Universidade Federal do Rio Grande do Sul

Instituto de Matemática e Estatística

Programa de Pós-Graduação em Matemática

Transferência de Cores Entre Duas Imagens

Dissertação de Mestrado

Gustavo Schreiber Nunes

Porto Alegre, Fevereiro de 2022.

Dissertação submetida por Gustavo Schreiber Nunes como requisito parcial para a obtenção do grau de Mestre em Matemática pelo Programa de Pós-Graduação em Matemática do Instituto de Matemática e Estatística da Universidade Federal do Rio Grande do Sul.

Professor Orientador:

Prof. Dr. Diego Marcon Farias (PPGMat-UFRGS)

Professor Coorientador:

Prof. Dr. Ricardo Misturini (PPGMat-UFRGS)

Banca Examinadora:

Prof. Dr. Carlos Hoppen (PPGMap-UFRGS)

Prof. Dr. Lucas da Silva Oliveira (PPGMat-UFRGS)

Prof. Dr. Leandro Moraes Valle Cruz (Align Technology / Instituto de Sistemas e Robótica - Universidade de Coimbra)

Prof. Dr. Alexandre Tavares Baraviera (PPGMat-UFRGS)

Resumo

Nesta dissertação, apresentamos o problema de transferência de cores entre duas imagens, ou seja, o problema de representar a estrutura de uma imagem utilizando as cores de outra. Realizamos a construção teórica do problema e construímos um código de programação em linguagem Python que o resolve.

Para realizar a transferência de cores, resolvemos um problema de transporte ótimo discreto entre as distribuições de cores das duas imagens, utilizando um algoritmo de programação linear. Além disso, para melhorar o resultado visual, realizamos um relaxamento na condição usual de conservação de massa pontual da teoria de transporte ótimo e uma regularização dos mapas de transporte.

Este trabalho é baseado no artigo: Ferradans, Papadakis, Peyré, and Aujol, "Regularized discrete optimal transport", SIAM J Imaging Sciences, Vol. 7, No. 3, pp. 1853-1882.

Palavras-chave: Transferência de cores, transporte ótimo, programação linear, Python.

Abstract

In this thesis, we present the color transfer problem between two images, that is, the problem of representing the structure of an image by using the colors of another. We present the theoretical construction of the problem and we build a programming code in Python in order to solve it.

To perform the color transfer, we solve a discrete optimal transport problem between the color distributions of the two images, by using a linear programming algorithm. Moreover, to improve the final visual result, we perform a relaxation in the usual point mass conservation condition from optimal transport theory and a regularization of the transport map.

This work is based in the paper: Ferradans, Papadakis, Peyré, and Aujol, "Regularized discrete optimal transport", SIAM J Imaging Sciences, Vol. 7, No. 3, pp. 1853-1882.

Keywords: Color transfer, optimal transport, linear programming, Python.

Conteúdo

Agradecimentos

Até o momento, realizar o mestrado está sendo o projeto mais difícil da minha vida. É um desafio imenso e por várias vezes pensei em desistir. Entretanto, diversas pessoas foram responsáveis por me ajudar e por me manter firme nessa jornada.

Primeiramente agradeço à minha família, que sempre foi a minha base e me apoiou em todos os momentos. Por diversos momentos foram eles os responsáveis por me manter de pé e seguindo, mesmo sem acreditar que iria conseguir concluir o curso.

Também agradeço aos meus amigos, por diversos motivos diferentes: aqueles que me inspiraram a iniciar o mestrado, me incentivando a estudar para a prova de ingresso e inclusive estudando junto comigo. Aqueles que me mantiveram firme durante o curso, me motivando em diversos momentos. Também aqueles que viveram momentos felizes comigo durante este tempo, alegrando meus dias e os deixando mais leves.

Por fim, agradeço a todo o corpo docente do Instituto de Matemática e Estatística da UFRGS, em especial a meu orientador Diego e meu coorientador Ricardo, por me proporcionarem um ensino de qualidade e por me capacitarem para realizar este projeto.

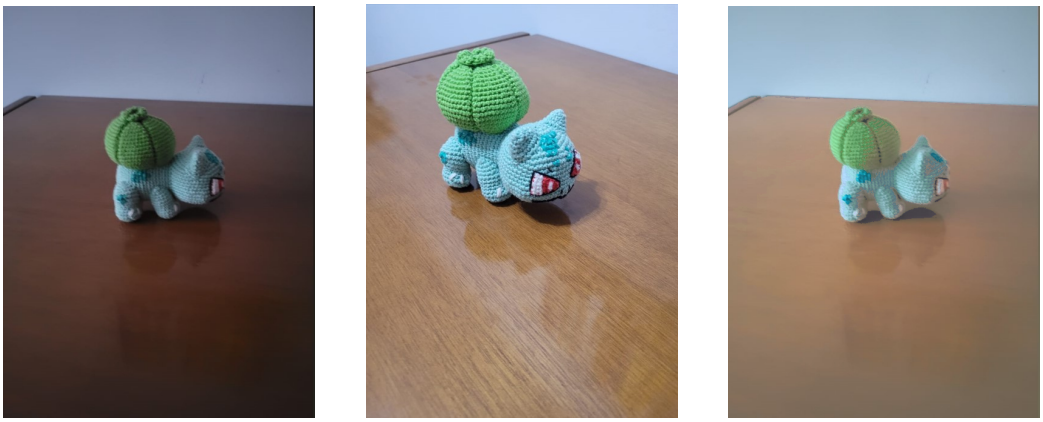
Introdução

Neste trabalho, estudamos o seguinte problema: dadas duas imagens, X^0 e Y^0, queremos obter uma terceira imagem $\tilde{X}^0$ que tenha a estrutura de X^0, mas utilizando as mesmas cores (ou cores muito semelhantes às) de Y^0. Dizemos que tal situação é um problema de transferência de cores entre imagens.

Diversas aplicações são possíveis para este problema: podemos por exemplo escolher Y^0 de tal forma que X^0 tenha tons mais claros ou mais escuros, aumente seu contraste ou fique com tons de cinza. Podemos também alterar a cor predominante de uma imagem de forma a mudar a sensação percebida ao observá-la [9]. Outra aplicação possível é a de conseguir comparar duas fotos do mesmo objeto, mas que foram tiradas em diferentes momentos ou de diferentes ângulos, alterando por exemplo a luminosidade [17] ou algum outro elemento de uma foto para outra (podemos ver um exemplo dessa aplicação na Figura 1).

Para atacar este problema, realizamos a construção de um algoritmo na linguagem de programação Python que resolve problemas de transferência de cores entre imagens utilizando como base o artigo "Regularized Discrete Optimal Transport" [4]. Este artigo é bem citado dentro da academia e possui diversos trabalhos feitos a partir dele, como por exemplo a aplicação do método em exames de ressonância magnética, estimativa de fluxo óptico, correspondência de sons [3] ou análise de problemas de tomografia sísmica [10].

O algoritmo funciona como uma função, onde inserimos as imagens X_0 e Y_0 (que chamamos de imagem base e imagem modelo, respectivamente) e temos como saída a imagem desejada $\tilde{X}_0$. Também são utilizados alguns parâmetros como entrada, sendo alguns deles de relaxamento e regularização do problema, a fim de melhorar o resultado visual, esses parâmetros serão explicados durante o texto. Na Página 80, vemos alguns resultados ao aplicarmos



(a) Imagem Base (b) Imagem Modelo (c) Resultado

Figura 1: Transferência de cores aplicada a uma situação de mudança de iluminação. Na imagem (a) vemos um objeto com baixa iluminação (imagem base), na imagem (b) vemos o mesmo objeto de outro ângulo e com melhor iluminação (imagem modelo), já a imagem (c) mostra o método aplicado às duas imagens para modificar a imagem base com as cores da imagem modelo.

o algoritmo construído.

No Capítulo 1, apresentamos algumas ferramentas preliminares importantes para o desenvolvimento da teoria apresentada durante o trabalho. No Capítulo 2, desenvolvemos a teoria referente ao problema de transferência de cores entre duas imagens. No Capítulo 3, temos a parte prática do texto, onde construímos o código em linguagem Python para resolver o problema. Por fim, no Capítulo 4, temos a apresentação dos principais resultados obtidos com a aplicação do método construído.

Esqueleto do Algoritmo

Nesta seção, montamos um esboço do algoritmo utilizado para a construção de $\tilde{X^0}$ a partir de X^0 e Y^0. Nos próximos capítulos, detalhamos cada uma das partes do algoritmo.

Em resumo, seguimos os seguintes passos no nosso algoritmo:

1. Escolhemos a imagem base X^0, que irá definir a estrutura da imagem final. Também escolhemos a imagem modelo Y^0, que irá fornecer as

cores da imagem final.

