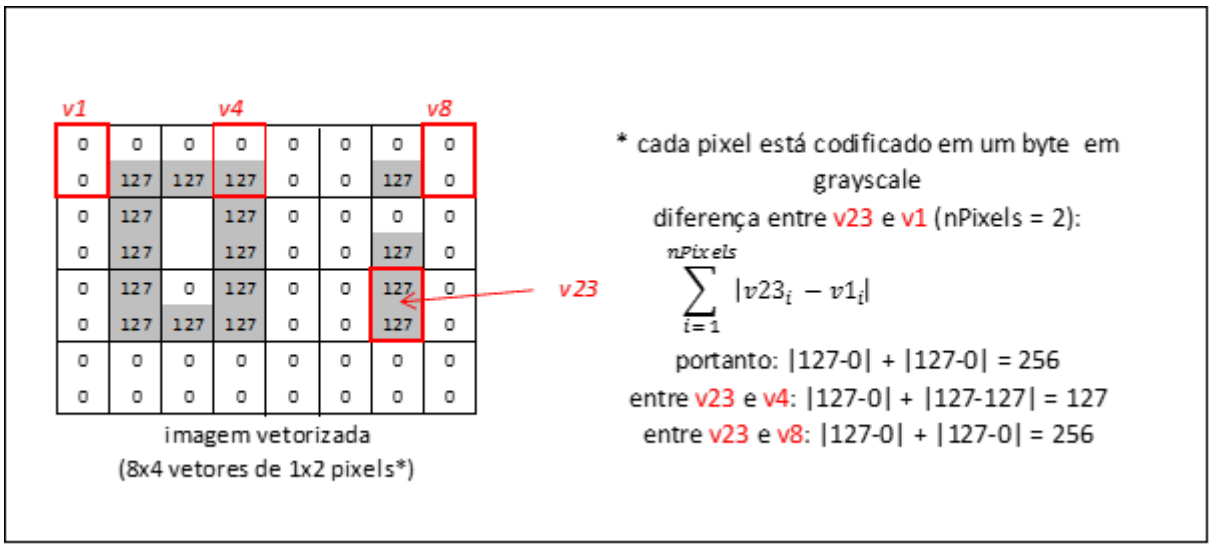


Figura 2.5 – Cálculo da distância entre os vetores

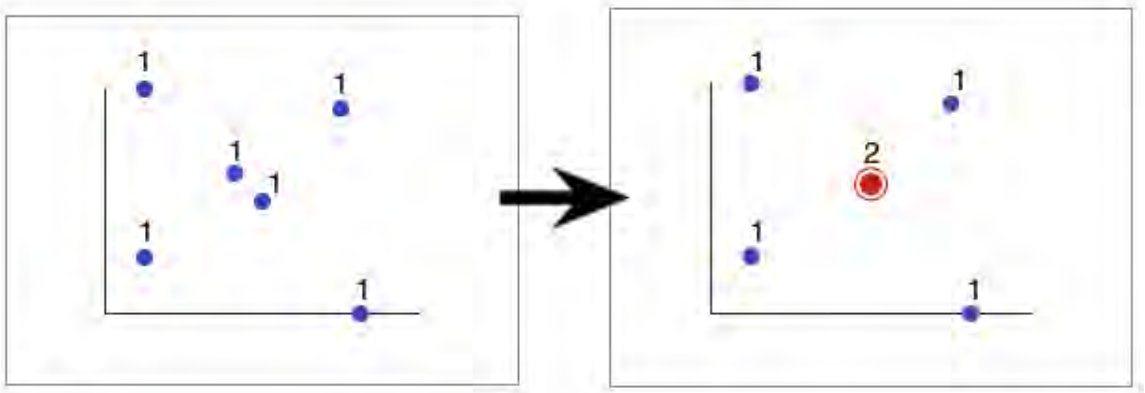


Figura 2.6 – Primeiro *Merge* no algoritmo PNN

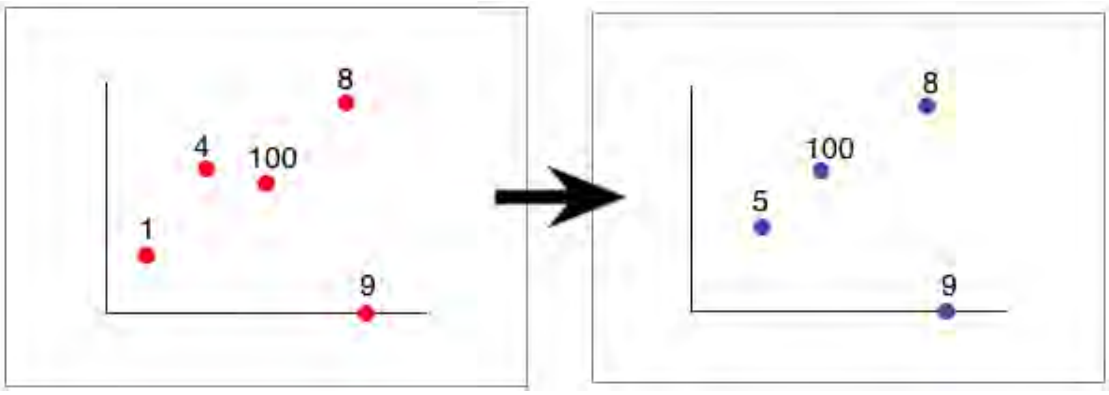


Figura 2.7 – *Merge* típico no algoritmo PNN

Isso aconteceu porque esses dois, naquele momento são os *nearest neighbors*, ou seja os vizinhos mais próximos. Assim funciona o PNN: a cada iteração, o algoritmo calcula quais são os dois pontos mais próximos dentre todo o espaço de pontos e os agrupa.

Na Figura 2.7, vemos que o algoritmo se utiliza de outras informações para declarar os vizinhos como de fato próximos. Ao invés de juntar o *cluster* de 4 pontos com o de 100 pontos, que estavam de fato mais próximos, decidiu por juntar os de 1 e o 4 pontos. Isso se deu pelo fato de o algoritmo considerar que *clusters* muito grandes acumulam também muito erro e portanto preferiu avaliar os de menor tamanho.

Outra abordagem seria usar vetores ao invés de pontos e usar o cálculo de menor distorção definido no LBG para definir os vizinhos mais próximos (como sendo os que tem menor distorção entre si.).

Por fim, podemos também extrapolar estes conceitos para a redução de cores. Podemos dizer que ao invés de um espaço bidimensional, possamos ter igualmente o processo de clusterização em um espaço tridimensional. Nesse último, podemos considerar cada dimensão como sendo um dos componentes RGB. Assim, traduzindo as fórmulas de cálculo de vizinhos mais próximos para esse último, poderíamos usar o PNN para clusterizar também as cores e assim diminuir o numero de cores em uma imagem e criar uma palheta reduzida.

3. TRABALHOS RELACIONADOS

Durante este capítulo serão apresentados alguns trabalhos relacionados, ou seja, que tenham uma linha de pesquisa semelhante este que será desenvolvido.

No primeiro trabalho[SCD92], o objetivo é resolver um problema das estações de planejamento de missão: armazenar e visualizar até 200GB de imagens no formato ADRG.

Já o segundo trabalho [MM94], visa analisar e otimizar as técnicas de compressão para CADRG.

3.1 *Compression of digitized Map Images*

[SCD92] propõe reduzir o espaço ocupado pelas imagens ADRG atuando em 3 componentes da imagem:

1. Dimensão: reduzindo o número de pixels da imagem original;
2. Espaço de cores: reduzindo a gama de cores disponível para codificar cada pixel;
3. Compressão: baseado na característica redundante de varias áreas dos mapas, comprimir a imagem.

3.1.1 Dimensão

As imagens utilizadas nas estações de planejamento de missão eram provenientes de mapas de papel. Essas imagens eram digitalizadas para o ADRG com densidade de pixels que acabava por ser 2.5 vezes maior que as dos displays utilizados. A partir disso o autor sugere que as imagens sejam reduzidas 2 vezes vertical e horizontalmente, o que já reduz em 4 vezes o tamanho da imagem armazenada.

O processo de digitalização dos mapas de papel conferia a imagem alguns artefatos e ruídos indesejados, pois poderiam afetar o redimensionamento dos dados posteriormente. Portanto se utilizavam filtros de tratamento de ruídos para remover os mesmos logo antes da compressão.

Após a aplicação dos filtros de ruídos o próximo passo descrito é o de redução de tamanho da imagem. Para tal [SCD92] comenta que os melhores resultados foram atingidos com o algoritmo de filtro cúbico (com parâmetros 1/3 e 1/3) proposto por [MN88].

3.1.2 Espaço de cores

Mapas são imagens que contem grandes quantidades de pixels com cores repetidas:

- Mapas que contenham somente esquemáticos, sem imagens de satélite, repetem as cores, pois tem por objetivo organizar as informações de forma a distinguir uma da outra. Assim estradas se diferenciam de rios, *airways* se diferenciam de demarcações territoriais, etc.
- Mapas que contém imagens de satélite também repetem cores: florestas densas, desertos rios e mares representam grandes extensões que podem ser representadas por uma quantidade reduzida de cores.

Portanto, o autor propõe que a representação corrente do ADRG, que é RGB 24 bits seja reduzida para uma representação diferente e que mais se aproxime do objetivo dos mapas de distinguir áreas e linhas. Então ele define que a representação de cores será em 8 bits, reduzindo assim o tamanho final da imagem em 3 vezes. A representação de cores escolhida pelo autor, dentre as diversas pesquisadas, foi a representação LAB[1] e o algoritmo para redução de cores teve um empate, entre duas das opções, ambas avaliadas como excelentes: PNN e *Greedy Seeding*.

O algoritmo *Greedy Seeding* para redução de cores consiste em:

1. Ordenar as cores por luminância;
2. Estabelecer uma tolerância para diferenças de luminância;
3. Escolher um pixel qualquer e adicionar na tabela de cores (inicialmente vazia);
4. Para cada pixel da imagem, achar o mais próximo na tabela de cores e:
   (a) Se o mais próximo tiver diferença maior do que a tolerância, adicionar este pixel na tabela de cores;
   (b) Na imagem final, adicionar este pixel referenciando o pixel mais próximo da tabela.
5. Ao fim, forma-se uma tabela de cores com luminâncias distintas (uma tabela reduzida de cores conforme a tolerância especificada).

[1] LAB (CIE L*a*b) é um espaço de cores onde os componentes de luminância e crominância são normalizados para a percepção humana.

3.1.3 Compressão

Da mesma maneira que mapas repetem dados em cores, também repetem certos padrões de pixels. Os mapas esquemáticos tem grandes áreas de cores sólidas e formas geométricas; e os baseados em imagem de satélite tem grandes áreas de lagos e oceanos e ainda apresentam certos padrões de relevo. Contando com isso o autor propõe a quantização vetorial como método de compressão. São citadas como vantagens:

- Tamanho determinístico da imagem de saída;
- Compressão rápida;
- Descompressão muito rápida.

Para a elaboração do algoritmo de geração do *codebook* o autor compara LBG com PNN e mostra que PNN é muito mais rápido, além de fornecer melhores resultados para imagens coloridas.

3.2 *Analysis of compression techniques for Common Mapping Standard (CMS) raster data*

O trabalho de [MM94] vem dois anos depois para complementar o de [SCD92]. Ele revisita e analisa todos os aspectos da compressão de imagens, acrescentando mais um requisito de software: não só é necessário suportar as estações de planejamento de missão mas também os sistemas de exibição no cockpit de uma aeronave. Esse novo requisito impõe novas restrições que incluem considerar o sistema de mapa em movimento dos displays e também a visibilidade dos dados do ponto de vista do piloto da aeronave dada a sua distancia em relação ao display.

O sistema usado no cockpit de uma aeronave é um sistema embarcado de poder de processamento inferior ao das estações de planejamento de missão. Ele tem estritas limitações de performance, tanto no carregamento dos dados quanto na descompressão das imagens exibidas. Ao mostrar o mapa ao piloto, e a sua localização no mapa, ele move o mapa na tela de maneira a carregar os novos arquivos de imagens a medida em que a aeronave se desloca. Assim, em tempo real (considerada a escala do mapa e a velocidade da aeronave), o mapa vai sendo exibido e o piloto percebe o deslocamento. Para que isso tudo aconteça, o tempo de carregamento das imagens antes do vôo deve estar de acordo com os procedimentos (6 minutos para cada 100MB) e também a complexidade da descompressão da imagem deve estar dentro das capacidades do sistema de mapa em movimento.

A visibilidade do piloto é essencial para a efetividade do mapa. Foram feitos diversos testes com pilotos e ficou estabelecido que a redução proposta por [SCD92], de 4:1 não resultava em um mapa suficiente legível ao fim do processo. Portanto [MM94] sugere adoção de 2.25:1 para obter uma relação aceitável entre compressão e legibilidade.

As demais conclusões de [SCD92], incluindo a escolha de quantização vetorial e do algoritmo PNN foram reiteradas. Ainda, devido aos requisitos de performance do mapa em movimento no cockpit, o artigo também testou e concluiu um tamanho de imagem ideal, de 1536x1536, dividida em *subframes*[2] de 256x256.

Todos os resultados desse trabalho vieram mais tarde a dar origem ao padrão CADRG, que define e solicita exatamente esse tamanho de imagem e a quantização vetorial como processo.

3.3 Comparação entre os trabalhos

O primeiro trabalho foi escolhido porque ele serviu de base para definir o problema de armazenamento da época e possíveis soluções. Já o segundo solucionou o problema propondo uma base para o novo formato de imagens especificando como comprimi-las. Ao longo dos dois trabalhos diversos algoritmos são testados e dentre os diversos, LBG e PNN se sobressaem por conta de sua performance e qualidade de resultados.

[2] *Subframes* são segmentos de imagem.

4. RECURSOS UTILIZADOS

Os seguintes recursos foram usados para implementação deste trabalho.

4.1 Software

Em termos de software foram utilizados:

- OSX Yosemite
- LyX 2.1.0
- *Global Mapper* http://www.bluemarblegeo.com/products/global-mapper.php, a ser usado na validação(veja capítulo 5.4)
- Gerador de Makefile Cross-Platform CMAKE 3.2.2 http://www.cmake.org/
- OpenCV 2.4.10 opencv.org

4.2 Ambiente de desenvolvimento

A linguagem escolhida foi C++ por apresentar maior flexibilidade de integração com bibliotecas em C (que é a linguagem adotada por diversos autores e implementações, inclusive [SCD92]). O ambiente de desenvolvimento foi OSX Yosemite.

4.3 Hardware

O hardware utilizado esta listado abaixo:

- Desktop
  - Processador i5 1.3GHz *dual-core*
  - Memória RAM 4 GB

5. PLANEJAMENTO

Neste capítulo é descrito o plano usado para a pesquisa e desenvolvimento dos algoritmos e do sistema integrador.

5.1 Objetivo

O propósito deste trabalho foi elaborar um software livre integrado que, a partir de um *bitmap* qualquer, gere uma imagem comprimida resultante, que se encaixe no padrão CADRG, ou seja, atenda aos seguintes parâmetros:

1. Tenha sido comprimida por quantização vetorial, gerando assim um codebook e uma matriz de índices;
2. O *codebook* tenha 4096 entradas;
3. Cada entrada seja um *kernel* de compressão[1] de 4x4 pixels;
4. As cores da imagem estejam codificadas em uma palheta de 216 cores RGB;
5. A imagem resultante deve ter as dimensões 1536x1536;

Originalmente o padrão define 8 bits de cores, totalizando 256 possibilidades. Porém ele solicita que 40 destas 256 cores sejam reservadas para o sistema.

5.2 Escopo

O escopo primário deste trabalho foi implementar o algoritmo de geração do *codebook* e o algoritmo de geração da imagem baseada no *codebook*.

Para a geração de *codebooks*, tinhamos o LBG e o PNN. O algoritmo LBG é mais popular dentre os artigos encontrados pelo *CiteSeerX*[2] no âmbito de compressão de imagens. O PNN, por sua vez, é o mais rápido e também o que mais tem artigos derivativos, com novas versões sugeridas como por exemplo o *fast-PNN*. Decidiu-se implementar o LBG por sua popularidade e por ser a base para todos os artigos de PNN, direta ou indiretamente. Assim o aprendizado e a implementação puderam focar na resolução do problema original de compressão com quantização vetorial e não nas otimizações dessa solução (que é o que o PNN oferece em cima do LBG).

[1] Um *kernel* de compressão é a definição do formato ou forma de uma entrada do *codebook*.
[2] http://citeseerx.ist.psu.edu

Para o algoritmo que gera a matriz de índices temos os candidatos *k-d tree*[3], *Classified* e *Full-Search* mencionados como os testados por [SCD92]. Eles são todos algoritmos de busca, que varrem cada vetor imagem original vetorizada e encontram o melhor índice do *codebook* para representá-la. Como o algoritmo *Classified* é bem mais sofisticado (leva em conta características da imagem necessitando algoritmos para classificá-las assim), e o *k-d tree* é complexo (envolve balanceamento de árvores multi-dimensionais), optou-se pelo *Full-Search* por uma questão de tempo e escopo.

O escopo secundário, porém não menos importante, é o arcabouço de software necessário para viabilizar a utilização do algoritmo que consiste de:

1. uma biblioteca pronta e livre para a leitura de imagens;
2. uma biblioteca pronta e livre para redução do espaço de cores;
3. um software a ser desenvolvido que integre as bibliotecas e os algoritmos do escopo primário e quaisquer outros componentes específicos e APIs para gerar o resultado especificado no objetivo 5.1.

Foi também cogitado um módulo para redimensionamento da imagem. Porém os frames CADRG já são pré-recortados do mapa com a dimensão certa para uma determinada escala para um determinado quadrante georreferenciado. Assim mudar as dimensões implicaria em mudar a escala. Entendeu-se então que essa operação de redimensionamento está fora do escopo e não acrescentaria uma funcionalidade relevante ao software.

Para os itens 1 e 2, será utilizada a biblioteca OpenCV por sua popularidade e diversidade de tutoriais na internet. Ela inclui suporte para diversos espaços de cores e oferece a possibilidade de trabalhar também com espaços reduzidos (tabelas de *lookup*).

5.3 Arcabouço de Software

O arcabouço de software compreende tudo aquilo que deve ser implementado para que a imagem resultante do algoritmo se transforme em uma estrutura que atenda a especificação de imagem CADRG. Não está incluído aí um gerador de arquivos CADRG reais e sim uma estrutura que disponha dos ingredientes para que o mesmo seja gerado posteriormente. Essa estrutura deve conter:

- Uma paleta de 216 cores RGB;
- O *codebook*, cujas entradas tenham dimensões 4x4 pixels e cada pixel referencie uma cor da paleta;

[3] *k-d tree é uma árvore onde cada nodo não-folha abaixo da raiz é um ponto multidimensional.*

- Uma "imagem", que na realidade é uma matriz de 384x384[4] pixels que referenciam o *codebook*.

5.4 Validação do Software

Para validar se o software final (algoritmos + Arcabouço) funciona da maneira esperada, a imagem resultante será salva em um *bitmap* e a tabela de cores será gerada em um arquivo de saída CSV (*comma separated values*).

Foi cogitado inicialmente um módulo para inserção da imagem gerada em um *frame* CADRG qualquer. Assim, para validar o software final, poderíamos simplesmente abrir o arquivo resultante dessa inserção no *Global Mapper*. Porém não foi encontrado um *frame* CADRG disponível livremente na internet para poder ser modificado. Então para validar a imagem usando o *Global Mapper*, os standards RPF e CADRG teriam que ser implementados, com ao menos um mínimo *subset* de *headers*. Essa implementação demandaria tempo além do cronograma e portanto ela não foi efetuada. Concluiu-se que o impacto desta funcionalidade é mínimo pois a um software que já suporte CADRG basta incorporar os outputs do algoritmo atual (matriz de correspondência, *codebook* e tabela de cores). Ainda, caso tivéssemos um *header* mínimo, este provavelmente seria diferente do implementado pelo utilizador, pois os *headers* são variáveis e suportam uma extensa gama de *payloads* de dados e georreferenciamento.

[4] 384 pixels do lado da matriz multiplicados por 4 pixels da entrada do *codebook* resultam em 1536 pixels, que é a dimensão do lado da imagem final.

6. PROTÓTIPO

6.1 Algoritmos utilizados

6.1.1 Redução de cores

Para satisfazer o requisito do CADRG de termos 216 cores, é preciso uma abordagem especifica. As abordagens mais populares para reduzir o numero de cores consistem em transformar o espaço de cores existente em um outro codificado em menos bits. Um exemplo disso é a conversão de imagens RGB em escalas de cinza (24 bits para 8 bits). Ela oferece uma boa representação da imagem, mas não serve o propósito deste trabalho. A escolha, no CADRG, de uma paleta de cores tem um motivo: Ela se baseia no fato de uma determinada imagem topográfica não utilizar todas as cores dos 24 bits.

Na sua maioria, as imagens CADRG descrevem um relevo que, no escopo de um *frame*, contém um número limitado de cores. E essa característica faz com que a redução de cores a uma paleta limitada não impacte drasticamente na qualidade da imagem final. O algoritmo encontrado para resolver o problema de palheta de cores foi o *k-means*. Ele já está disponível na biblioteca *OpenCV* e também é um algoritmo de quantização vetorial baseado em *clustering* assim como o PNN.

O *k-means* permite especificar exatamente o número de cores desejado, o critério de parada e o método desejado de escolha das cores iniciais. Existem 2 possíveis critérios. O primeiro é o número de iterações do algoritmo e o segundo é a precisão das cores resultantes (distancia máxima desejada entre a cor de qualquer pixel resultante e a cor mais próxima dele). Quanto as cores iniciais existem 3 maneiras possíveis: fornecer as cores iniciais, usar um algoritmo especifico de obtenção das cores ou o método aleatório, que consistem em escolher as cores iniciais baseado em pixels da imagem escolhidos aleatoriamente.

Neste projeto foi escolhido como critério de parada o número de iterações e foi especificada apenas uma iteração. Quanto as cores iniciais, foi escolhido o método aleatório. Estas opções foram escolhidas pois nessa combinação produzem resultado em menos de 5 segundos. Qualquer combinação das outras alternativas produzem resultados em tempos entre 30 segundos e 5 minutos. Como a redução de cores não é escopo primário deste trabalho, optou-se por essa combinação.

6.2 Algoritmos desenvolvidos

6.2.1 Vetorização

Como primeiro passo da quantização vetorial temos a vetorização da imagem de entrada. Para a imagem de 1536 x 1536 pixels temos uma matriz de 384 x 384 vetores de 16 pixels cada (com kernel de 4x4 pixels). Como não precisamos trabalhar com uma matriz de vetores e sim com todos eles individualmente, assim como mostrado capitulo 2.2(Compressão de imagens baseado em QV), indexamos os vetores de 1 a 384^2. A matriz de pixels é obtida da biblioteca OpenCV como uma sequencia de 1536^2 pixels. Assim, para obter gerar os vetores, em essência precisamos traduzir essa sequência de pixels gerando 384^2 vetores.

Para executar essa tradução de pixels em vetores, temos o algoritmo (6.1). Nele estão descritas as estruturas utilizadas, em linguagem C++.

```
typedef struct { unsigned char b[3]; } PIX;
typedef struct { PIX pixel[16]; } VEC;
enum SENTIDO {IMAGEM_PARA_VETOR, VETOR_PARA_IMAGEM};
VEC vetores [384*384];
PIX img [1536*1536];

traduz(VEC vetor, PIX img[], int vt, SENTIDO sentido)
{
    linha = vt/384;
    coluna = vt%384;

    for(px = 0; px < 15; px++)
    {
        x = coluna*4 + px/4;
        y = linha*4 + px%4;

        unidimensional = y*1536 + x;

        if (sentido == IMAGEM_PARA_VETOR)
            vetor.pixel[px] = img[unidimensional];
        else
            img[unidimensional] = vetor.pixel[px];
    }
}
```

Algorithm 6.1 – Vetorização

Primeiramente definimos um pixel como sendo um array de 3 bytes. Então definimos um vetor, que é a representação serial do kernel de 4x4 pixels. Cada linha da matriz do kernel é disposta uma após a outra, totalizando 16 pixels no array. Depois define-se o sentido da conversão, que pode ser do vetor para a matriz ou vice-versa. Por último, temos a definição da imagem.

No algoritmo em si, o primeiro passo é traduzir o índice do vetor para linhas e colunas da matriz de vetores. Em seguida, executamos o loop principal do algoritmo para cada pixel do vetor de entrada. Dentro do loop convertemos as linhas e colunas em coordenadas da matriz de pixels. Depois convertemos essas coordenadas x e y em uma coordenada unidimensional para acessar a imagem de entrada. Por fim, copiamos o pixel para a matriz ou vice-versa de acordo com o sentido especificado.

Para traduzir a imagem inteira executa-se o algoritmo para todos os vetores.

6.2.2 Geração de um *codebook*

Para gerar o codebook, temos um método chamado divide_e_encontraVetorDeTreino, descrito no algoritmo 6.2. Ele recebe como parâmetros a lista de vetores, a profundidade e a imagem de saída.

```
struct COMP_STRUCT { int quanta; int indice; };
std::list<COMP_STRUCT> vetList;
PIX img[1536*1536];

divide_e_encontraVetorDeTreino(std::list<COMP_STRUCT> vetList,
    int profundidade, PIX img[], VEC vetores[])
{
    VEC treinador = obtemVetorDeTreino(vetList, vetores);

    if (vetList.size() < 4 || profundidade == 0)
    {
        consertaCor(treinador);
        codebook.insere(treinador);

        for (int i=0; i < vetList.size(); i++)
            traduz(treinador, img, item.indice,
                    VETOR_PARA_IMAGEM);
    }
    else
    {
        for (int i=0; i < vetList.size(); i++)
        {
            item.quanta =
                comparaVetores (vetores[item.index], treinador);
        }

        vetList.sort();

        divide_e_encontraVetorDeTreino(LEFT(vetList),
                                       profundidade - 1);
        divide_e_encontraVetorDeTreino(RIGHT(vetList),
                                       profundidade - 1);
    }
}
```

Algorithm 6.2 – Geração do *codebook*

O algoritmo pode ser descrito em essencialmente 4 passos:

1. obtém um vetor de treino baseado na lista de vetores recebida;

2. para a recursão de acordo com a condição de parada e adiciona o vetor de treino no *codebook*;
3. ordena a lista segundo a diferença dos ítens em relação ao vetor de treino;
4. divide a lista ao meio em duas novas listas e roda o algoritmo novamente.

A ordenação no passo 3 é feita utilizando o método *sort* de listas da *std lib*.

Para realizar este algoritmo precisamos de 3 estruturas: a primeira estrutura guarda o índice dos vetores e a quantidade de diferença entre eles. A segunda é uma lista da primeira, para guardar as diferenças de todos os vetores. A terceira é a imagem de saída, do mesmo tipo definido para a imagem do algoritmo 6.1.

No algoritmo 6.2 descrito temos algumas funções auxiliares:

- *obtemVetorDeTreino*: é a função que calcula o vetor de treino para uma dada lista de comparação;
- *consertaCor*: troca a cor dos pixels para uma das 216 cores da palheta;
- *comparaVetores*: compara dois vetores e retorna a quantidade de diferença entre eles (quanta);
- *LEFT*: divide uma lista pela metade e retorna a metade da esquerda;
- *RIGHT*: divide uma lista pela metade e retorna a metade da direita.

A chamada inicial desse algoritmo e com uma lista de todos os vetores e profundidade 12, pois 12 iterações geram 2^{12} vetores no *codebook*, atendendo a especificação do CADRG.

6.2.3 Vetor de treino

Para a obtenção do vetor de treino, fazemos um "vetor médio" a partir de uma lista de vetores, conforme descrito no algoritmo 6.3. Esse vetor é calculado fazendo-se a média, individualmente, para cada um dos pixels, cada um em sua posição. Para cada pixel, fazemos a média de seu componente RGB. Abaixo temos um código que ilustra esse comportamento. No primeiro loop acumula-se o valor de cada componente RGB. No segundo loop faz-se a média desse valor.

Essa foi uma estratégia escolhida neste trabalho. Assim como no capítulo 2.2 usamos um valor médio na escala de cinzas como vetor inicial (127), aqui fazemos o mesmo. Porém, aplicando este conceito a cada componente de cor, e com base num conjunto de vetores específico passado como argumento.

```
VEC
obtemVetorDeTreino (std::list<COMP_STRUCT> vetList,
    VEC vetores[])
{
    int media[16][3];
    int tam_lista = listaDeVetores.size();
    VEC retVet;

    for (int vt=0; vt < tam_lista; vt++)
        for (int px=0; px < 16; px++)
            for (int comp=0; comp < 3; comp++)
                int index = vetList[vt];
                media[px][comp] +=
                    vetores[index].pixel[px].b[comp];

    for (int px=0; px < 16; px++)
        for (int comp=0; comp < 3; comp++)
            retVet.pixel[px].b[comp] =
                    media[px][comp] / tam_lista;

    return retVet;
}
```

Algorithm 6.3 – Vetor de treino

6.2.4 Atribuição da palheta de cores

O algoritmo de atribuição da palheta de cores é feito com força bruta: Para cada pixel do vetor fornecido, calcula-se a distancia deste pixel até todas as cores da tabela de cores e escolhe-se a que tem menor distância. A distância é calculada somando-se as diferenças absolutas entre a intensidade de cada componente de cor R,G e B. Assim funciona o método "consertaCor", modificando o vetor de treino encontrado para conformar com as cores da palheta.

6.2.5 Comparação de vetores

Para comparar os vetores, a função "comparaVetores" utiliza a fórmula descrita na figura 2.5.

6.3 Exemplo de QV passo a passo

Conforme visto no Capítulo 6.2.2 a geração do *codebook* foi implementada neste trabalho em quatro passos. Eles serão demonstrados aqui tomando como exemplo a figura usada no Capítulo 2.2.

O primeiro passo consiste em vetorizar a imagem de entrada. Podemos ver na Figura 6.1 a imagem vetorizada (vetores de 1 a 32) com os valores correspondentes a intensidade dos seus pixels codificados em *grayscale*. O valor 255 representa intensidade máxima (cor branca) e o valor 127 representa intensidade média (cor cinza).

Figura 6.1 – Imagem vetorizada com valores dos pixels.

O segundo passo consiste em achar o vetor de treino para essa imagem. Para isso calcula-se a média de todos os pixels em cada uma das duas posições do vetor de treino em questão, como demonstrado na Figura 6.2.

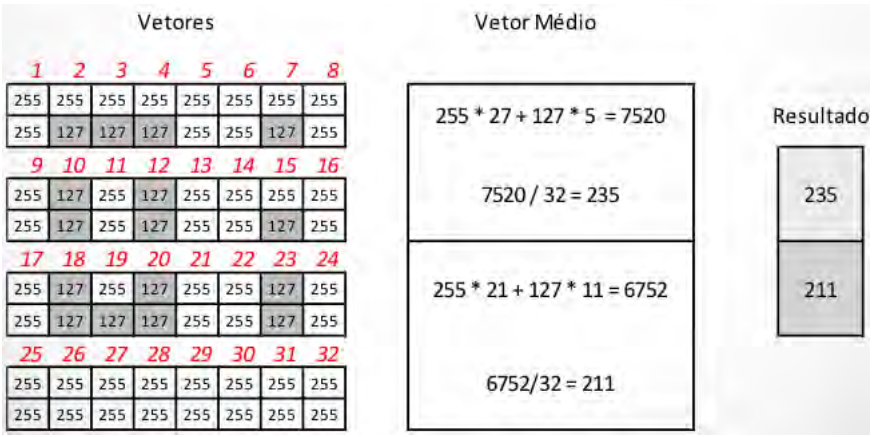


Figura 6.2 – Cálculo do vetor de treino

Neste cálculo vemos que para o primeiro pixel do vetor de treino, a cor branca aparece 27 vezes e a cor cinza 5 vezes. Já para o segundo pixel do vetor de treino a cor

branca aparece 21 vezes e a cor cinza 11 vezes. Ao fim temos o vetor de treino médio como sendo {235, 211}, ou seja, cinza claro para o primeiro pixel e cinza escuro para o segundo.

O terceiro passo consiste em ordenar os 32 vetores de acordo com a diferença entre eles e o vetor de treino. Primeiro então calculamos as diferenças. Como temos visivelmente somente 3 possibilidades, podemos escolher os vetores v1, v4 e v23 para calcular, como está ilustrado na Figura 6.3. Assim vemos que as diferenças podem ser os valores 64 (vetores totalmente brancos), 104 (vetores branco-e-cinzas) ou 192 (vetores totalmente cinzas).

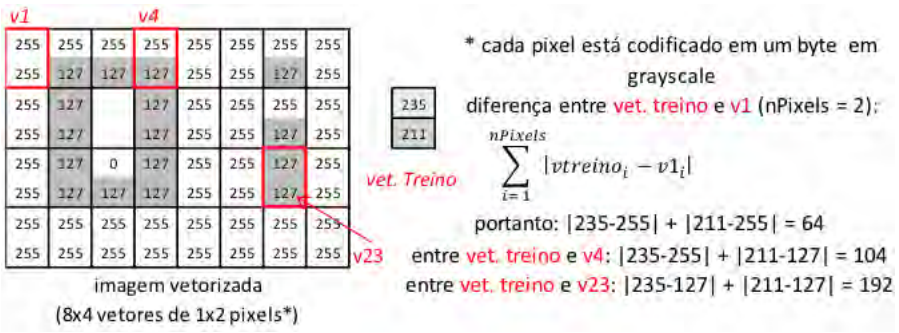


Figura 6.3 – Diferenças do vetor de treino

Com as distâncias calculadas, vamos a ordenação. Para tal, utilizamos uma estrutura chamada "array de {índex, quanta}". Essa estrutura serve para organizar os vetores (indexados no componente "índex") de acordo com seus pesos (especificados no componente "quanta"). Na Figura 6.4 temos esta estrutura preenchida com as diferenças. Ela inicialmente está na ordenada por índex, ou seja, na mesma ordem em que os vetores aparecem na imagem.