2. Escolhemos parâmetros $\lambda_X, \lambda_Y, k_X, K_X, k_Y, K_Y \in \mathbb{R}^+$, onde $k_X \leq 1 \leq K_X$ e $k_Y \leq 1 \leq K_Y$. Os parâmetros λ_X e λ_Y controlam o quão regular será o nosso resultado, enquanto k_X, K_X, k_Y e K_Y controlam o relaxamento na condição de conservação de massa que é usual em problemas de transporte ótimo. Estudamos cada um destes parâmetros no Capítulo 2.

3. Escolhemos um parâmetro $N \in \mathbb{N}$ e, utilizando o algoritmo k-means, realizamos uma clusterização das imagens X^0 e Y^0. Essa clusterização gera N cores a partir de X^0 e N cores a partir de Y^0. Em seguida, construímos os conjuntos de cores $X = \{x_1, \ldots, x_N\}$ e $Y = \{y_1, \ldots, y_N\}$, onde cada elemento $x_i \in X \subseteq \mathbb{R}^3$ e $y_j \in Y \subseteq \mathbb{R}^3$ é uma cor gerada a partir do algoritmo. Estudamos a clusterização de imagens na seção 2.1.

4. Encontramos, através de um algoritmo de programação linear e utilizando os parâmetros do Item 2, uma matriz $\pi \in \mathbb{R}^{N \times N}$ que é um transporte ótimo de X para Y. Estudamos problemas de programação linear na seção 1.4 e sobre transporte ótimo nas seções 1.3 e 2.2.

5. Após encontrar a matriz π, construímos uma função $T : X \to \mathbb{R}^3$ que associa a cor $x_i \in X$ à cor $T(x_i) \in \mathbb{R}^3$ dada por

$$T(x_i) = \frac{\sum_{j=1}^{N} \pi_{i,j} y_j}{\sum_{j=1}^{N} \pi_{i,j}}. \tag{1}$$

   A construção dessa função T é feita porque nem sempre a matriz π é uma bijeção entre X e Y. Intuitivamente, T calcula o baricentro ponderado das imagens de cada x_i por π.

6. Para finalizar a construção de $\tilde{X^0}$ (que chamamos de imagem objetivo), realizamos um pós-processamento através de uma interpolação, utilizando a função $T^0 : \mathbb{R}^3 \to \mathbb{R}^3$

$$T^0(x) = T(x_{i(x)}) + x - x_{i(x)}, \tag{2}$$

   onde $i(x) = \underset{1 \leq i \leq N}{\arg\min} \|x - x_i\|$.

A imagem $\tilde{X^0}$ é obtida então da seguinte forma: partindo da imagem X^0, um pixel com a cor x é recolorido com a cor $T^0(x)$. Estudamos sobre a interpolação na seção 2.5.

Durante o restante do texto, detalhamos cada um dos passos do algoritmo e também mostramos uma maneira de implementá-lo utilizando a linguagem Python.

Capítulo 1

Ferramentas Preliminares

Nesta seção, trazemos alguns resultados, conceitos e definições importantes que utilizamos durante o restante do texto. Na Seção 1.1, mostramos como interpretar uma imagem matematicamente, através das cores de cada um dos seus pixels. Na Seção 1.2, trabalhamos a ideia do histograma (para imagens em preto e branco) e da distribuição de cores (para imagens coloridas) de uma imagem. Na Seção 1.3, trazemos alguns conceitos básicos da teoria de transporte ótimo. Na Seção 1.4, mostramos no que consiste um problema de programação linear, iremos resolver um problema deste tipo para realizar a transferência de cores entre duas imagens. Por fim, nas Seções 1.5 e 1.6, exemplificamos o Método Simplex e o Método Simplex de duas fases, que são os principais métodos utilizados na resolução de problemas de programação linear.

1.1 Leitura e Representação de uma Imagem

Começamos abordando a ideia de como representar e trabalhar com uma imagem matematicamente. Dada uma imagem X^0, temos que ela pode ser divida em várias pequenas unidades, chamadas de pixels. A cada pixel é associada uma cor e existem diversas maneiras de representar a cor de um pixel. Neste texto, utilizamos o sistema de cores RGB, onde cada cor é formada pela integração de 3 escalas: vermelho(R), verde(G) e azul(B). A cada uma destas escalas é atribuído um valor de 0 a 255. Assim, cada pixel pode ter a sua cor representada como um elemento $p \in \{0, 1, 2 \ldots, 255\}^3$. Na Tabela 1.1, vemos a representação no sistema RGB de algumas cores do

nosso espectro visual.

Cor	Nome	Representação RGB
	Branco	$(255, 255, 255)$
	Preto	$(0, 0, 0)$
	Vermelho	$(255, 0, 0)$
	Verde	$(0, 255, 0)$
	Azul	$(0, 0, 255)$
	Amarelo	$(255, 255, 0)$
	Magenta	$(255, 0, 255)$
	Ciano	$(0, 255, 255)$
	Âmbar	$(255, 191, 0)$
	Bordô	$(128, 0, 0)$
	Cinza	$(128, 128, 128)$
	Laranja	$(255, 165, 0)$
	Rosa	$(255, 203, 219)$

Tabela 1.1: Representação no sistema RGB de algumas cores do nosso espectro visual.

Assim, se definirmos $L \in \mathbb{N}$ como sendo o número de pixels de largura da nossa imagem e $A \in \mathbb{N}$ como sendo o número de pixels de altura, temos que cada um dos $n = L \times A$ pixels está associado a uma cor, representada pelas suas coordenadas RGB. Sendo assim, podemos pensar em X^0 como um elemento de $\{0, 1, 2, \ldots, 255\}^{n \times 3}$.

Ao visualizar X^0 desta maneira, podemos realizar vários tipos de transformações na nossa imagem. Por exemplo, se quisermos alterar a cor específica de um pixel da imagem, basta alterarmos as suas componentes no sistema RGB para a cor desejada. Uma transformação comum em imagens é alterar todas as cores para tons de cinza. Temos que a cor de um pixel está em um tom de cinza quando as suas três componentes tem o mesmo valor (chamado de luminância), então uma possível maneira de alterar a cor de cada pixel é considerar uma média ponderada entre as suas três componentes, como na transformação $f : \{0, 1, 2, \ldots, 255\}^3 \to \mathbb{R}$ dada por

$$f(x, y, z) = \frac{30x + 59y + 11z}{100}. \tag{1.1}$$

Como o valor de (1.1) nem sempre será dado por um número inteiro, precisamos realizar o arredondamento do mesmo. Na Figura 1.1, vemos o resultado

(a) Imagem original. (b) Imagem em tons de cinza.

Figura 1.1: Resultado da aplicação da função (1.1) em uma imagem.

desta transformação em uma imagem, onde alteramos a cor de cada pixel para o tom de cinza respectivo da sua luminância, utilizando a função (1.1).

Apesar de também ser possível considerar a média aritmética simples entre os valores das três componentes de cada pixel, temos que a visão humana é mais sensível a tons de verde, ao mesmo tempo em que é menos sensível a tons de azul. Por isso que utilizamos a média ponderada e damos um peso maior ao valor da componente verde.

1.2 Histograma de uma Imagem

Uma característica importante de uma imagem é o seu histograma de cores, que indica a quantidade de pixels que possuem cada uma das cores do sistema RGB.

Começamos definindo o histograma de uma imagem que possui apenas tons de cinza. Seja X^0 tal imagem, com L pixels de largura e A pixels de altura. Seja $p = (l, a)$ um pixel de X^0, com $l \in \{0, \ldots, L-1\}$ e $a \in \{0, \ldots, A-1\}$, e seja $x_p = (r_p, g_p, b_p)$, com $r_p, g_p, b_p \in \{0, \ldots, 255\}$ a cor associada a p. Como x_p está em um tom de cinza, temos que $r_p = g_p = b_p$.

O histograma de X^0 é uma função que conta o número de vezes que cada tom de cinza aparece na imagem. Temos então que a cada tom de cinza $k \in \{0, 1, 2, \ldots, 255\}$ está associado um valor $r_k \in \mathbb{N}$ que indica o número de pixels representados por este respectivo tom. Assim, visualizamos

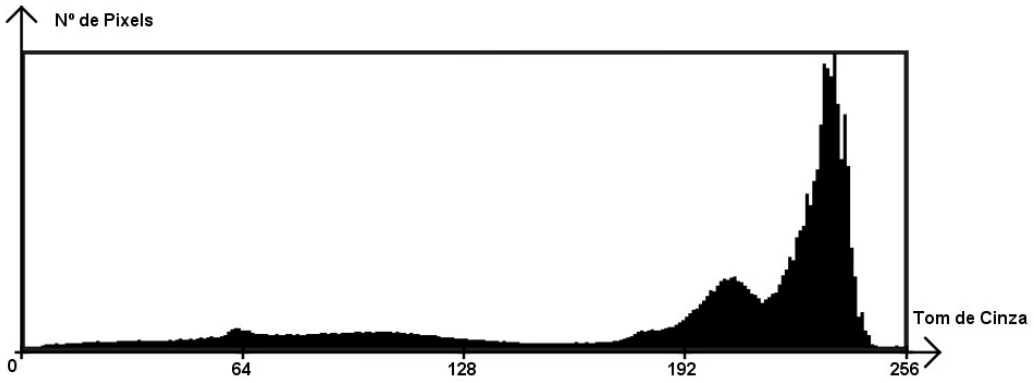


Figura 1.2: Histograma da Figura 1.1b.

o histograma como uma função $h : \{0, \ldots, 255\} \to \mathbb{N}$ dada por

$$h(k) = r_k. \tag{1.2}$$

Na Figura 1.2, vemos o histograma da Figura 1.1b. Através deste histograma, vemos que a imagem tem uma predominância maior de tons mais claros, o que se confirma ao observarmos a imagem.