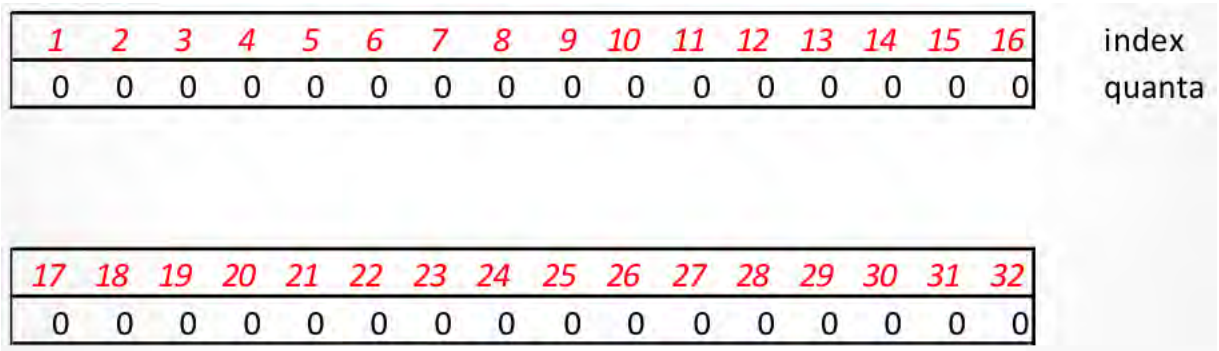
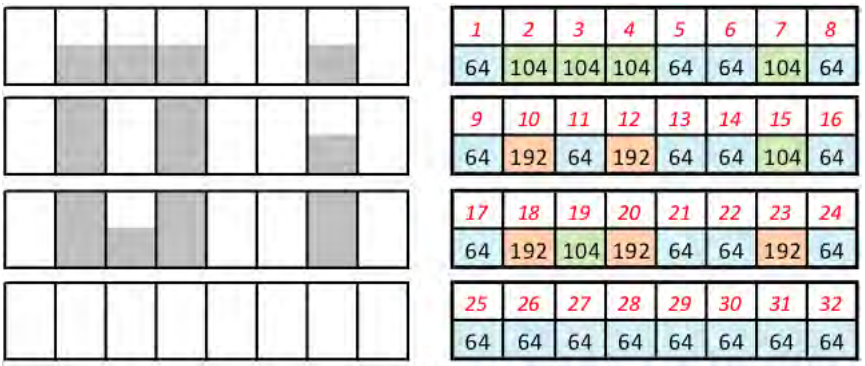


Figura 6.4 – Array de índex-quanta

Após o cálculo das distâncias, temos este array preenchido, como podemos ver na Figura 6.5. Por fim, esta estrutura é ordenada e temos como resultado a Figura 6.6. Nela o array esta representado em duas partes.

O quarto e último passo consiste em dividir o array em dois pedaços e chamar o algoritmo recursivamente para cada uma delas. No caso desta imagem, os dois pedaços correspondem exatamente as partes da Figura 6.6.



1	2	3	4	5	6	7	8
64	104	104	104	64	64	104	64

9	10	11	12	13	14	15	16
64	192	64	192	64	64	104	64

17	18	19	20	21	22	23	24
64	192	104	192	64	64	192	64

25	26	27	28	29	30	31	32
64	64	64	64	64	64	64	64

Figura 6.5 – Imagem original e Array preenchido

1	5	6	8	9	11	13	14	16	17	21	22	24	25	26	27
64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64

28	29	30	31	32	2	3	4	7	15	19	10	12	18	20	23
64	64	64	64	64	104	104	104	104	104	104	192	192	192	192	192

Figura 6.6 – Array de índex-quanta ordenado

6.4 Funcionamento do software

O funcionamento do código implementado (arcabouço) se dá da seguinte maneira:

```
./vvq imagem.bmp
```

O programa é chamado com o nome do arquivo da imagem de entrada como parâmetro. Ele então produz três arquivos: *"lut.csv"*, "*mat.csv*" e "*codebook.csv*". "lut.csv" é um arquivo separado por virgulas onde cada linha descreve uma cor com suas componentes RGB. "*mat.csv*", que é um arquivo onde cada linha representa uma linha da matriz e cada coluna é um valor separado por virgula. Por fim "*codebook.csv*" é um arquivo onde cada linha é uma entrada do *codebook* e os valores dos pixels são separados por vírgula na linha, assim como os valores RGB. Estes arquivos são formatados em modo texto para facilitar a sua manipulação em qualquer plataforma, sem se preocupar com alinhamentos de estrutura em memória, *endianness*, etc.

Opcionalmente o usuário pode embutir o core do software em seu projeto e usar as estruturas lá definidas (PIX e VEC, conforme o algoritmo 6.1). Para tal, basta usar as funções nesta ordem:

1. função de redução de cores (que gera a tabela de cores);

2. função de geração de *codebook* (que gera a matriz e gera o *codebook* com as cores indexadas pela tabela de cores).

Estas funções poderiam executar independentemente, pois ambas tem como entrada a imagem original sem alterações. Porém esta ordem é imposta para que a segunda já possa, ao criar o *codebook*, ajustar as suas cores para referirem-se a tabela de cores gerada. Esta imposição foi feita tendo em vista a performance. Já que o código para geração de *codebook* gera uma árvore pré-determinada de execuções, fica mais fácil modificar a cor no fim das execuções. Quando essa árvore chega na profundidade 12, na execução de cada folha dessa árvore, e logo antes de inserir o codebook as cores do vetor de treino são corrigidas para ter uma cor do *codebook*. Assim não há necessidade de uma busca posterior por cada vetor para mudar essas cores, pois ela já é feita antes de o vetor de treino ser inserido no codebook. Isso pode ser visto na linha 12 do algoritmo 6.2, quando a função "consertaCor" é chamada, antes da inserção no codebook.

7. RESULTADOS

Para testar o algoritmo foram utilizadas quatro imagens topográficas distintas para testar o algoritmo de compressão:

1. Mapa topográfico de uma região ao sudeste da França, obtido e recortado de http://i.imgur.com/nfgbYzU.jpg ;
2. Mapa topográfico de uma região do Chipre, obtido e recortado de http://photojournal.jpl.nasa.gov/catalog/PIA04965;
3. Mapa topográfico de uma região ao norte do Marrocos, obtido e recortado de http://photojournal.jpl.nasa.gov/catalog/PIA04965;
4. Mapa topográfico de uma região do planeta Mercúrio, obtido e recortado de http://photojournal.jpl.nasa.gov/catalog/PIA17385.

Essas imagens podem ser visualizadas sem recortes, juntamente com a versão comprimida no Anexo B. Aí podemos ver que não houve muita perda de dados. Para todas as imagens temos mantidas as características de relevo (ressaltadas com cores e sombreamento) bem definidas. Isso se deve ao fato de que as imagens estão redimensionadas para caber na página e sua alta resolução de 1536x1526 "restaura" as perdas da compressão quando a densidade de pixels aumenta.

7.1 Imagens lado a lado

Para obter mais detalhes, foram dispostas lado a lado no Anexo A. Agora sim podemos ver perdas de qualidade evidentes a olho nu. Na Figura A.1 é claro o padrão quadriculado nas montanhas sombreadas em tons de marrom na imagem da direita. Esse quadriculado não se trata de pixels e sim de *kernels* de compressão (entradas do *code-book*). Estes *kernels* se repetem para tentar imitar as montanhas sombreadas mas não tem muito sucesso. Isso porque o número de cores também é limitado pelas 216 entradas da palheta. Essa limitação do número de cores pode ser vista na transição drástica entre dois tons de verde na parte inferior da imagem no lado direito. Essa transição é suave no lado esquerdo e reflete exatamente a limitação imposta pelo número de cores da paleta.

Na figura A.2 também podemos observar os artefatos de compressão quadriculados na parte cinza da imagem. Outra característica que chama atenção é o fato de a imagem comprimida estar mais esverdeada do que a da esquerda. Na ilha na parte inferior da imagem vemos que a imagem da esquerda é mais amarela e a da direita mais verde. Isso foi ocasionado pela fato de existirem poucos tons de amarelo na imagem da direita,

forçando o algoritmo a escolher tons esverdeados para os tons amarelos mais claros da imagem.

Na figura A.3 temos um defeito bem perceptível no meio da imagem: dois retângulos verticais na cor cinza, parecendo uma peça de Tetris, onde a imagem deveria ser preta. Esse defeito ocorreu porque toda aquela área cinza na volta do defeito se dividiu em duas partes iguais. porém uma das partes iguais continha vetores muito claros e acabou puxando a média do vetor de treino para cima, gerando vetores muito discrepantes para aquela área específica.

Por fim, na última imagem, na figura A.4, vemos que o maior defeito está no círculo escuro na parte inferior direita da imagem. Ele contém diversos artefatos claros na versão comprimida. Esse fenômeno é o mesmo acontecido na figura A.3 comentada anteriormente. Como são poucos vetores com pixels pretos, em algum momento da compressão eles se juntarão a outros pixels claros e seu vetor de treino vai ficar mais claro no cálculo da média das cores.

Para ilustrar o progresso do algoritmo, foi elaborada a figura 7.1. Ela foi dividida em 14 retângulos verticais que representam da esquerda para a direita o algoritmo sendo executado com n iterações do algoritmo, cada uma dobrando o número de entradas no *codebook*. A primeira iteração gera 2 entradas, a segunda gera 4 e assim sucessivamente, passando por 12 iterações que geram as 4096 entradas desejadas e chegando até 14 iterações que geram 16384 entradas. Esse progresso pode ser visto em 14 imagens uma por passo no Anexo C.

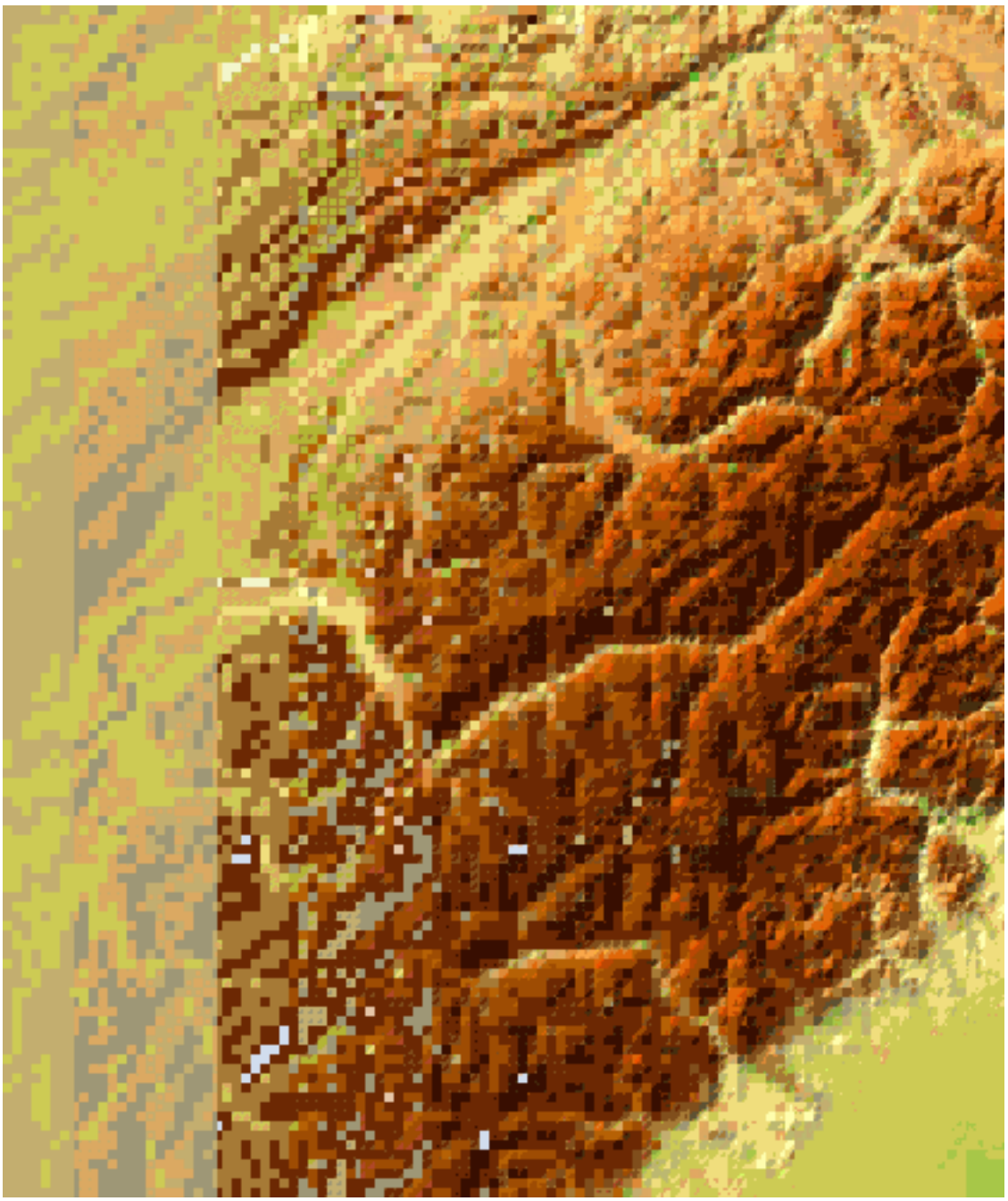


Figura 7.1 – Progresso das iterações do algoritmo. De 1 a 14 iterações, da esquerda para a direita.

7.2 Tempo de conversão

O tempo de conversão total é de em média 7,3 segundos. São 2,8 segundos fixos do algoritmo de geração de *codebook* (que leva o mesmo tempo para qualquer imagem de 1536x1536). Já o algoritmo *k-means* consome:

- 4,2 segundos para a imagem sem cortes do Marrocos;
- 4,9 segundos para a imagem sem cortes de Mercúrio;

- 4,2 segundos para a imagem sem cortes do Chipre;
- 5 segundos para a imagem sem cortes da França.

Na tabela 7.1, temos os tempos totais de geração da imagem comprimida. Para o efeito de comparação, essas imagens foram todas transformadas em BMP e comprimidas usando o algoritmo referido nesse trabalho (vvq) e a biblioteca *OpenCV* para geração de *JPG*, *PNG*, *TIFF* e *JPEG2000*.

imagem	vvq	jpg	png	tiff	jpeg2000
Marrocos	7000	256	200	222	1390
Mercúrio	7700	175	231	247	1337
Chipre	7000	270	220	241	1492
França	7800	170	180	230	1500

Tabela 7.1 – Tempo em milissegundos para a compressão

Na tabela 7.2, temos os tamanhos de arquivo gerados para cada conversão da tabela 7.1. O tamanho do *bitmap* foi adicionado para referência.

imagem	bitmap	vvq	jpg	png	tiff	jpeg2000
Marrocos	7.077.942	287.368	1.336.455	4.136.577	5.042.310	2.981.042
Mercúrio	7.077.942	287.368	542.584	3.621.962	4.486.448	2.635.707
Chipre	7.077.942	287.368	1.462.840	4.842.842	5.765.574	3.339.803
França	7.077.942	287.368	1.372.392	5.013.117	6.097.264	3.591.265

Tabela 7.2 – Tamanhos em *bytes* dos arquivos comprimidos

O cálculo do tamanho em bytes da imagem do quantizador é feito da seguinte maneira: (tamanho da matriz) + (tamanho do *codebook*) + (tamanho da palheta de cores). Assim, traduzindo em números temos:

$\frac{(384^2 \times 12)}{8} + (4096 \times 16) + (216 \times 3) = 287368$

É importante notar que cada célula da matriz se refere a uma entrada no *codebook*, portanto ela precisa de 12 bits para armazenar essa informação.

Para efetuar os testes de tempo de conversão o algoritmo 7.1 foi compilado em C++, usando a função *imwrite* da biblioteca *OpenCV*.

```
int main( int argc, char** argv ) {
    imread( argv[1], 1 ); //leitura do bitmap
    imwrite("output.jp2", src); //escrita no formato desejado
    return 0;
}
```

Algorithm 7.1 – Exemplo de conversão para *JPEG2000* usando *OpenCV*

Podemos ver que se comparado aos demais algoritmos o tempo de execução é em torno de 5 vezes maior do que o mais lento (*JPEG2000*). Porém vale lembrar, que, por definição, a descompressão dessa quantização vetorial é um processo simples. Nela os *codebooks* são simplesmente indexados para montar a imagem final, sem necessidade de comparações ou processamento posterior[1]. Isso é a maior vantagem da quantização vetorial nas imagens do CADRG, em comparação a por exemplo *JPG*, que tem complexidade similar na compressão e descompressão[2]. Essa vantagem se acentua em sistemas embarcados, onde os recursos de hardware são limitados em comparação com o computador *desktop*.

7.3 Tempo de descompressão

Na Tabela 7.3 podemos ver os tempos de descompressão para cada um dos formatos. Para efeito de comparação, todos os formatos de imagens foram abertos usando OpenCV e salvas em formato bitmap em todos os casos.

imagem	bitmap	vvq	jpg	png	tiff	jpeg2000
Marrocos	32	66	91	96	104	1214
Mercúrio	33	65	84	115	124	1102
Chipre	36	59	104	106	117	1287
França	32	60	95	96	130	1378

Tabela 7.3 – Tempos em milissegundos para descompressão

Podemos observar que os algoritmos para abrir bitmap e vvq variam pouco para a imagem de entrada em comparação aos demais algoritmos. Isso se deve ao fato de agirem da mesma forma independente da imagem de entrada, pois a mesma vai ter sempre o mesmo tamanho e também porque não há algoritmo de descompressão propriamente dito, somente indexação de valores. Vemos que o terceiro mais rápido após o vvq é o jpg, que é em média 50% mais lento.

[1] Veja o capítulo 2.2.
[2] Conforme [BK97].

Analisando os dados desta maneira é possível concluir que de fato o melhor custo benefício em todos os casos é o jpg, pois apresenta:

- tempo de compressão baixo (30 vezes mais rápido);
- descompressão próximo (50% mais lento);
- tamanho de imagem de 2 a 5 vezes maior.

Porém para casos extremos onde o tamanho da imagem seja crucial para qualquer tipo de imagem, o CADRG ainda tem um papel importante. Além disso podemos considerar que o algoritmo jpg use instruções multimídia avançadas presentes somente em processadores modernos e que se mostraria consideravelmente mais lento em sistemas embarcados.

7.4 Publicação

O projeto vai ser publicado como software de código aberto no SourceForge no seguinte endereço:

- https://sourceforge.net/projects/vvqimagecompressionforcadrginc/

8. CONCLUSÃO

Este trabalho serviu o seu propósito, que era o de introduzir os conceitos de compressão de imagens por quantização vetorial e como consequência elaborar um algoritmo que sirva para atender às especificações de imagem do CADRG, apesar de não gerar o arquivo CADRG final, como explicado no Capítulo 5.4.

O algoritmo funciona e produz resultados relevantes para as quatro imagens exploradas. Alguns artefatos indesejados e defeitos nas imagens ocorrem, mostrando que o algoritmo precisa ser refinado e aprimorado.

Os defeitos mais evidentes, como os pixels cinza dentro de áreas escuras por exemplo, podem ser consertados através de uma escolha de um bom pivô para a separação dos vetores. Atualmente o array de vetores é divididos exatamente ao meio para a avançar para o próximo passo. Isso é claramente uma má escolha. Supondo uma imagem de 100 vetores de pixel, com 70 vetores com pixels totalmente brancos e 30 vetores com pixels totalmente pretos. A primeira divisão vai gerar um array com 50 vetores brancos (que terão um vetor de treino perfeito) e outro array contendo os 20 vetores brancos restantes e mais 30 pretos, que terão um vetor de treino acinzentado. Este vetor acinzentado é uma má escolha. Seria preciso entender que no array ordenado existe um momento onde a diferença entre os pixels é muito grande, e este momento não é necessariamente o meio do array. Para isso é necessário calcular o pivô que divida os arrays na posição onde onde eles passam a se diferenciar drasticamente (do branco para o preto) e não exatamente no meio como é feito atualmente.

Porém isso acaba por desbalancear a árvore de chamadas recursivas do algoritmo. Neste caso, ao fim do algoritmo, já serão menos de 4096 entradas. Por um lado isso é bom, pois poderíamos tentar encontrar mais entradas que possam ser usadas. Mas para isso talvez seja interessante modificá-lo para rodar mais algumas iterações sobre todo o conjunto de vetores.

Outra ideia para melhorar o algoritmo seria comparar os vetores de treino finais. Ele atualmente agrupa os vetores segregando-os em sub-grupos que se dividem a cada passo do algoritmo e que não se encontram mais depois de divididos. Isso acaba eliminando a possibilidade de agrupar similaridades que venham a existir nestes sub-grupos. Especialmente levando em consideração o problema dos pixels pretos e brancos mencionado anteriormente.

8.1 Trabalhos Futuros

Existem diversas possíveis aplicações para a compressão de imagens com quantização vetorial. Ela oferece a vantagem de ser muito fácil de ser descomprimida (pois basta indexar a lista de *codebook* com a matriz de índices e usar a lista de cores) e portanto é amigável para com sistemas embarcados diversos, sistemas legados e certificados, etc. Dentre eles os presentes em ambientes de industria e da área da saúde. Além disso tem a possibilidade de ser usada para softwares que exijam extrema performance de descompressão, como por exemplo jogos que usam arquivos de textura.

Para estas finalidades, este trabalho pode ser expandido para tratar imagens de qualquer tamanho. Mas existe um fator complicador muito importante ao tratar desse tipo de imagem: se a mesma não puder ser redimensionada para um tamanho que seja múltiplo do número de vetores nos eixos X e Y, haverão o que se chama de pixels transparentes na quantização vetorial. Pixels transparentes são aqueles que ficam na borda da imagem, e fazem parte de vetores que não tem o número total de pixels por conta do tamanho da imagem. Esses pixels são complicados de tratar, pois não podem ser configurados como pretos nem como brancos, pois afetariam na média dos demais.

REFERÊNCIAS BIBLIOGRÁFICAS

[BK97] Bhaskaran, V.; Konstantinides, K. “Image and Video Compression Standards: Algorithms and Architectures”. Springer US, 1997.

[Dat14] Data-Compression.com. “Data-compression.com lbg animation”. Source: http://www.data-compression.com/vqanim.shtml, November 2014.

[Equ89] Equitz, W. “A new vector quantization clustering algorithm”, *IEEE Transactions on Acoustics Speech and Signal Processing*, vol. 37–10, October 1989, pp. 1568–1575.

[GG92] Gersho, A.; Gray, R. “Vector Quantization and Signal Compression”. Springer US, 1992.

[LCCBG80] Linde, Y.; Codex Corp., Mansfield, M.; Buzo, A.; Gray, R. “An algorithm for vector quantizer design”, *IEEE Transactions on Communications*, –28, January 1980, pp. 84 – 95.

[McK12] McKenna, E. “Product focus: Electronic flight bags”, *www.aviationtoday.com*, July 2012.

[MM94] Markuson, N.; MA., M. C. B. “Analysis of Compression Techniques for Common Mapping Standard (CMS) Raster Data”. Defense Technical Information Center, 1994.

[MN88] Mitchell, D. P.; Netravali, A. N. “Reconstruction filters in computer graphics”. In: SIGGRAPH, 1988.

[NIM90] NIMA. “Arc digitized raster graphic (adrg)”. Source: http://www.digitalpreservation.gov/formats/fdd/fdd000282.shtml, February 1990.

[NIM94a] NIMA. “Compressed arc digitized raster graphic (cadrg)”. Source: http://www.digitalpreservation.gov/formats/fdd/fdd000282.shtml, October 1994.

[NIM94b] NIMA. “Raster product format”. Source: http://www.digitalpreservation.gov/formats/fdd/fdd000298.shtml, October 1994.

[SCD92] Southard, D.; Corporation, M.; Division, U. S. A. F. S. C. E. S. “Compression of Digitized Map Images”. 1992.

[Wei14] Weisstein, E. W. “Voronoi diagram”. Source: http://mathworld.wolfram.com/VoronoiDiagram.html, 2014.

[Wik14a] Wikipedia. “Lloyd’s algorithm”. Source: http://en.wikipedia.org/wiki/Lloyd’s_algorithm, September 2014.

[Wik14b] Wikipedia. “Voronoi diagram”. Source: http://en.wikipedia.org/wiki/Voronoi_diagram, September 2014.

APÊNDICE A – IMAGENS ANTES E DEPOIS

Este anexo mostra as imagens antes e depois lado a lado.