Ao analisar o histograma de uma imagem em tons de cinza, conseguimos obter algumas informações importantes: se há uma maior concentração de pixels na parte esquerda do histograma, temos que a imagem analisada é mais escura, caso haja uma maior concentração de pixels na direita do histograma, nossa imagem será mais clara (como no nosso exemplo), se a maioria dos pixels estão localizados em uma faixa pequena de tons de cinza, temos que a imagem apresenta pouco contraste.

Podemos realizar certos tipos de transformações no histograma de uma imagem, através de uma função $T : \{0, 1, 2, \ldots, 255\} \to \{0, 1, 2, \ldots, 255\}$. Através de T, conseguimos construir um histograma h' tal que:

$$h'(k) = \sum_{i \in T^{-1}(k)} r_i. \tag{1.3}$$

Normalmente, exigimos que T seja uma função monótona não-decrescente, a fim de evitar alguns artefatos causados pela inversão de intensidades dos tons de cinza. Existem funções para deixar uma imagem mais clara, mais escura, com menos contraste, entre outras. Há ainda transformações que

se utilizam justamente da inversão das intensidades dos tons de cinza para serem realizadas, como por exemplo a função

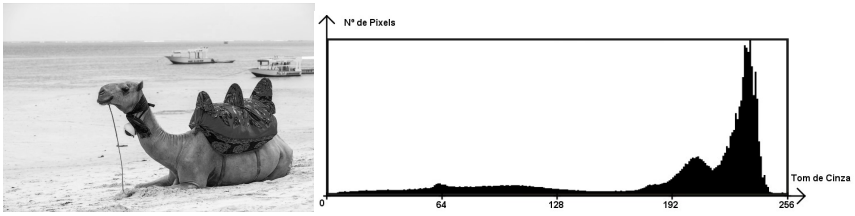
$$T(k) = 255 - k, \tag{1.4}$$

que gera o negativo de uma imagem. Na Figura 1.3, vemos a representação de algumas dessas transformações.

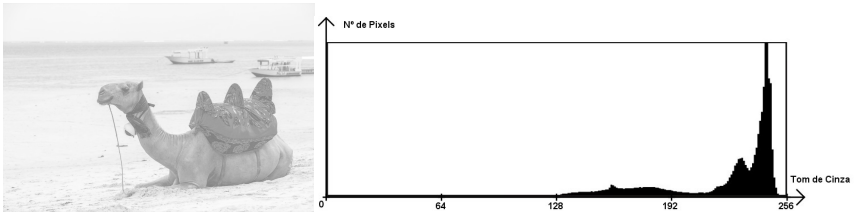
Quando trabalhamos com imagens coloridas, não conseguimos visualizar tão facilmente o histograma, visto que ele é o gráfico de uma função cujo domínio é $\{0, 1, 2, \ldots, 255\}^3$. Uma alternativa é construirmos a distribuição de cores da imagem, ou seja, plotar todas as cores utilizadas para a construção da mesma, onde cada ponto possui 3 coordenadas (escalas de verde, vermelho e azul). Além disso, para conseguirmos visualizar tal distribuição, nós a projetamos em um plano do espaço (por exemplo, no plano verde $\times$ vermelho). Na Figura 1.4, vemos a distribuição de cores da Figura 1.1a, utilizando uma clusterização com 200 cores[1].

Durante o nosso trabalho, nós realizamos transformações nas imagens através das suas distribuições de cores.

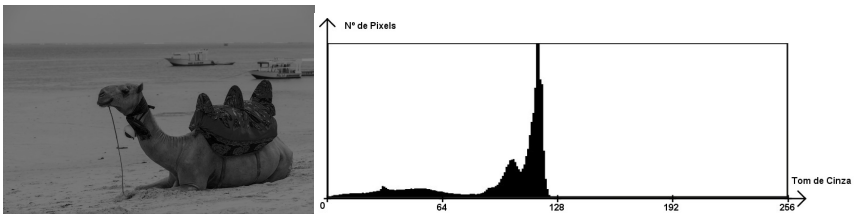
[1]Não utilizamos todas as cores da imagem para não poluir demais a distribuição de cores. Estudamos a clusterização de uma imagem na Seção 2.1.



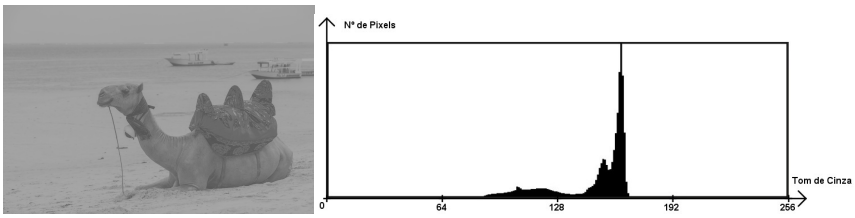
(a) Imagem original



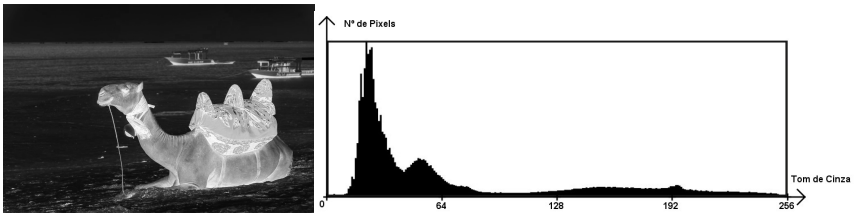
(b) Imagem clareada



(c) Imagem escurecida



(d) Imagem com menos contraste



(e) Imagem negativada

Figura 1.3: Diferentes transformações no histograma de uma imagem.

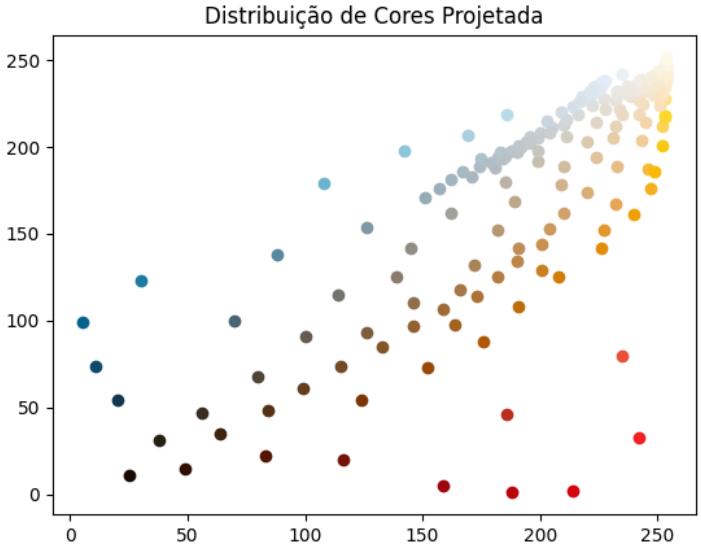


Figura 1.4: Distribuição de cores da Figura 1.1a projetada no plano Vermelho × Verde.

1.3 Transporte Ótimo

Vamos introduzir agora os conceitos que fundamentam o nosso estudo sobre transferência de cores entre duas imagens. Neste texto, trabalhamos com a versão discreta do transporte ótimo, portanto trazemos apenas ideias iniciais sobre o problema de transporte ótimo contínuo. Materiais mais aprofundados sobre o assunto podem ser encontrados em [16], [13] e [15].

O problema de Transporte Ótimo foi formalizado inicialmente no século XVIII pelo matemático Gaspard Monge. Ele consiste na seguinte situação: dados dois conjuntos X e Y e uma função custo $C : X \times Y \to \mathbb{R}^+$ tal que $C(x, y)$ indica o custo de se transportar uma unidade de massa da posição $x \in X$ para a posição $y \in Y$, queremos obter uma função $T : X \to Y$ que transporte toda a massa de X para Y com o menor custo total possível, sem divisão de massa.

Mais especificamente, dadas duas medidas de probabilidade[2] μ em X e ν

[2]Para aqueles que não estão habituados com o conceito de medida, recomendamos o material [1]. De qualquer maneira, como trabalhamos com a versão discreta do transporte

em Y, queremos minimizar um funcional da forma

$$\mathcal{C}[T] := \int_X C(x, T(x)) \, \mathrm{d}\mu(x) \tag{1.5}$$

dentre todas as aplicações mensuráveis $T : X \to Y$ que satisfazem $T_{\#}\mu = \nu$. A condição $T_{\#}\mu = \nu$ significa que a medida ν é o push-foward de μ pela aplicação T, ou seja,

$$\nu(B) := \mu(T^{-1}(B)) \text{ para todo } B \subseteq Y \text{ mensurável.} \tag{1.6}$$

Essa formulação de Monge possui alguns problemas, entre eles está o fato de que o problema nem sempre tem solução. De fato, se tivermos $X = \{1\}, Y = \{2, 3\}, \mu = \delta_1$ (onde δ_1 é a função delta de Dirac no ponto 1) e $\nu = (\delta_2 + \delta_3)/2$, temos que não é possível transportar a massa de X para Y sem dividi-la.

Uma forma de contornar esta situação é "relaxar" o problema, ou seja, permitir que a divisão de massa seja possível. Essa formulação de transporte relaxada foi feita pelo matemático Leonid Kantorovich em 1942. Basicamente, nesta situação, é possível que a massa em cada ponto $x \in X$ seja dividida e transportada para diversos pontos em Y, além de que cada ponto $y \in Y$ pode receber massa de vários pontos de X.

Matematicamente, o que temos é que o problema é descrito por uma medida de probabilidade $\gamma \in \mathcal{P}(X \times Y)$, onde $\mathcal{P}(X \times Y)$ indica o espaço das probabilidades que atuam sobre $X \times Y$. Ainda temos a restrição de que toda a massa de X deve ser transportada e que Y deve receber toda a massa possível, ou seja, para todo $A \subseteq X$ e $B \subseteq Y$ mensuráveis,

$$\gamma(A \times Y) = \mu(A) \text{ e } \gamma(X \times B) = \nu(B). \tag{1.7}$$

Dizemos que uma medida de probabilidade $\gamma \in \mathcal{P}(X \times Y)$ que satisfaz a equação (1.7) tem marginais μ e ν e a chamamos de um plano de transporte. Descrevemos então o problema de Kantorovich da seguinte forma: dadas duas medidas de probabilidade $\mu \in \mathcal{P}(X)$ e $\nu \in \mathcal{P}(Y)$, devemos minimizar

$$\mathcal{C}[\gamma] := \int_{X \times Y} C(x, y) \, \mathrm{d}\gamma(x, y) \tag{1.8}$$

dentre todas as probabilidades $\gamma \in \mathcal{P}(X \times Y)$ com marginais μ e ν.

ótimo, não precisamos nos aprofundar tanto nestes conceitos.

Na formulação de Kantorovich, sob hipóteses bastante gerais, sempre existe uma solução para o problema de otimização. Por exemplo, o problema que mencionamos anteriormente de o conjunto das medidas admissíveis ser vazio não acontece pois a medida produto $\gamma = \mu \times \nu$ é tal que $\gamma \in \mathcal{P}(X \times Y)$ e satisfaz (1.7).

Quando trabalhamos com o transporte ótimo em sua versão discreta, temos que o problema (1.8) se torna um problema de programação linear. Estudamos sobre problemas de programação linear na Seção 1.4 e sobre a versão discreta do transporte ótimo na Seção 2.2.

1.4 Programação Linear

Agora vamos discutir sobre problemas de programação linear. Eles serão importantes porque iremos resolver o problema de transferência de cores utilizando um algoritmo de programação linear.

Um problema de programação linear é um problema de otimização cujo objetivo é minimizar ou maximizar uma função linear z sob um conjunto de restrições dado por um sistema de equações e/ou inequações, também lineares.

Como exemplo, considere a situação onde queremos maximizar a função

$$z = 2x - 5y \tag{1.9}$$

de tal modo que x e y satisfaçam o sistema de inequações

$$\begin{aligned} 3x + 2y &\leq 20 \\ -x + 20y &\geq 10 \\ -3x + y &\leq 4. \end{aligned} \tag{1.10}$$

Como o sistema possui apenas duas variáveis, podemos representá-lo no plano $\mathbb{R}^2$ e cada inequação representa uma região diferente do plano. Na Figura 1.5, vemos a região relativa a cada uma das inequações.

Para que um par (x, y) seja solução do sistema de inequações (1.10), ele deve estar dentro das três regiões ao mesmo tempo. Seja R a região dada pela intersecção das três regiões da Figura 1.5. Chamamos essa região de *região admissível* do problema. Sobre a região admissível R, temos 3 possibilidades:

- R é limitada e não-vazia, de modo que conseguimos utilizar técnicas (como por exemplo o método Simplex) para encontrar um ponto que minimize ou maximize z.

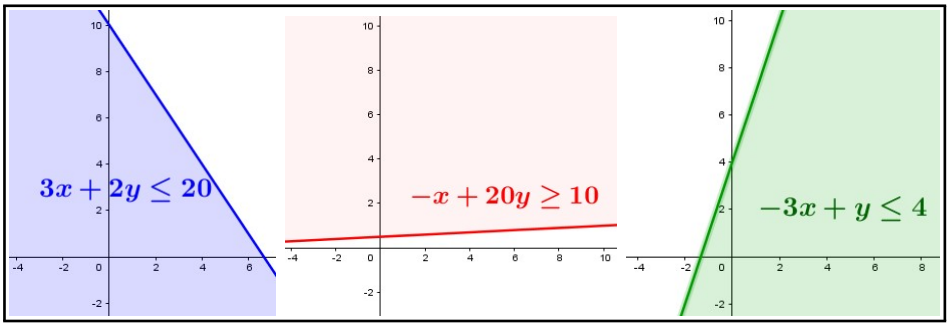


Figura 1.5: Regiões no plano relativas às desigualdades de (1.10).

- R é vazia, ou seja, o nosso sistema de inequações não tem solução. Neste caso, dizemos que o problema é *inviável*.
- R é ilimitada, logo pode ser que $\max z = +\infty$ ou $\min z = -\infty$. Neste caso dizemos que o problema é *ilimitado*.

Estamos interessados em problemas do primeiro tipo, onde R é não-vazia e limitada. Podemos ver pela Figura 1.6 que o nosso sistema de inequações gera

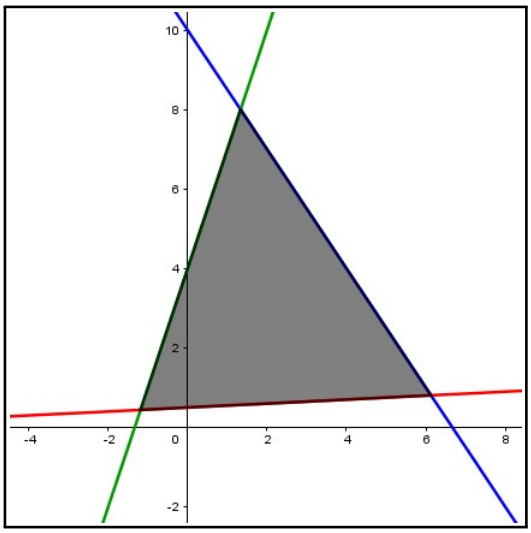


Figura 1.6: Região admissível do sistema de inequações (1.10).

uma região admissível R que realmente é não vazia e limitada. Nosso objetivo é então encontrar, dentro da região R, um par (x, y) que maximize a função

$z = 2x - 5y$. Para isso, podemos visualizar os diversos valores possíveis de z como curvas de nível no plano, de tal maneira que uma solução que otimize o problema estará contida na curva de maior nível que intersecte R. Na Figura 1.7, vemos como algumas curvas de nível intersectam R, podemos observar que a curva com maior nível passando por R é a reta $r : 2x - 5y = 255/31$, o ponto de intersecção rosa indica exatamente os valores de x e y que otimizam o problema. Realizando algumas contas, vemos que $(x, y) = (190/31, 25/31)$ é o ponto procurado.

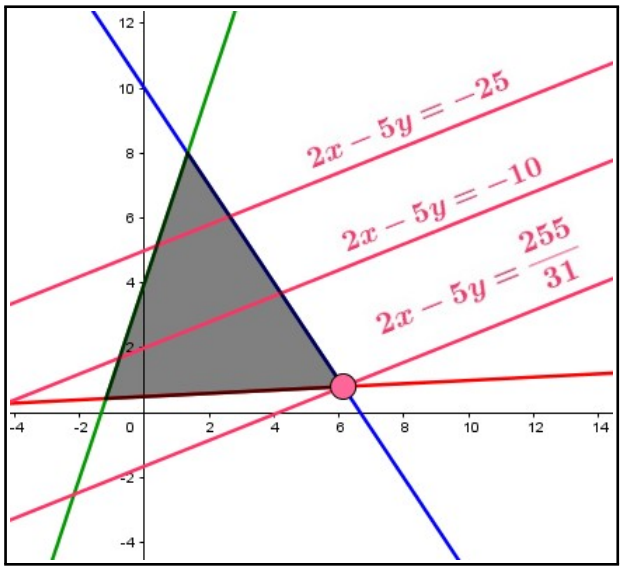


Figura 1.7: Curvas de nível da função $z = 2x - 5y$ usadas para determinar a solução minimizante do problema dentro da região admissível R.