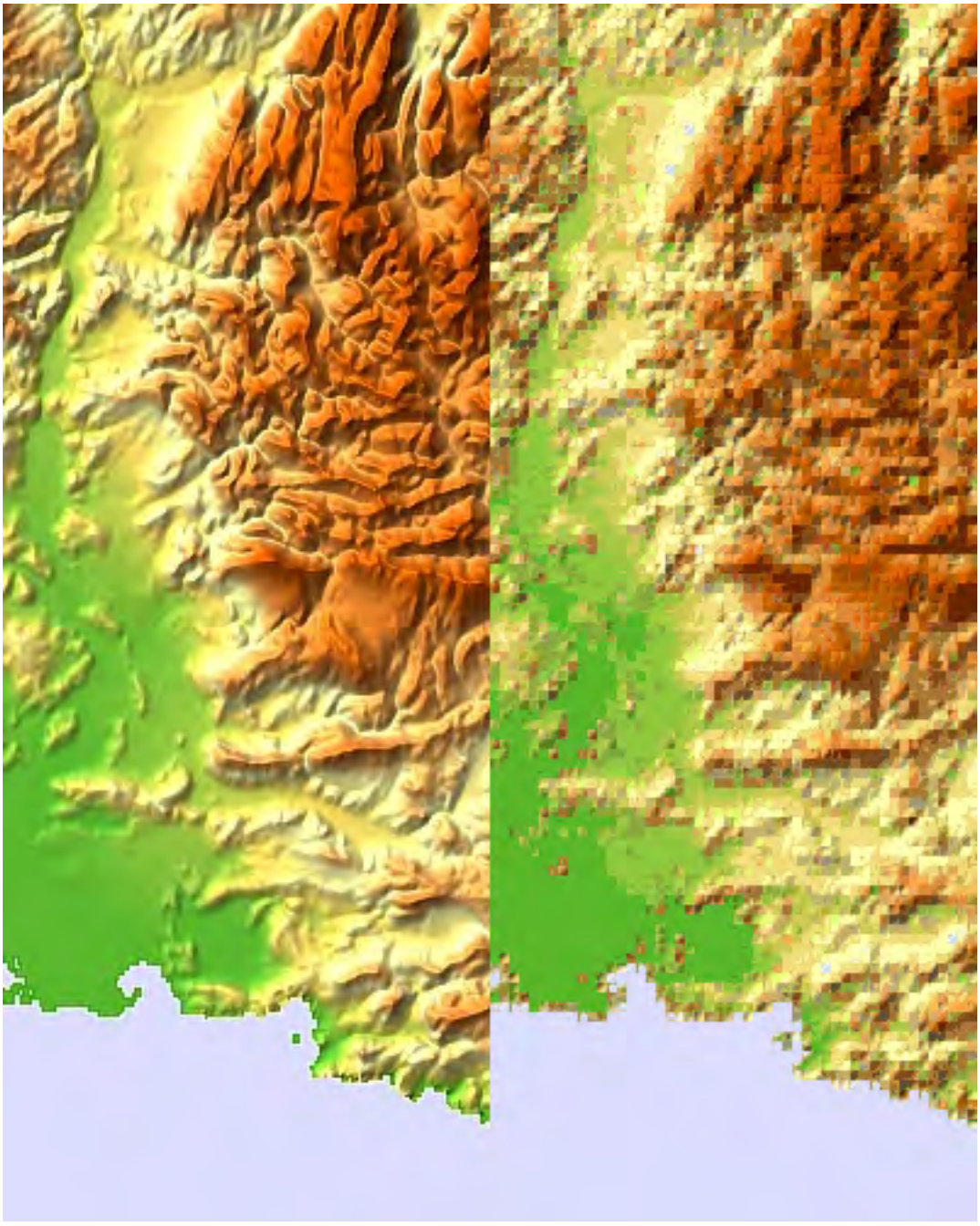


Figura A.1 – França antes e Depois

Figura A.2 – Chipre antes e depois

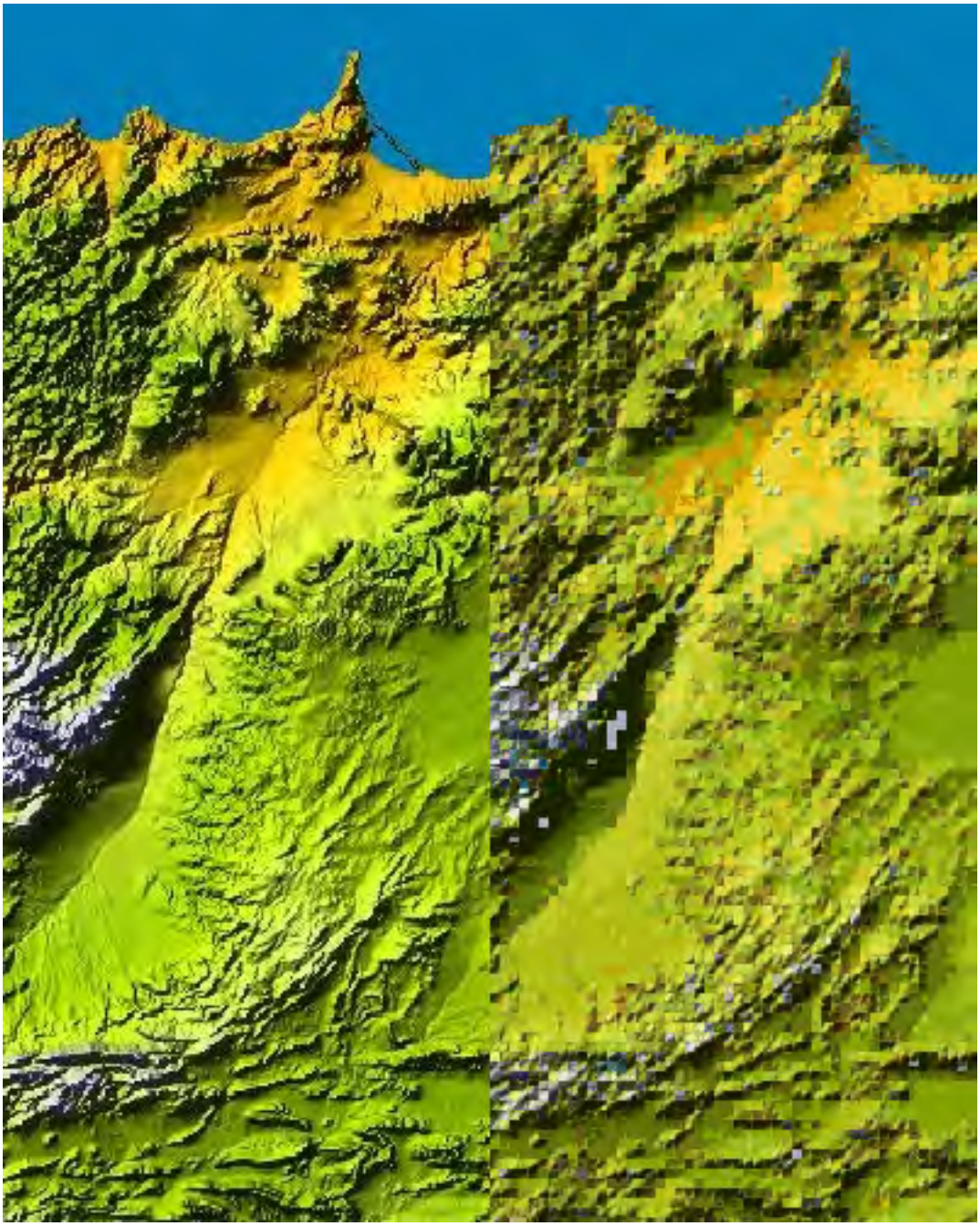


Figura A.3 – Marrocos antes e depois

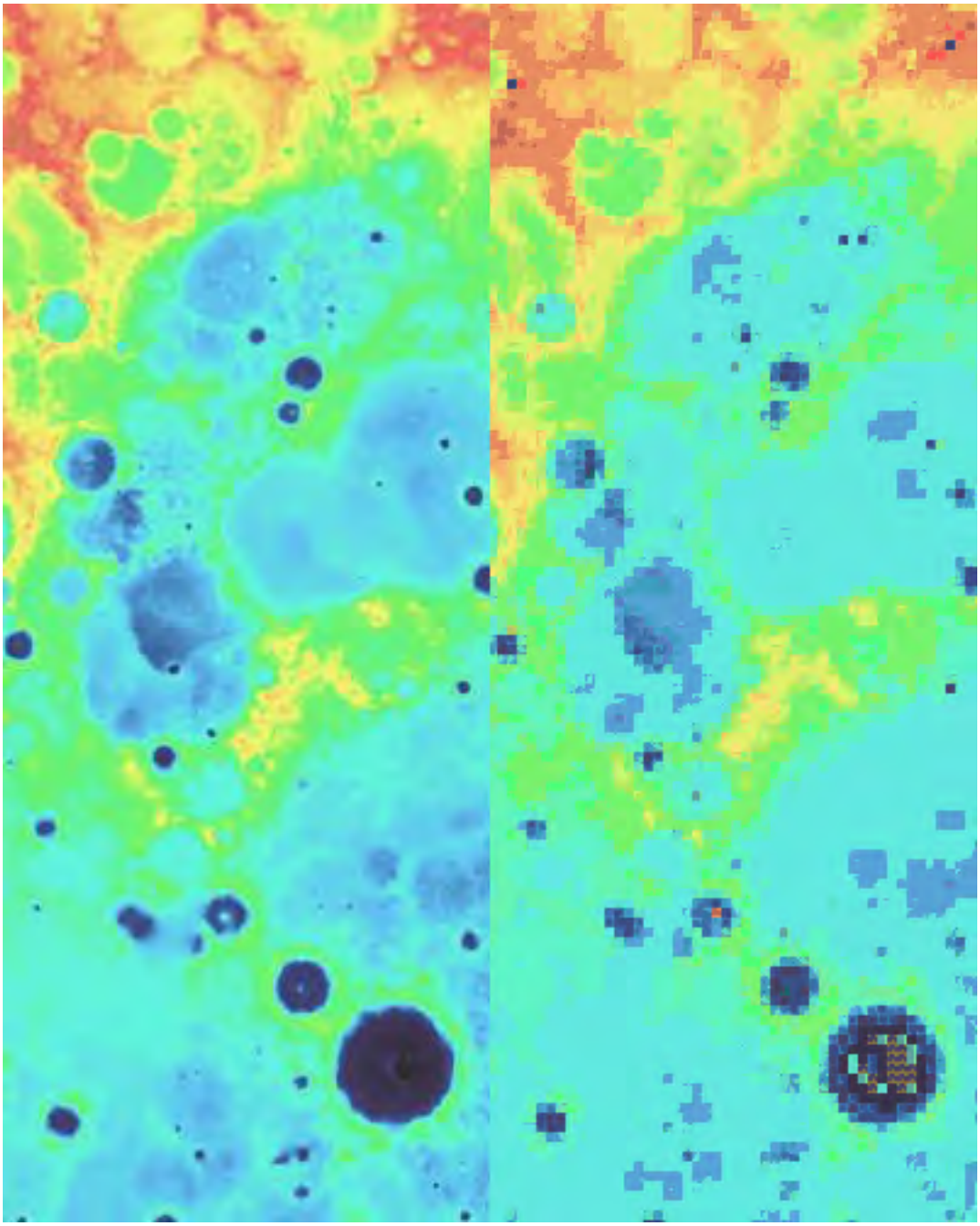


Figura A.4 – Planeta Mercúrio antes e depois

APÊNDICE B – FIGURAS SEM RECORTE ANTES E DEPOIS

Este anexo mostra as imagens antes da compressão, sem recorte.

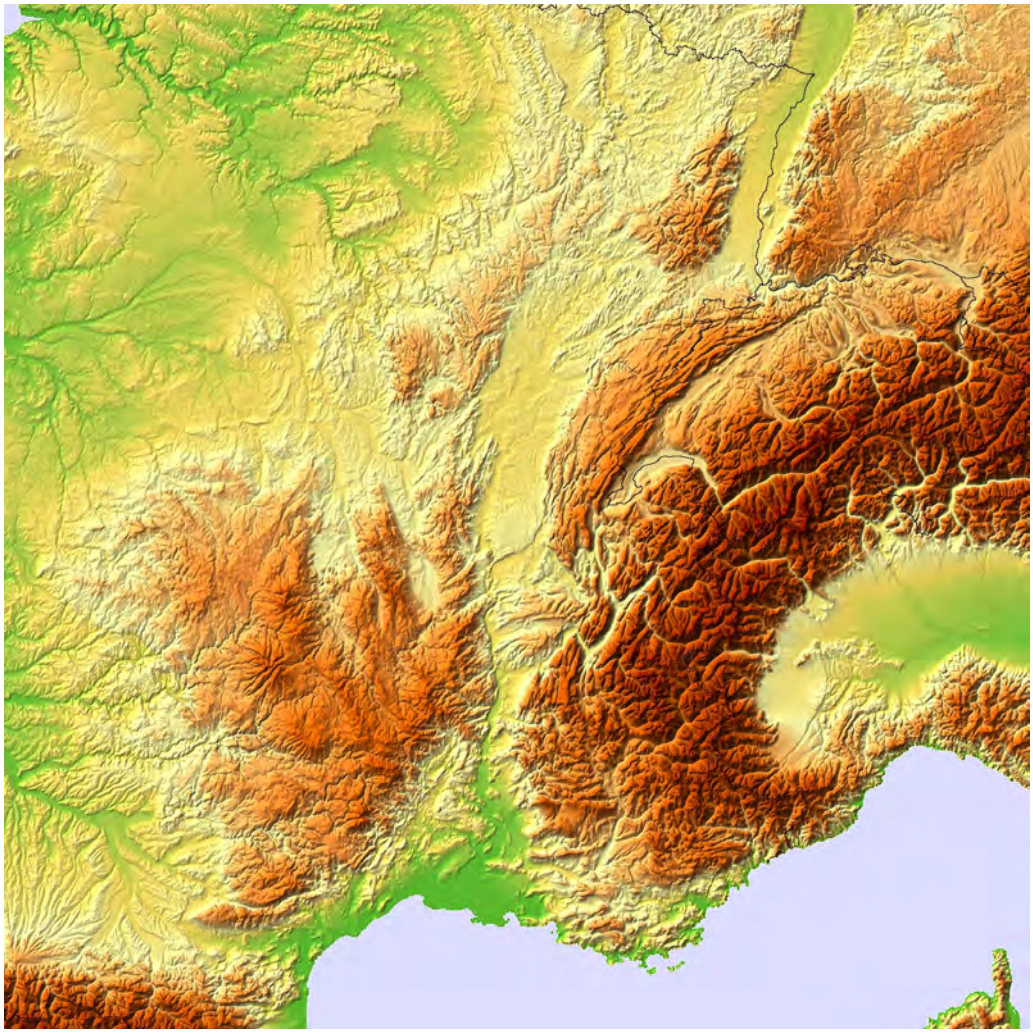


Figura B.1 – França sem recorte antes da compressão

Figura B.2 – França sem recorte depois da compressão

Figura B.3 – Chipre antes da compressão

Figura B.4 – Chipre depois da compressão

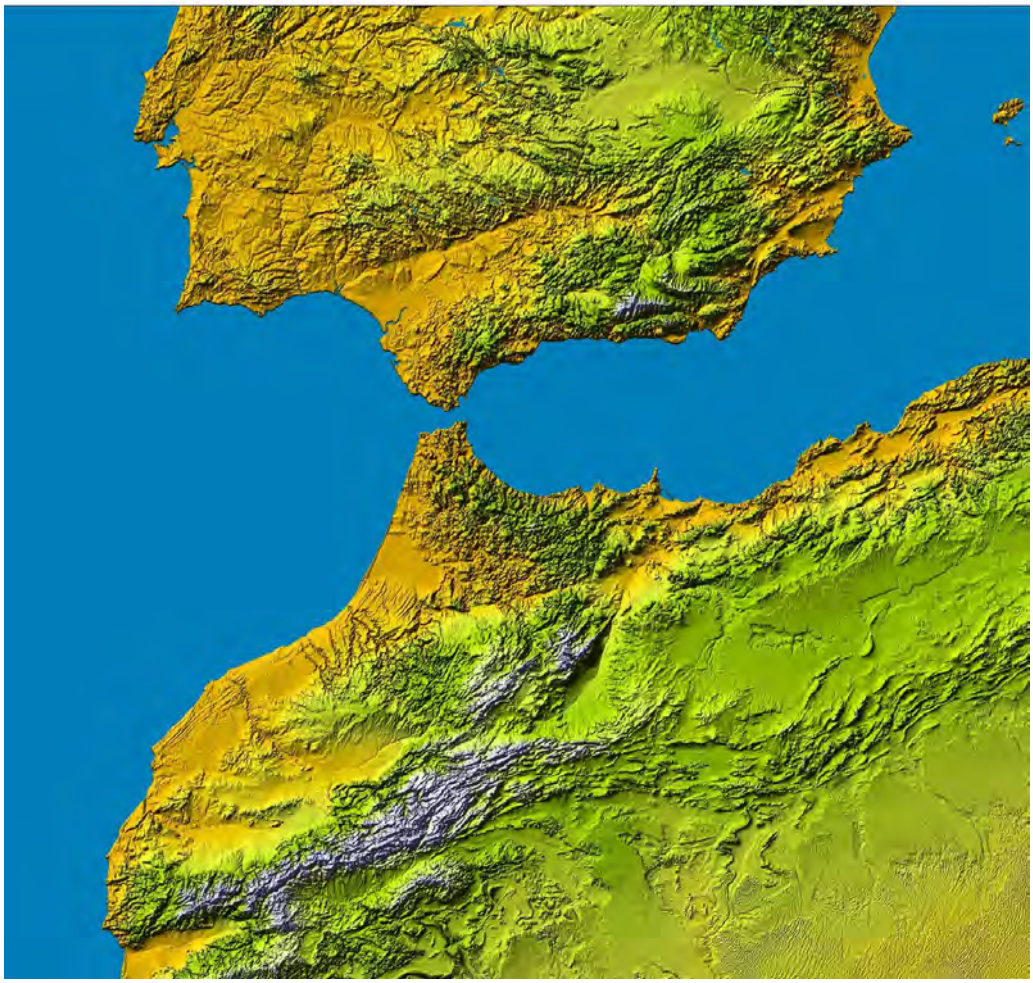


Figura B.5 – Marrocos antes da compressão

Figura B.6 – Marrocos depois da compressão

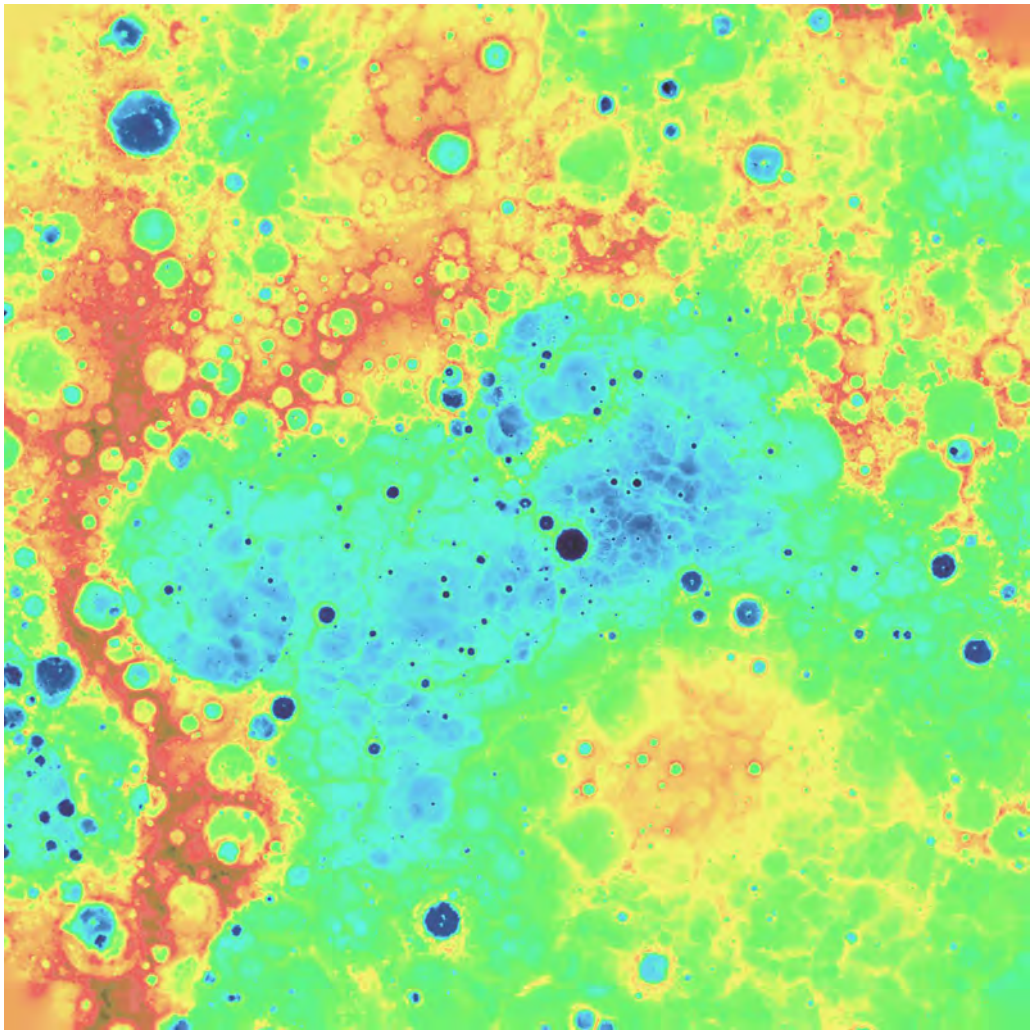


Figura B.7 – Mercúrio antes da compressão

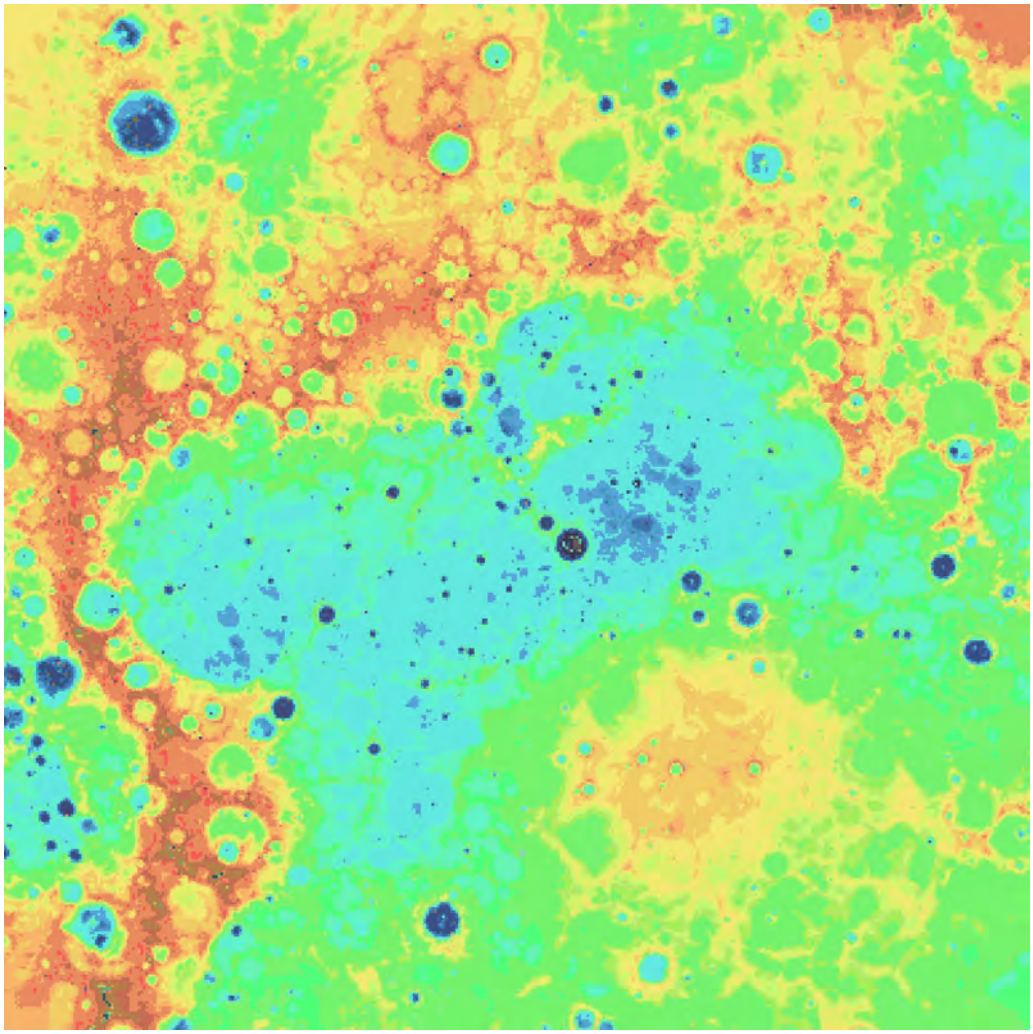


Figura B.8 – Mercúrio depois da compressão

APÊNDICE C – PROGRESSO DOS PASSOS DO ALGORITMO

Este anexo mostra imagens inteiras que ilustram o progresso dos 12 passos do algoritmo e mais 2 passos extras.