No caso geral do problema, onde o sistema possui m inequações e n variáveis, podemos descrever o problema como

$$\max\{zx \mid Ax \leq b\}, \tag{1.11}$$

onde $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ e $Ax \leq b$ representa as inequações que descrevem a região admissível. Além disso, $z \in \mathbb{R}^n$ é o vetor linha formado pelos coeficientes da função linear a ser otimizada.

É possível mostrar que um mesmo problema de programação linear possui diversas formas equivalentes [14]. Os seguintes problemas são equivalentes no sentido de que qualquer um deles pode ser reduzido a outro da lista, sendo

apenas necessárias alterações na matriz A e/ou no vetor b.

$$\begin{aligned}
&\text{(i) } \max\{zx \,|\, Ax \leq b\} \\
&\text{(ii) } \max\{zx \,|\, x \geq 0, Ax \leq b\} \\
&\text{(iii) } \max\{zx \,|\, x \geq 0, Ax = b\} \\
&\text{(iv) } \min\{zx \,|\, Ax \geq b\} \\
&\text{(v) } \min\{zx \,|\, x \geq 0, Ax \geq b\} \\
&\text{(vi) } \min\{zx \,|\, x \geq 0, Ax = b\}.
\end{aligned} \tag{1.12}$$

Com isso, um problema de minimização pode ser visto como um problema de maximização e vice-versa.

Temos que se a região admissível R for não-vazia e limitada, então ela será um politopo[3] convexo e compacto de dimensão menor ou igual a n; de fato, R é convexa porque é a intersecção de conjuntos convexos e é compacta porque é limitada e também a intersecção de conjuntos fechados. Um resultado importante nos mostra que, se o problema for admissível, então existirá um vértice de R que irá otimizar o problema.

Teorema 1.1. ***(Teorema Fundamental da Programação Linear)***
Seja R a região admissível compacta de um problema de programação linear $\mathcal{P}$, de dimensão n. Seja $z : \mathbb{R}^n \to \mathbb{R}$ a função linear a ser otimizada. Então existe um vértice de $y \in R$ que é solução do problema.

Demonstração. Sem perda de generalidade, vamos supor que $\mathcal{P}$ é um problema de minimização. Por R ser um politopo compacto e pela continuidade de z, temos que existe uma solução minimizante y_0 do problema.

Se z for uma função constante, temos que todo ponto de R é um solução minimizante, em particular seus vértices. Se z não for uma função linear constante, então temos que $z(x) = k$ é um hiperplano de $\mathbb{R}^n$, para todo $k \in \mathbb{R}$. Seja $k_0 = z(y_0)$.

Vamos supor que y_0 não é um vértice de R. Temos então três possibilidades:

1. y_0 pertence a alguma aresta a de R e o hiperplano $H : z(x) = k_0$ é paralelo a a. Neste caso, temos que a está contida em H, logo os vértices de a também estão em H e portanto são soluções minimizantes de $\mathcal{P}$.

[3] Um politopo é a generalização em dimensão d de um polígono. As faces de um politopo de dimensão d serão também politopos, de dimensão $d-1$.

2. y_0 pertence a alguma aresta a de R e o hiperplano $H : z(x) = k_0$ não é paralelo a a. Neste caso H divide R em duas regiões não-vazias disjuntas R_1 e R_2, portanto existe $\varepsilon > 0$ tal que $H' : z(x) = k_0 - \varepsilon$ é um hiperplano que intersecciona R_1 ou R_2, ou seja, y_0 não é uma solução minimizante de $\mathcal{P}$, absurdo.

3. y_0 é um ponto interior de R. Neste caso, o hiperplano $H : z(x) = k_0$ divide R em duas regiões não-vazias disjuntas R_1 e R_2. Assim como no item anterior, concluímos que y_0 não é uma solução minimizante de $\mathcal{P}$, absurdo. □

O Teorema 1.1 nos dá uma possível maneira de encontrar uma solução ótima para qualquer problema de programação linear: basta aplicar a função z em todos os vértices da região admissível R e tomar aquele com o maior ou menor valor. Infelizmente, conforme aumentamos a dimensão do problema, o número de vértices a serem testados se torna extremamente grande, o que inviabiliza o cálculo por este método, pois tem um custo computacional muito alto.

Felizmente, existem métodos mais práticos e com menor custo computacional para resolver problemas de programação linear, como passamos a descrever nas próximas seções.

1.5 Método Simplex

O Método Simplex é um dos métodos mais comuns para solucionar problemas de programação linear. Ele soluciona problemas de maximização quando todas as variáveis são positivas e quando todas as restrições são limitantes superiores positivas, ou seja, soluciona problemas do tipo

$$\max\{zx \,|\, x \geq 0, Ax \leq b\}, \tag{1.13}$$

onde devemos ter $b \geq 0$.

Como vimos anteriormente, a não ser pela restrição $b \geq 0$, temos que outros problemas de programação linear são equivalentes ao tipo que é resolvido pelo Método Simplex. Portanto, podemos adaptá-los para que se encaixem no algoritmo. Para problemas que não obedeçam a restrição $b \geq 0$, podemos utilizar uma versão mais elaborada do Método Simplex, chamada de Método

Simplex de Duas Fases, a ser estudado na Seção 1.6. Um material mais completo e detalhado sobre o Método Simplex e sobre o Método Simplex de duas fases pode ser encontrado em [8].

Na prática, o que o Método Simplex faz é, começando por um vértice inicial v_0, percorrer os vértices de R, indo de v_i a v_{i+1} de modo que z sempre aumente durante este percurso. Quando todos os vértices adjacentes de um determinado vértice $v^* \in R$ determinarem valores menores ou iguais a $z(v^*)$, pode-se concluir que v^* é uma solução maximizante do problema.

Vamos ilustrar o Método Simplex utilizando um exemplo: seja $\mathcal{P}$ o problema de programação linear que pretende maximizar a função objetivo $z : \mathbb{R}^2 \to \mathbb{R}$ dada por

$$z(x_1, x_2) = 5x_1 + 6x_2, \tag{1.14}$$

dadas as restrições

$$\begin{aligned} &x_1 + 2x_2 \leq 60 \\ &5x_1 + x_2 \leq 150 \\ &x_1 + x_2 \leq 35 \\ &x_1, x_2 \geq 0. \end{aligned} \tag{1.15}$$

Inicialmente, devemos converter a função objetivo e as restrições em equações. Na função objetivo devemos considerar z como sendo uma variável adicional, enquanto nas restrições devemos adicionar variáveis de sobra s_i para transformar cada inequação em uma equação. Como resultado temos o seguinte sistema de equações:

$$\begin{array}{rcrcrcrcrcr} z & - & 5x_1 & - & 6x_2 & & & & & = 0 & \\ & & x_1 & + & 2x_2 & + & s_1 & & & = 60 & (1.16) \\ & & 5x_1 & + & x_2 & & + & s_2 & & = 150 & \\ & & x_1 & + & x_2 & & & & + \quad s_3 & = 35. & \end{array}$$

É importante observar que todas as variáveis do sistema (z, x_1, x_2, s_1, s_2 e s_3) devem ser maiores ou iguais a 0. Caso contrário, é necessário fazer alguma substituição de variáveis[4]. Também é necessário que todos os valores à direita das igualdades sejam positivos, caso contrário devemos inverter as desigualdades. Após isso, representamos o sistema de equações em uma

[4]Por exemplo, se $x_1 \geq -3$, definimos $y_1 = x_1 + 3$ e temos $y_1 \geq 0$.

matriz:

$$\begin{array}{c} \begin{matrix} z & x_1 & x_2 & s_1 & s_2 & s_3 \end{matrix} \\ \left[\begin{array}{cccccc|c} 1 & -5 & -6 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 & 60 \\ 0 & 5 & 1 & 0 & 1 & 0 & 150 \\ 0 & 1 & 1 & 0 & 0 & 1 & 35 \end{array}\right] \end{array} \tag{1.17}$$

Para resolver o nosso problema, devemos agora classificar as nossas variáveis. A variável z e as variáveis de sobra são classificadas como *variáveis básicas*. As outras variáveis são classificadas como *variáveis não-básicas*. Podemos ver essa classificação na Figura 1.8.