Figura C.1 – Progresso do Algoritmo com 1 passo

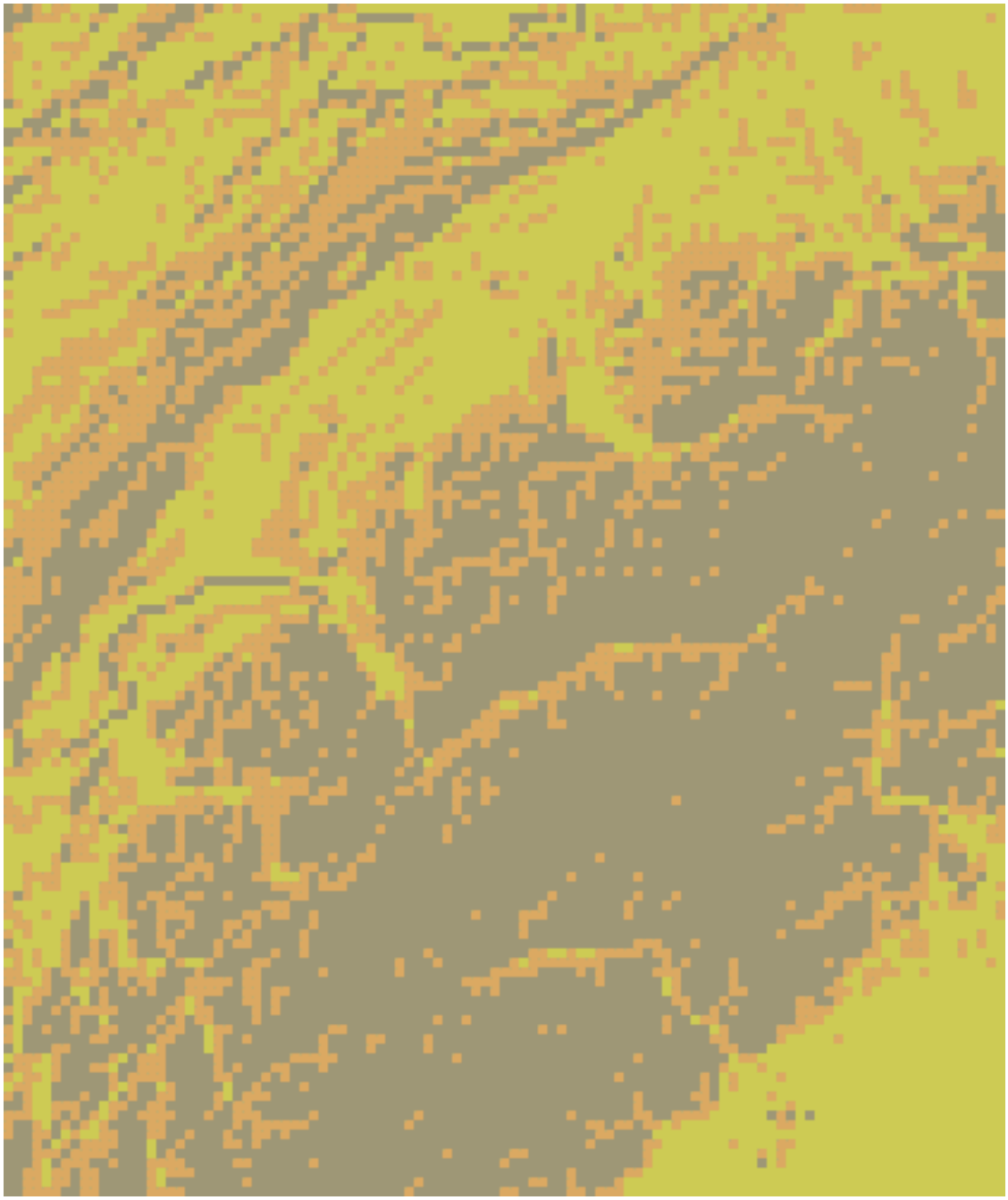


Figura C.2 – Progresso do Algoritmo com 2 passos

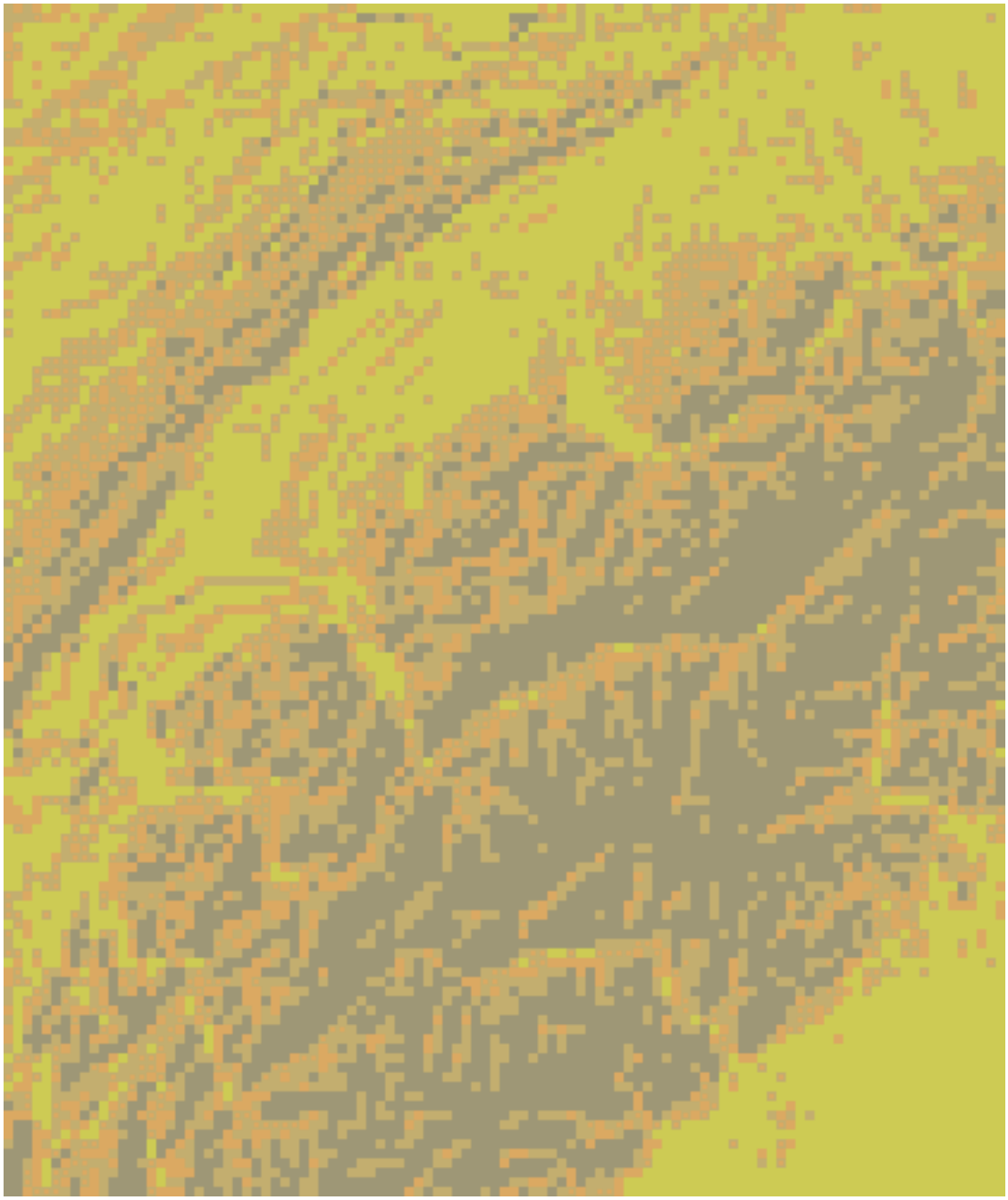


Figura C.3 – Progresso do Algoritmo com 3 passos

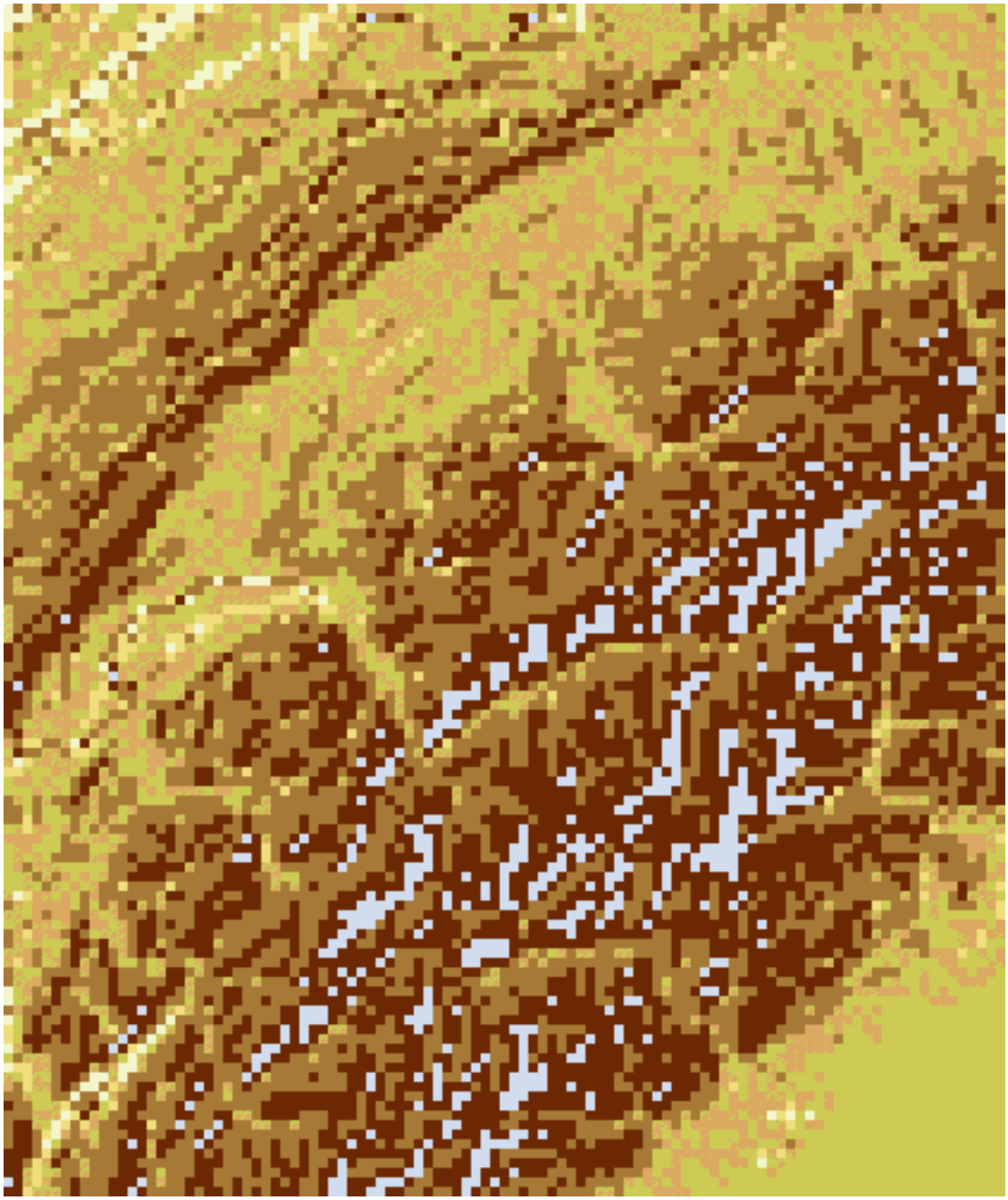


Figura C.4 – Progresso do Algoritmo com 4 passos

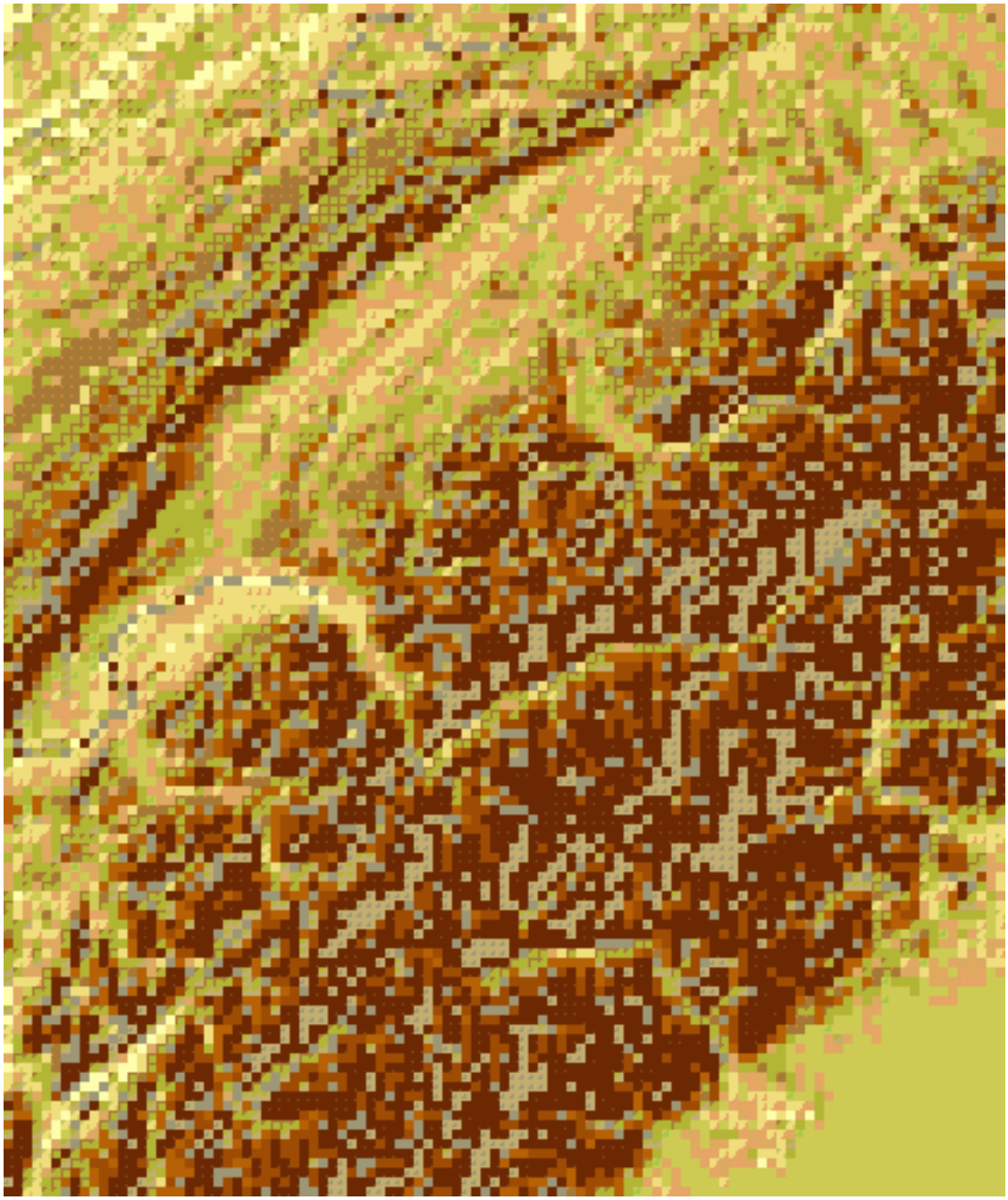


Figura C.5 – Progresso do Algoritmo com 5 passos