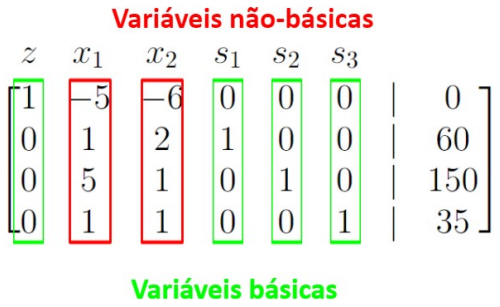


Figura 1.8: Variáveis Básicas e Variáveis Não-Básicas

Podemos observar que as colunas relativas às variáveis básicas possuem apenas um valor positivo e todos os outros valores nulos. Conseguimos assim associar uma variável básica a cada linha da nossa matriz, justamente nesse único valor positivo da coluna.

$$\begin{array}{c} \begin{matrix} & z & x_1 & x_2 & s_1 & s_2 & s_3 \end{matrix} \\ \begin{matrix} z \\ s_1 \\ s_2 \\ s_3 \end{matrix} \left[\begin{array}{cccccc|c} 1 & -5 & -6 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 & 60 \\ 0 & 5 & 1 & 0 & 1 & 0 & 150 \\ 0 & 1 & 1 & 0 & 0 & 1 & 35 \end{array}\right] \end{array} \tag{1.18}$$

Temos que uma *solução básica* do problema é dada quando anulamos todas as variáveis não-básicas e resolvemos o sistema. Na prática, temos que as variáveis básicas são justamente uma base para uma solução básica do

problema. A proposta do Método Simplex é, a cada etapa, trocar o papel de uma variável básica com o de uma não-básica, de forma a melhorar o valor da função objetivo. A seguir, fazemos essas trocas através de operações elementares na nossa matriz.

Inicialmente, temos que a solução básica atrelada à matriz (1.18) é

$$z = 0,\ x_1 = 0,\ x_2 = 0,\ s_1 = 60,\ s_2 = 150,\ s_3 = 35. \tag{1.19}$$

O ponto $(0, 0)$ é um dos vértices da região admissível R de $\mathcal{P}$, como podemos ver na Figura 1.9. Já vimos anteriormente que um dos vértices de R será uma solução ótima de $\mathcal{P}$. Portanto, devemos percorrer os vértices de R de forma que um novo vértice sempre tenha valor maior que o anterior, o processo irá se encerrar quando não for possível encontrar um novo vértice com valor maior que o atual.

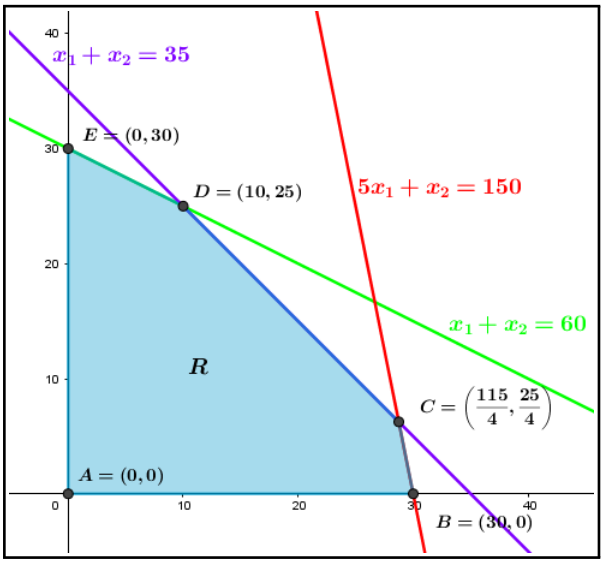


Figura 1.9: Região Admissível de $\mathcal{P}$.

Analisando a matriz (1.18), vemos que um aumento nos valores de x_1 ou x_2 acarretaria em um aumento no valor total de z (pois $z - 5x_1 - 6x_2 = 0$), ou seja, essa primeira solução básica que encontramos não é uma solução ótima. Seguimos o processo da seguinte maneira:

- Devemos escolher um elemento da matriz para ser pivotado[5], de forma

[5]Por pivotar, queremos dizer que realizamos operações elementares de matrizes a fim de anular todos os valores da coluna, com exceção do elemento a ser pivotado.

a alterar o valor resultante de z.

- Para escolher a coluna a ser pivotada, selecionamos aquela que possuir o valor negativo de maior módulo na primeira linha (no nosso caso, devemos escolher a terceira coluna da matriz, correspondente à variável x_2). Escolhemos dessa maneira porque pivotar esta coluna resulta no maior aumento possível da função objetivo. Também seria possível escolher qualquer outra coluna com valor negativo na primeira linha, mas provavelmente iríamos precisar de mais iterações do método para chegar ao resultado otimizado.

- Para descobrir a linha a ser pivotada, realizamos em cada linha a divisão do elemento da última coluna pelo elemento da coluna a ser pivotada. Aquela divisão que resultar no valor positivo de menor módulo irá indicar o elemento escolhido. Fazemos a escolha dessa maneira para evitar que, após a pivotagem, obtenhamos algum valor negativo na última coluna[6]. Na nossa matriz, temos as seguintes divisões:

  - Linha 2: $\frac{60}{2} = 30$.

  - Linha 3: $\frac{150}{1} = 150$.

  - Linha 4: $\frac{35}{1} = 35$.

O menor valor encontrado foi o da linha 2, logo devemos pivotar o elemento da linha 2, coluna 3. Sendo assim, trocamos os papéis de s_1 e de x_2. Após pivotar este elemento, temos a seguinte matriz resultante[7]:

$$\begin{array}{c} \\ z \\ x_2 \\ s_2 \\ s_3 \end{array} \begin{array}{c} \begin{array}{cccccc} z & x_1 & x_2 & s_1 & s_2 & s_3 \end{array} \\ \left[\begin{array}{cccccc|c} 1 & -2 & 0 & 3 & 0 & 0 & 180 \\ 0 & 1 & 2 & 1 & 0 & 0 & 60 \\ 0 & 9 & 0 & -1 & 2 & 0 & 240 \\ 0 & 1 & 0 & -1 & 0 & 2 & 10 \end{array}\right] \end{array} \tag{1.20}$$

Após essa pivotagem, as variáveis básicas são z, x_2, s_2 e s_3, enquanto x_1 e s_1 são as variáveis não-básicas. A solução básica obtida a partir da matriz

[6] Na prática, apenas a primeira linha da matriz pode apresentar valor negativo na última coluna, pois a função objetivo pode assumir valores menores do que zero. Todas as outras componentes da última coluna são referentes ao vetor b, que precisa ser não negativo.

[7] Realizamos as seguintes operações básicas na matriz (1.17): $L_1 \to L_1 + 3L_2$, $L_2 \to L_2$, $L_3 \to 2L_3 - L_2$, $L_4 \to 2L_4 - L_2$.

resultante é a seguinte:

$$z = 180,\ x_1 = 0,\ x_2 = 30,\ s_1 = 0,\ s_2 = 120,\ s_3 = 5. \tag{1.21}$$

Essa solução está relacionada ao ponto $(0, 30)$, que também é um vértice de R conforme podemos observar na Figura 1.9. Além disso, conforme esperado, o valor de z aumentou de 0 para 180, mostrando que nos aproximamos da solução ótima. Entretanto, ainda existe um elemento negativo na linha 1 (-2, relativo à variável x_1), devemos então pivotar esta coluna. Para descobrir a linha a ser pivotada, temos as seguintes divisões:

- Linha 2: $\frac{60}{1} = 60$.
- Linha 3: $\frac{240}{9} = \frac{80}{3}$.
- Linha 4: $\frac{10}{1} = 10$.

Temos que o menor valor está localizado na linha 4. Logo, o elemento a ser pivotado é o da linha 4, coluna 2. Trocamos assim os papéis de x_1 e s_3. Após realizar a pivotagem, obtemos a seguinte matriz[8]:

$$\begin{array}{c} \\ z \\ x_2 \\ s_2 \\ x_1 \end{array} \begin{array}{c} \begin{array}{cccccc} z & x_1 & x_2 & s_1 & s_2 & s_3 \end{array} \\ \left[\begin{array}{cccccc|c} 1 & 0 & 0 & 1 & 0 & 4 & 200 \\ 0 & 0 & 2 & 2 & 0 & -2 & 50 \\ 0 & 0 & 0 & 8 & 2 & -18 & 150 \\ 0 & 1 & 0 & -1 & 0 & 2 & 10 \end{array}\right] \end{array}. \tag{1.22}$$

Agora as variáveis básicas serão z, x_1, x_2 e s_2, enquanto as variáveis não-básicas serão s_1 e s_3. A solução básica associada à matriz é

$$z = 200,\ x_1 = 10,\ x_2 = 25,\ s_1 = 0,\ s_2 = 75,\ s_3 = 0. \tag{1.23}$$

Esta solução está relacionada ao vértice $(10, 25)$ da região admissível R. Além disso, podemos ver que todos os valores da primeira linha são nulos ou positivos. Isso implica que a solução básica encontrada é ótima e terminamos o nosso algoritmo.

Para problemas de minimização, basta multiplicarmos toda a função objetivo por (-1). Por exemplo, se o objetivo de $\mathcal{P}$ fosse minimizar a função z, poderíamos definir o problema equivalente $\mathcal{P}'$ que visa maximizar $z'(x_1, x_2) = -5x_1 - 6x_2$ e o restante do algoritmo segue analogamente.

[8] Realizamos as seguintes operações básicas sobre a matriz (1.20): $L_1 \to L_1 + 2L_4$, $L_2 \to L_2 - L_4$, $L_3 \to L_3 - 9L_4$, $L_4 \to L_4$.

1.6 Método Simplex de Duas Fases

Uma mudança que realmente impacta no método é, ao adaptarmos o problema para que tenhamos $Ax \leq b$, encontrar um vetor b que não satisfaça $b \geq 0$. Vamos ver isso através de outro exemplo: seja $\mathcal{Q}$ o problema de minimizar a função

$$z(x_1, x_2) = -x_1 + x_2 \tag{1.24}$$

dadas as restrições

$$\begin{aligned} -x_1 + 3x_2 &\leq 11 \\ x_1 + x_2 &\leq 9 \\ 2x_1 - x_2 &\leq 6 \\ 2x_1 + 3x_2 &\geq 14 \\ x_1, x_2 &\geq 0. \end{aligned} \tag{1.25}$$

A complicação que temos está justamente na restrição $2x_1 + 3x_2 \geq 14$, pois ao multiplicarmos a inequação por (-1), temos $-2x_1 - 3x_2 \leq -14$, ou seja, uma componente de b é menor do que 0. Na prática, isso faz com que o ponto $(0, 0)$ não seja um vértice da região admissível R, impedindo que o algoritmo seja iniciado nesse ponto. Na Figura 1.10, vemos a região R e o papel dessa desigualdade.

Para resolver o problema $\mathcal{Q}$, utilizamos o *Método Simplex de Duas Fases*. A ideia é utilizar variáveis auxiliares para encontrarmos algum vértice de R (Fase 1). Após isso, seguimos o método Simplex normalmente, como foi explicado anteriormente (Fase 2).

Inicialmente, transformamos todas as inequações em equações, utilizando variáveis auxiliares s_1, s_2, s_3 e s_4. Entretanto, o coeficiente da variável s_4 é um valor negativo, impossibilitando que ela seja uma variável básica. Devemos então somar uma variável artificial $0 \leq a_1 \leq s_4$ na quarta desigualdade, para que ela faça o papel de variável básica. Assim, as variáveis básicas do nosso problema são z, s_1, s_2, s_3 e a_1. Ainda, mudamos a forma da função objetivo para maximizar $z^* = x_1 - x_2$, pois o Método Simplex não soluciona diretamente problemas de minimização. Obtemos então o sistema de

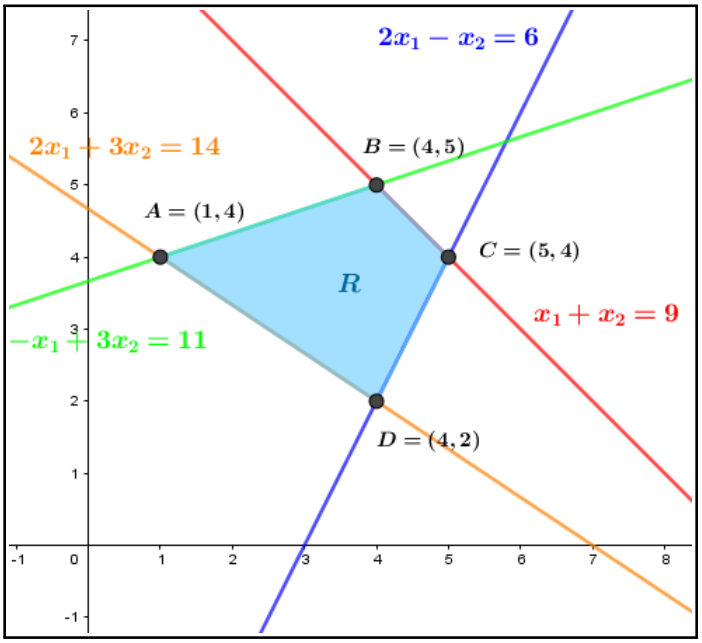


Figura 1.10: Região Admissível de $\mathcal{Q}$.

equações

$$\begin{array}{rrrrrrrrrrrrrrr} z^* & - & x_1 & + & x_2 & & & & & & & & & = 0 & \\ & - & x_1 & + & 3x_2 & + & s_1 & & & & & & & = 11 & \\ & & x_1 & + & x_2 & & & + & s_2 & & & & & = 9 & (1.26) \\ & & 2x_1 & - & x_2 & & & & & + & s_3 & & & = 6 & \\ & + & 2x_1 & + & 3x_2 & & & & & & & - & s_4 + a_1 & = 14. & \end{array}$$

Para encontrar a primeira solução básica do nosso problema, devemos primeiramente resolver um problema artificial, dado por maximizar a função $w = -a_1$ (se tivéssemos n variáveis auxiliares $a_1, \ldots, a_n$, teríamos que maximizar $w = -\sum_{i=1}^{n} a_i$).

Temos que a_1 é um variável artificial, portanto não queremos utilizá-la na nossa solução final. Por esse motivo que devemos maximizar w, pois o seu valor máximo é obtido justamente quando $a_1 = 0$.

Como queremos primeiramente resolver o problema artificial, não consideramos na primeira fase do problema a linha referente a z^* na nossa matriz.

Ainda, como a_1 é uma variável básica, precisamos que o seu coeficiente na primeira linha seja 0. Desta forma, realizamos alguns cálculos com a equação $w = -a_1$ antes de adicioná-la à matriz:

$$\begin{aligned} 2x_1 + 3x_2 - s_4 + a_1 &= 14 \\ a_1 &= 14 - 2x_1 - 3x_2 + s_4 \\ w &= -a_1 \\ w &= -14 + 2x_1 + 3x_2 - s_4 \\ w - 2x_1 - 3x_2 + s_4 &= -14. \end{aligned} \tag{1.27}$$

Assim, a matriz que representa o problema inicial é:

$$\begin{array}{c} \\ w \\ s_1 \\ s_2 \\ s_3 \\ a_1 \end{array} \begin{array}{c} \begin{array}{cccccccc} w & x_1 & x_2 & s_1 & s_2 & s_3 & s_4 & a_1 \end{array} \\ \left[\begin{array}{cccccccc|c} 1 & -2 & -3 & 0 & 0 & 0 & 1 & 0 & -14 \\ 0 & -1 & 3 & 1 & 0 & 0 & 0 & 0 & 11 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 9 \\ 0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 6 \\ 0 & 2 & 3 & 0 & 0 & 0 & -1 & 1 & 14 \end{array}\right] \end{array} \tag{1.28}$$

O valor do resultado na primeira linha da matriz deve ser maximizado para obtermos a nossa primeira solução básica. Utilizando a mesma análise do caso de maximização, vemos que devemos pivotar o elemento que está na linha 2 e na coluna 3, trocando os papéis de x_2 e s_1. Após a pivotagem, obtemos a seguinte matriz[9]:

$$\begin{array}{c} \\ w \\ x_2 \\ s_2 \\ s_3 \\ a_1 \end{array} \begin{array}{c} \begin{array}{cccccccc} w & x_1 & x_2 & s_1 & s_2 & s_3 & s_4 & a_1 \end{array} \\ \left[\begin{array}{cccccccc|c} 1 & -3 & 0 & 1 & 0 & 0 & 1 & 0 & -3 \\ 0 & -1 & 3 & 1 & 0 & 0 & 0 & 0 & 11 \\ 0 & 4 & 0 & -1 & 3 & 0 & 0 & 0 & 16 \\ 0 & 5 & 0 & 1 & 0 & 3 & 0 & 0 & 29 \\ 0 & 3 & 0 & -1 & 0 & 0 & -1 & 1 & 3 \end{array}\right] \end{array} \tag{1.29}$$

Ainda não maximizamos w, pois temos valores negativos na primeira linha da nossa matriz. Continuamos então o processo, pivotando o elemento da

[9] $L_1 \to L_1 + L_2, L_2 \to L_2, L_3 \to 3L_3 - L_2, L_4 \to 3L_4 + L_2, L_5 \to L_5 - L_2$.

linha 5 e da coluna 2, trocando os papéis de x_1 e de a_1. Após a pivotagem, obtemos a matriz[10], com variáveis básicas x_1, x_2, s_2 e s_3.

$$\begin{array}{c} \\ w \\ x_2 \\ s_2 \\ s_3 \\ x_1 \end{array} \begin{array}{c} \begin{array}{cccccccc} w & x_1 & x_2 & s_1 & s_2 & s_3 & s_4 & a_1 \end{array} \\ \left[\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 9 & 2 & 0 & 0 & -1 & 1 & 36 \\ 0 & 0 & 0 & 1 & 9 & 0 & 4 & -4 & 36 \\ 0 & 0 & 0 & 8 & 0 & 9 & 5 & -5 & 72 \\ 0 & 3 & 0 & -1 & 0 & 0 & -1 & 1 & 3 \end{array}\right] \end{array}. \tag{1.30}$$