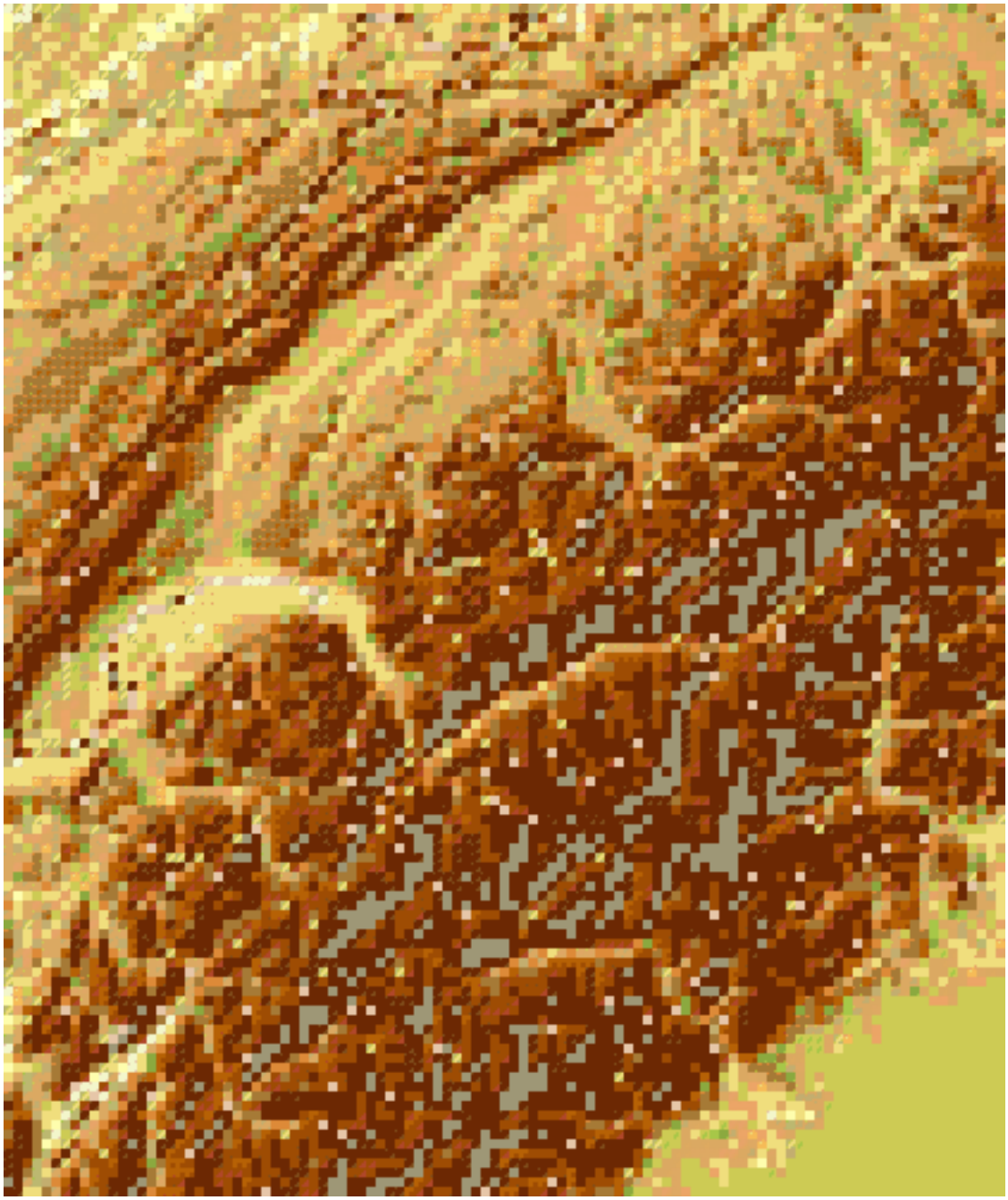


Figura C.6 – Progresso do Algoritmo com 6 passos

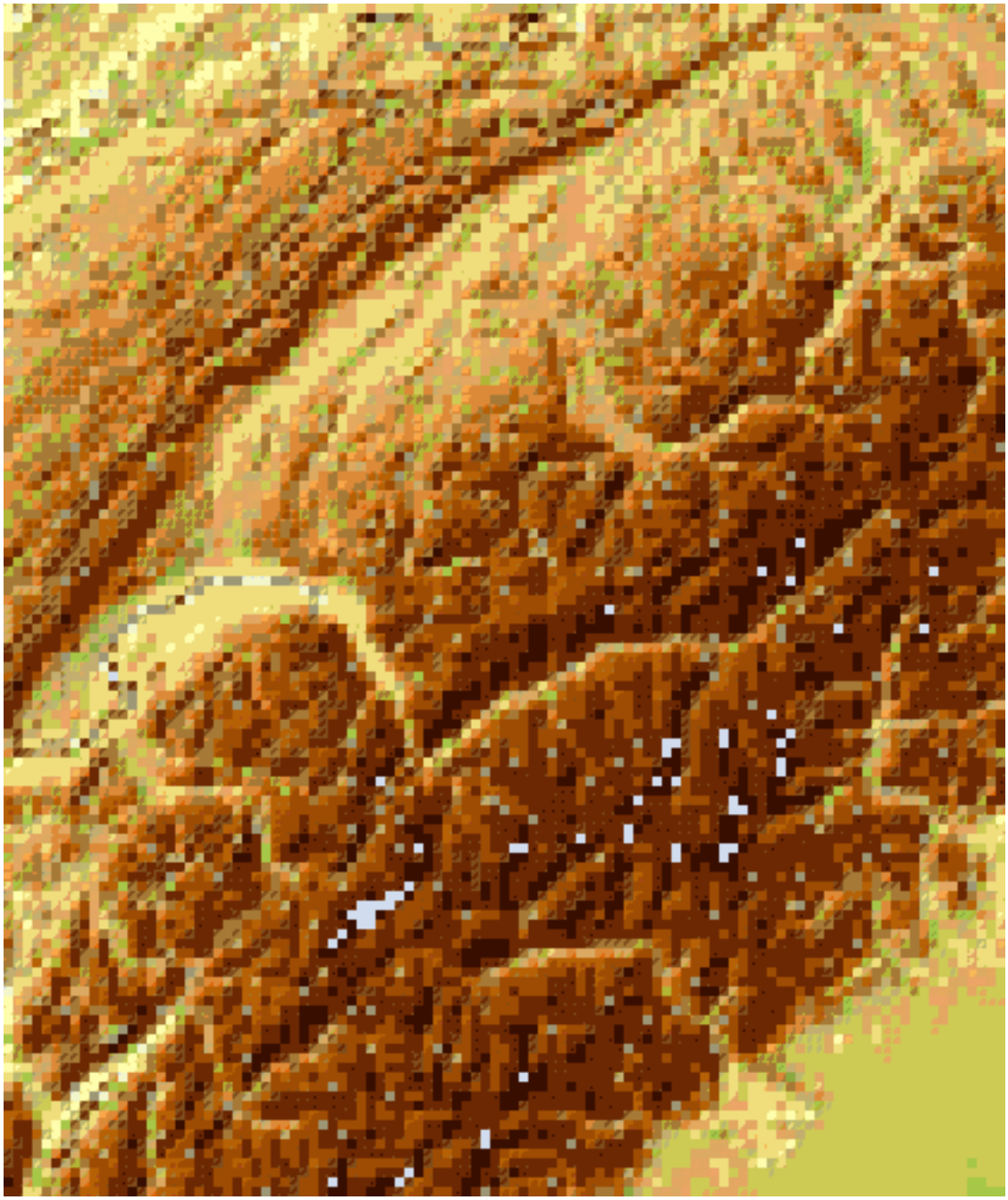


Figura C.7 – Progresso do Algoritmo com 7 passos

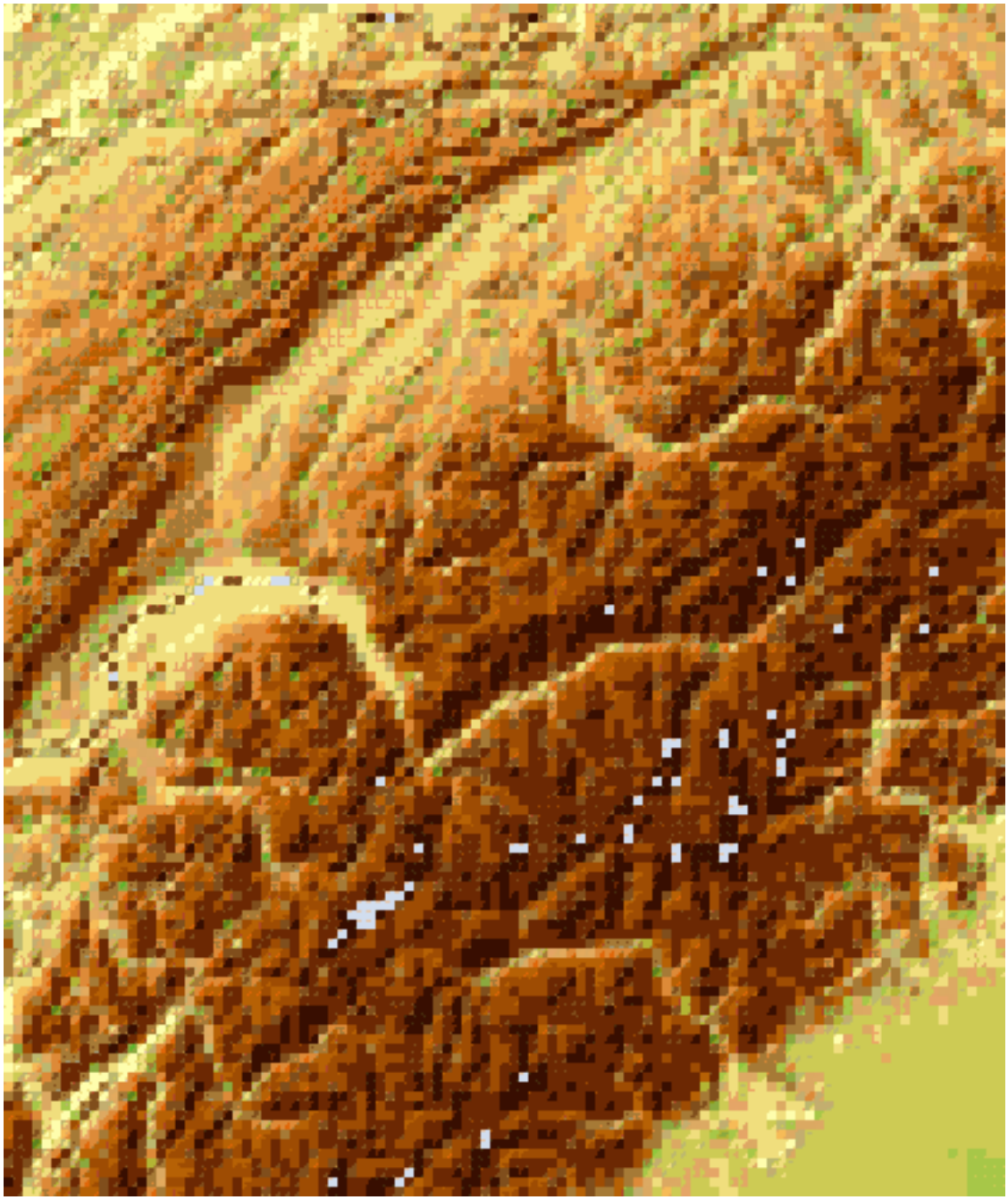


Figura C.8 – Progresso do Algoritmo com 8 passos

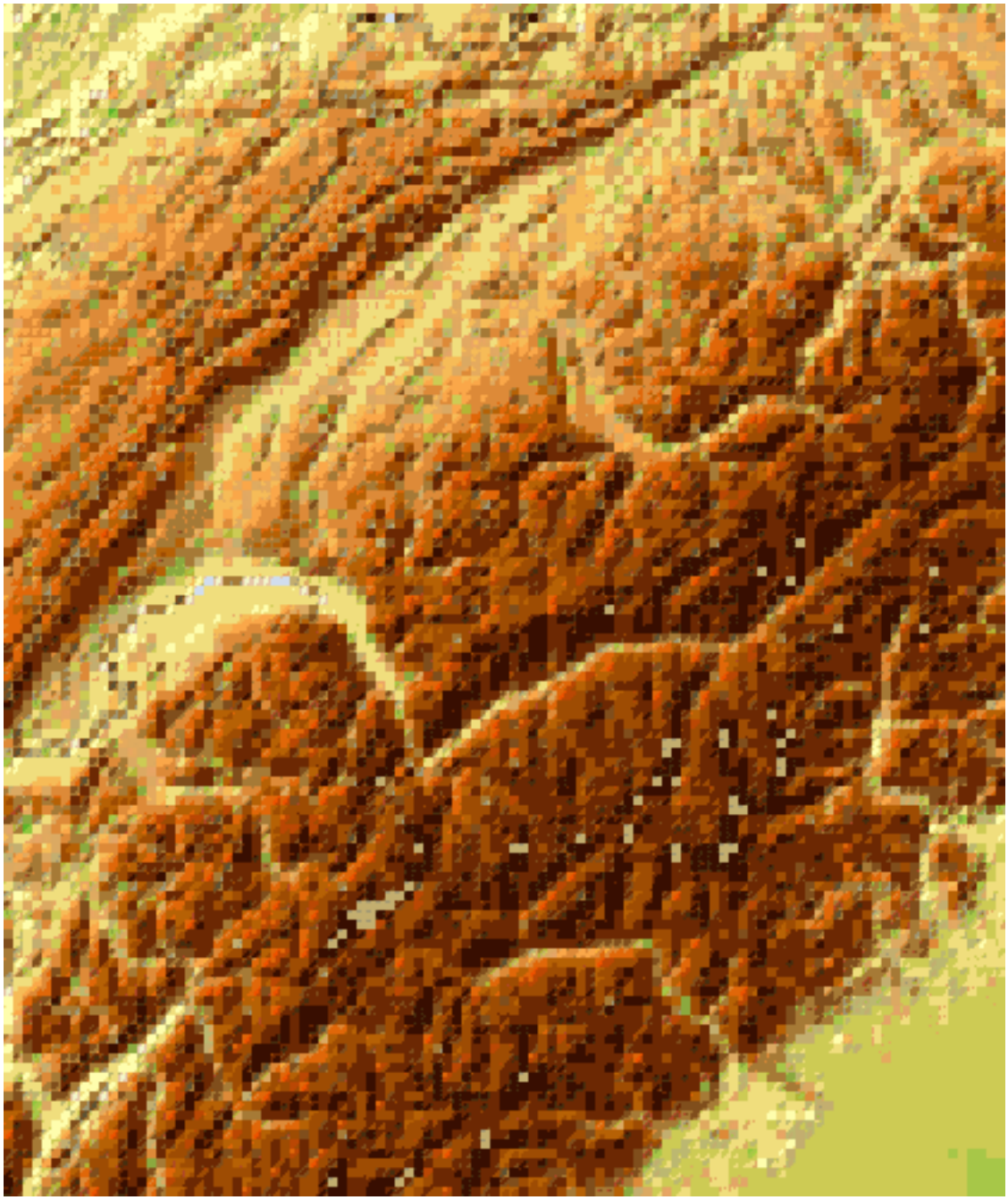


Figura C.9 – Progresso do Algoritmo com 9 passos

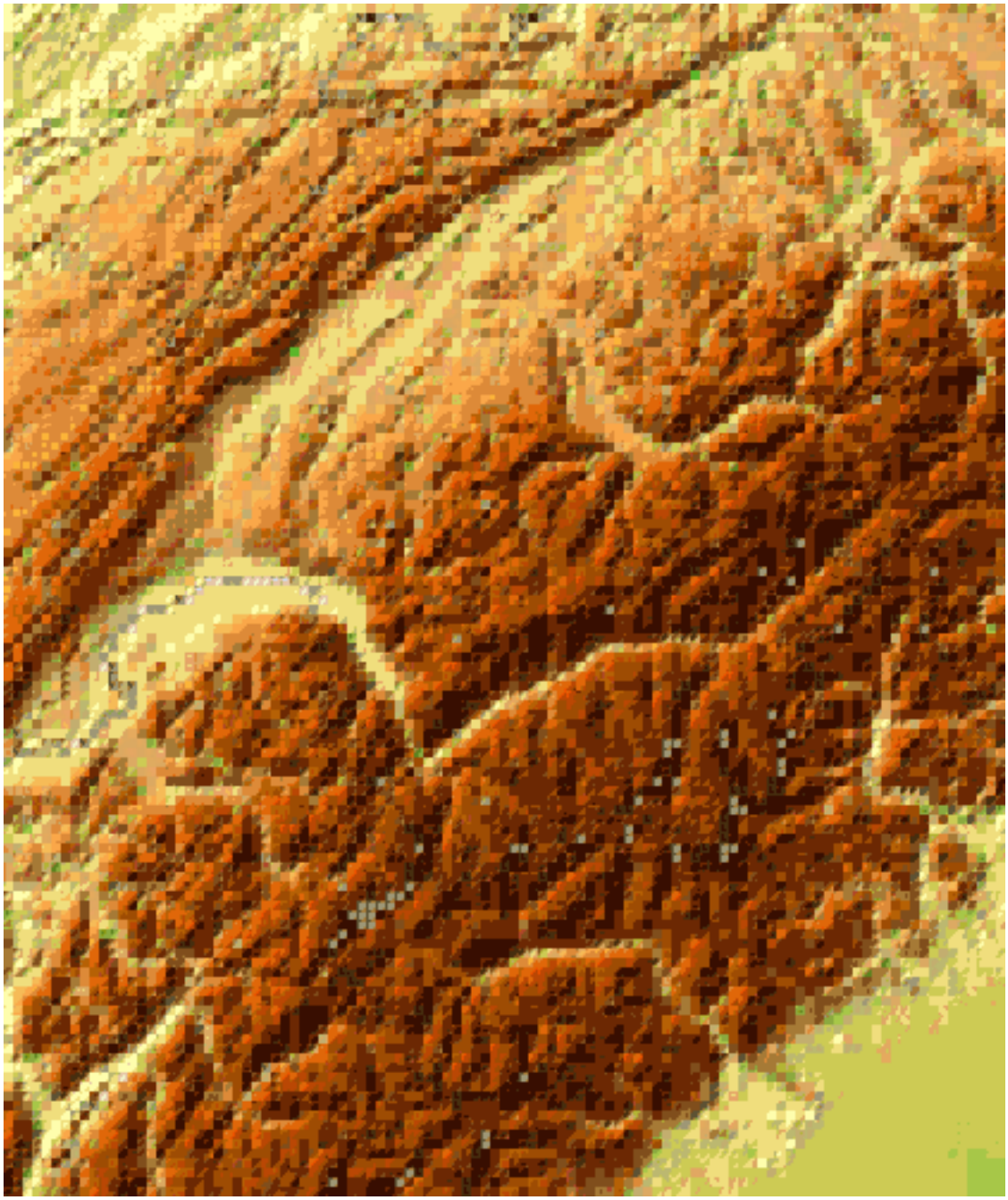


Figura C.10 – Progresso do Algoritmo com 10 passos

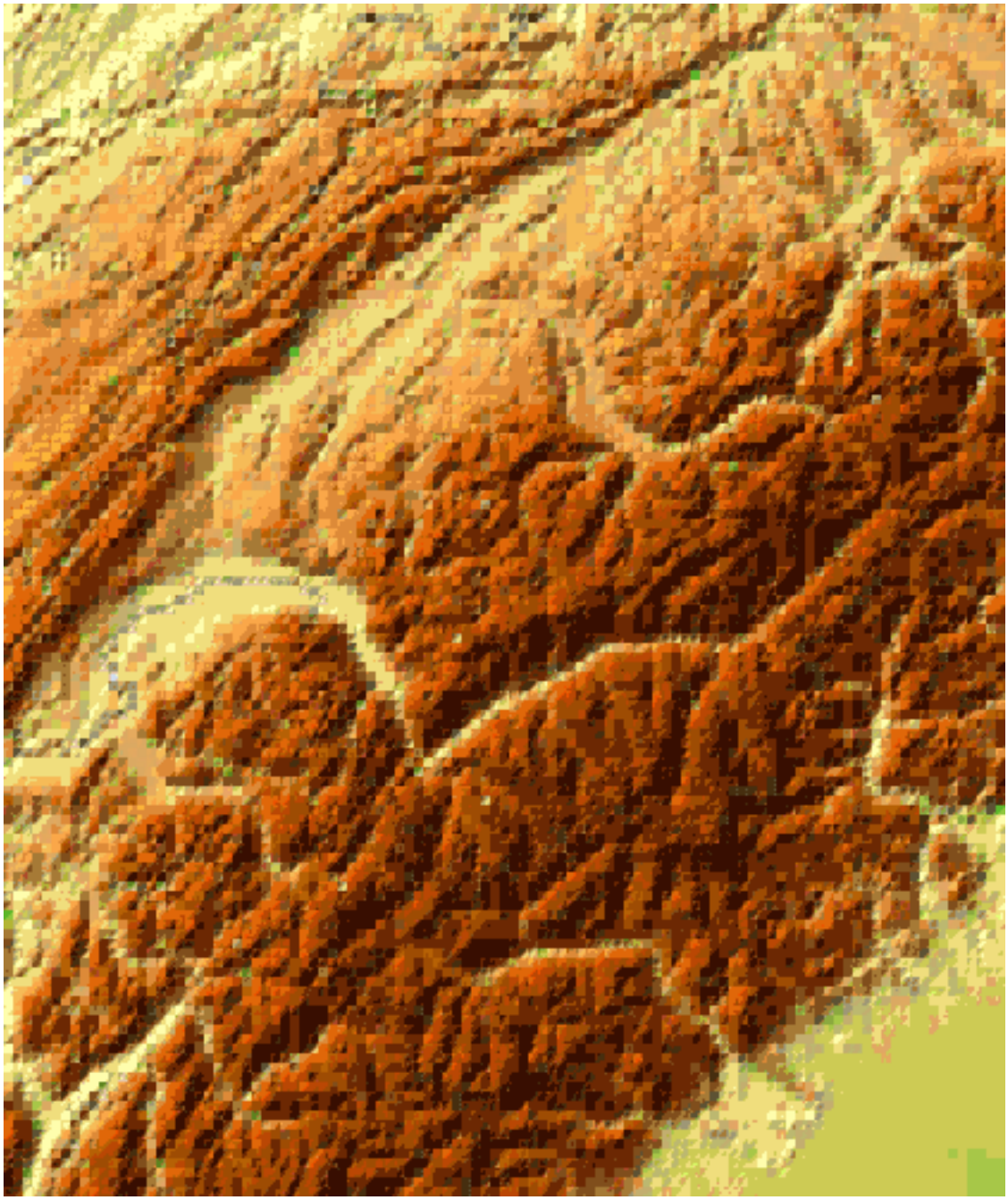


Figura C.11 – Progresso do Algoritmo com 11 passos

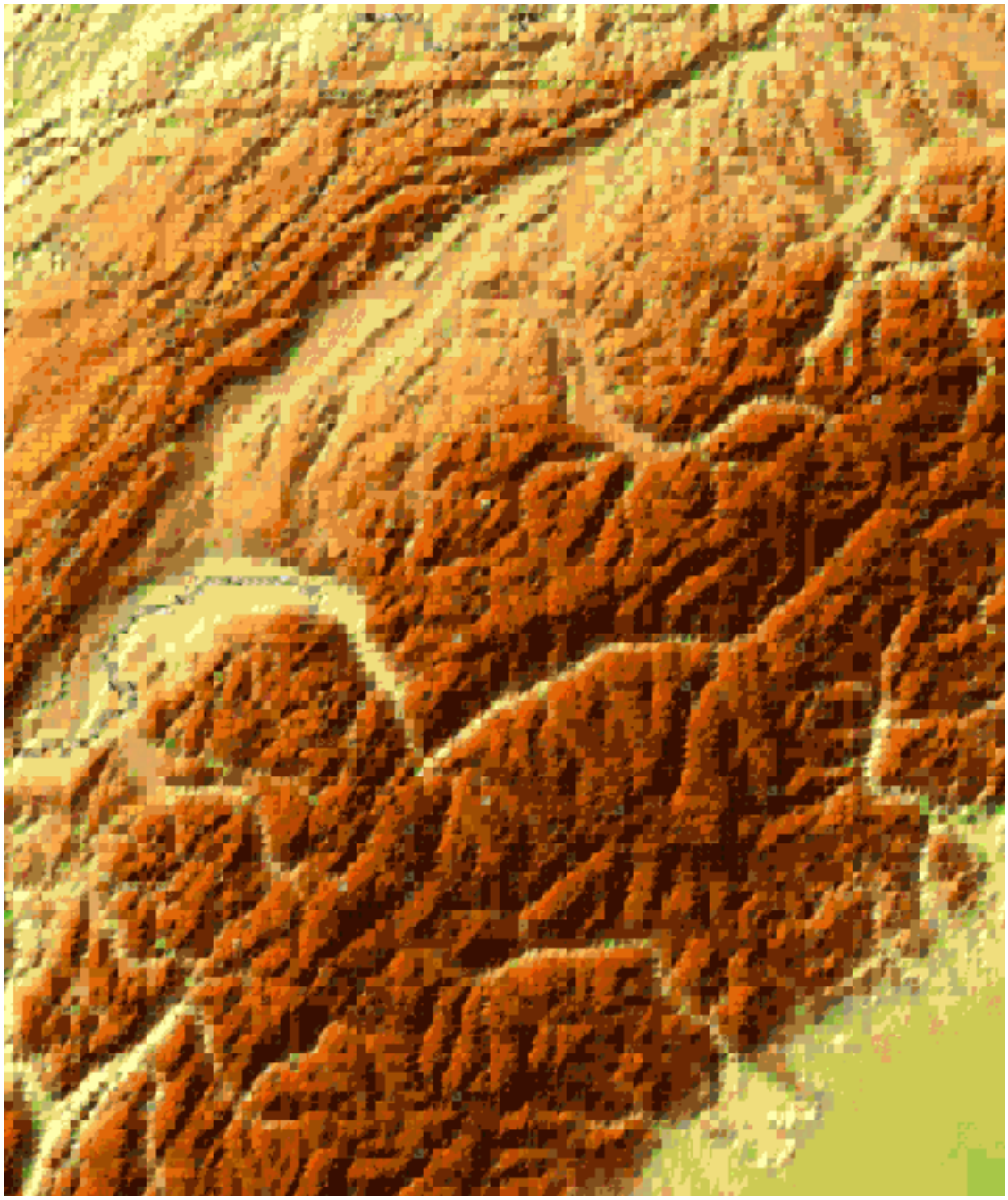


Figura C.12 – Progresso do Algoritmo com 12 passos

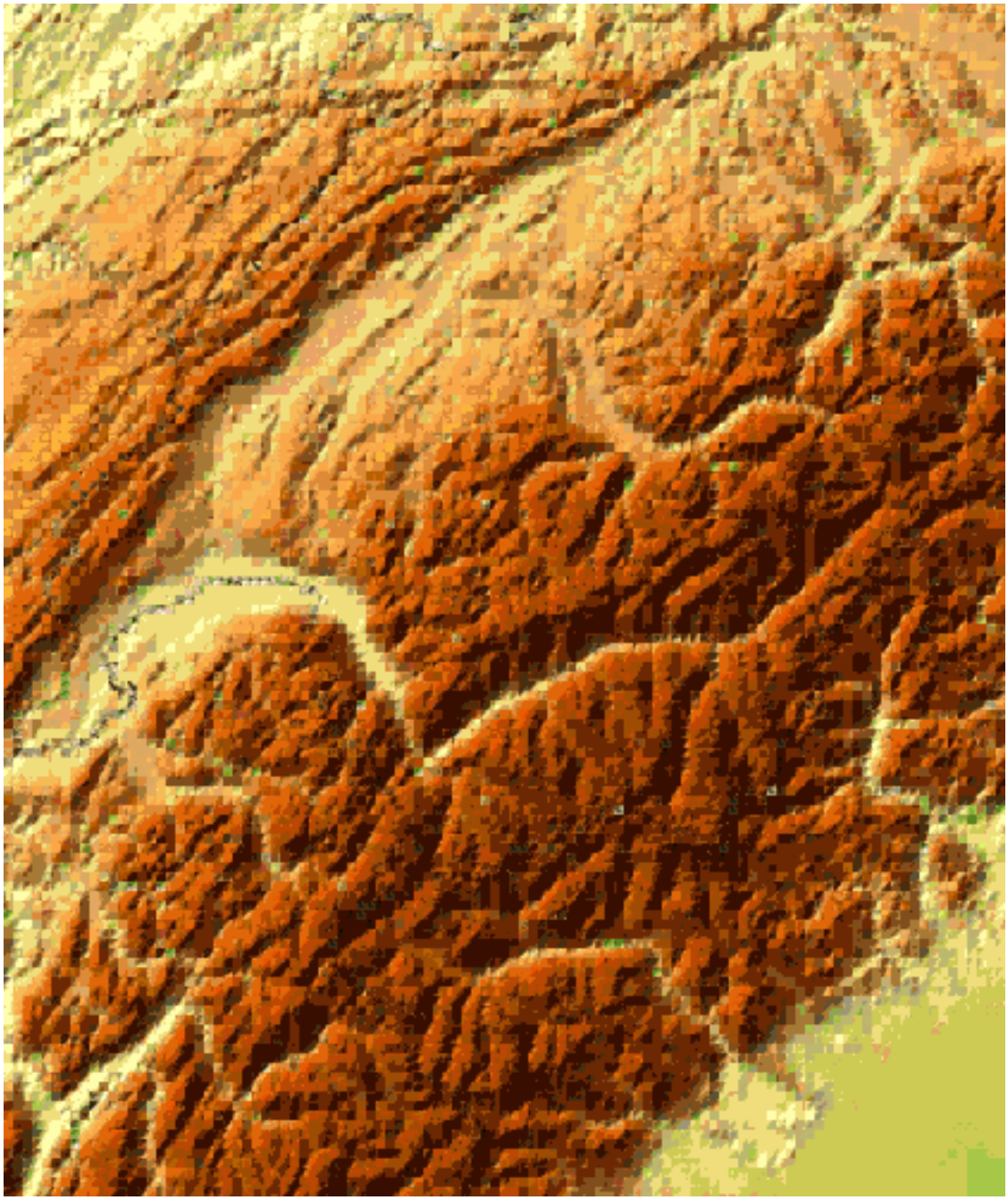


Figura C.13 – Progresso do Algoritmo com 13 passos

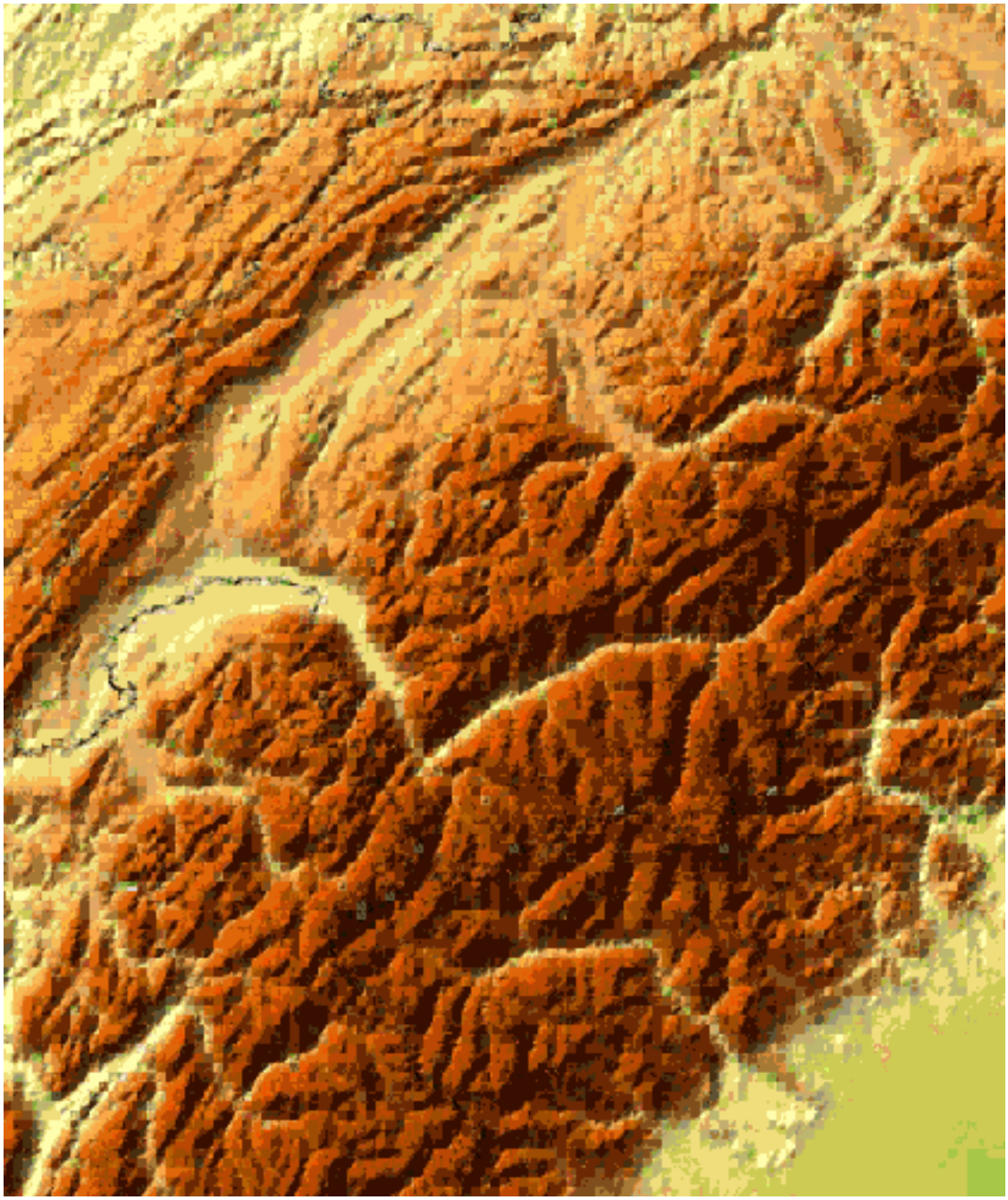


Figura C.14 – Progresso do Algoritmo com 14 passos