Agora já não temos nenhum elemento negativo na primeira linha da matriz, significando que conseguimos maximizar a função w. Podemos continuar o processo removendo a coluna referente a_1 (coluna 8), trocando a coluna de w por z^* e maximizando z^* através da primeira linha da matriz (ou seja, trocando os valores da primeira linha da matriz). Nossa nova matriz será

$$\begin{array}{c} \\ z^* \\ x_2 \\ s_2 \\ s_3 \\ x_1 \end{array} \begin{array}{c} \begin{array}{ccccccccc} z^* & x_1 & x_2 & s_1 & s_2 & s_3 & s_4 \end{array} \\ \left[\begin{array}{ccccccc|c} 1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 2 & 0 & 0 & -1 & 36 \\ 0 & 0 & 0 & 1 & 9 & 0 & 4 & 36 \\ 0 & 0 & 0 & 8 & 0 & 9 & 5 & 72 \\ 0 & 3 & 0 & -1 & 0 & 0 & -1 & 3 \end{array}\right] \end{array}. \tag{1.31}$$

Estamos quase finalizando a Fase 1 do problema. Como x_1 e x_2 são variáveis básicas, devemos zerar seus valores na primeira linha através de operações elementares. Começamos zerando o valor correspondente a x_1[11]:

$$\begin{array}{c} \\ z^* \\ x_2 \\ s_2 \\ s_3 \\ x_1 \end{array} \begin{array}{c} \begin{array}{ccccccccc} z^* & x_1 & x_2 & s_1 & s_2 & s_3 & s_4 \end{array} \\ \left[\begin{array}{ccccccc|c} 3 & 0 & 3 & -1 & 0 & 0 & -1 & 3 \\ 0 & 0 & 9 & 2 & 0 & 0 & -1 & 36 \\ 0 & 0 & 0 & 1 & 9 & 0 & 4 & 36 \\ 0 & 0 & 0 & 8 & 0 & 9 & 5 & 72 \\ 0 & 3 & 0 & -1 & 0 & 0 & -1 & 3 \end{array}\right] \end{array}. \tag{1.32}$$

[10] $L_1 \to L_1 + L_5, L_2 \to 3L_2 + L_5, L_3 \to 3L_3 - 4L_5, L_4 \to 3L_4 - 5L_5, L_5 \to L_5$.
[11] $L_1 \to 3L_1 + L_5, L_2 \to L_2, L_3 \to L_3, L_4 \to L_4, L_5 \to L_5$.

Depois zeramos o valor correspondente a x_2[12]:

$$\begin{array}{c} \\ z^* \\ x_2 \\ s_2 \\ s_3 \\ x_1 \end{array} \begin{array}{c} \begin{array}{ccccccc} z^* & x_1 & x_2 & s_1 & s_2 & s_3 & s_4 \end{array} \\ \left[\begin{array}{ccccccc|c} 9 & 0 & 0 & -5 & 0 & 0 & -2 & -27 \\ 0 & 0 & 9 & 2 & 0 & 0 & -1 & 36 \\ 0 & 0 & 0 & 1 & 9 & 0 & 4 & 36 \\ 0 & 0 & 0 & 8 & 0 & 9 & 5 & 72 \\ 0 & 3 & 0 & -1 & 0 & 0 & -1 & 3 \end{array}\right] \end{array}. \tag{1.33}$$

Aqui encerramos a Fase 1 do nosso problema. Agora resta apenas realizar a Fase 2 e solucionar o problema de maximização de z^* utilizando o Método Simplex normalmente. Observe que já temos uma solução básica inicial associada a esta matriz, que é dada por

$$x_1 = 1,\ x_2 = 4,\ z^* = -3. \tag{1.34}$$

Assim como no problema $\mathcal{P}$, essa solução está localizada justamente em um vértice de R, como podemos ver na Figura 1.10. Entretanto, essa solução não é ótima, afinal temos valores negativos na primeira linha da matriz. Continuamos o processo pivotando o elemento localizado na linha 4 e na coluna 4, trocando os papés de s_1 e s_3, resultando na matriz[13]

$$\begin{array}{c} \\ z^* \\ x_2 \\ s_2 \\ s_1 \\ x_1 \end{array} \begin{array}{c} \begin{array}{ccccccc} z^* & x_1 & x_2 & s_1 & s_2 & s_3 & s_4 \end{array} \\ \left[\begin{array}{ccccccc|c} 72 & 0 & 0 & 0 & 0 & 45 & 9 & 144 \\ 0 & 0 & 36 & 0 & 0 & -9 & -9 & 72 \\ 0 & 0 & 0 & 0 & 72 & -9 & 27 & 216 \\ 0 & 0 & 0 & 8 & 0 & 9 & 5 & 72 \\ 0 & 24 & 0 & 0 & 0 & 9 & -3 & 96 \end{array}\right] \end{array}. \tag{1.35}$$

Esta matriz está associada à solução básica

$$x_1 = 4,\ x_2 = 2,\ z^* = 2, \tag{1.36}$$

que é justamente a solução ótima do problema, pois já não há mais valores negativos na primeira linha da matriz. Observe que o ponto $(4, 2)$ é um dos vértices de R, como podemos ver na Figura 1.10. Terminamos assim

[12] $L_1 \to 3L_1 - L_2, L_2 \to L_2, L_3 \to L_3, L_4 \to L_4, L_5 \to L_5$

[13] $L_1 \to 8L_1 + 5L_4, L_2 \to 4L_2 - L_4, L_3 \to 8L_3 - L_4, L_4 \to L_4, L_5 \to 8L_5 + L_4$.

o processo e encontramos a nossa solução maximizante de z^* e portanto minimizante de z, que assume o valor $-4 + 2 = -2$.

Algumas outras variações são possíveis para definir um problema de Programação Linear. Um exemplo seria exigir que uma ou mais das variáveis x_i assuma apenas valores inteiros ou binários, ao invés de valores reais. Nestes casos, estaremos trabalhando com problemas de Programação Linear Inteira (PLI). Problemas desse tipo são muito mais difíceis de resolver e possuem métodos próprios de resolução, como o *Método Branch and Bound* [2] ou o *Método Branch and Cut* [11]. No algoritmo utilizado em nosso código de programação, utilizamos um método chamado *COIN-OR Branch and Cut Solver* [12], que é capaz de resolver problemas PLI. Entretanto, como o problema que resolvemos não tem nenhuma exigência adicional sobre as variáveis (podendo assumir valores reais), este método se comporta exatamente da mesma maneira que o Método Simplex (ou o Método Simplex de Duas Fases, quando necessário).

Capítulo 2

O Problema da Transferência de Cores

Neste capítulo, discutimos os principais conceitos usados para solucionar o problema de transferência de cores entre duas imagens. Começamos abordando a clusterização das imagens na Seção 2.1, a fim de diminuir o custo computacional do problema. Na Seção 2.2, adaptamos os conceitos de transporte ótimo vistos na Seção 1.3 para um espaço discreto, afinal temos um número finito de pixels em uma imagem. Na Seção 2.3, estudamos o relaxamento do problema de transporte, enfraquecendo a restrição relativa à conservação de massa entre os pontos. Na Seção 2.4, construímos uma forma de regularizar o problema de transporte, instituindo uma penalidade para mapas mais irregulares. Tanto o relaxamento como a regularização do problema de transporte são importantes para melhorar o resultado visual da imagem final. Por fim, na Seção 2.5 realizamos um pós processamento na imagem obtida para recuperar informações que de certa forma foram perdidas durante o processo de clusterização.

2.1 Clusterização de Imagens

Existem diversos possíveis tamanhos para uma imagem, ou seja, o número de pixels com que podemos trabalhar varia muito. Ainda, quanto maior for o número de pixels de uma imagem, maior será o tempo computacional necessário para processá-la. Uma possível solução para isso é realizar a clusterização da mesma.

Ao clusterizar uma imagem, devemos escolher um número natural N, que representa a quantidade de cores diferentes que a nossa imagem pode ter após a clusterização (essas N cores escolhidas serão em geral muito semelhantes às N cores mais dominantes da imagem).

No nosso contexto, ao fazer a clusterização das imagens base e modelo, conseguimos realizar a transformação entre listas com N elementos, muito menor do que o número total de pixels das imagens. Ao utilizar menos elementos na nossa transformação, conseguimos acelerar o processo e diminuir o custo computacional.

Para realizar a clusterização das nossas imagens, utilizamos um método chamado k-means, que funciona da seguinte maneira:

1. Dada uma lista de pontos $M = (x_1, x_2, \ldots, x_n)$, com $x_i \in \mathbb{R}^3$ escolhemos N pontos em posições aleatórias $c_1, c_2, \ldots, c_N \in \mathbb{R}^3$, que chamamos de centroides.

2. Cada ponto de M é associado ao centroide mais próximo, formando assim N nuvens (subconjuntos de M).

3. Alteramos a posição de cada centroide para o centro de massa da sua respectiva nuvem.

4. Como a posição dos centroides mudou, é possível que as associações entre os pontos e os centroides mudem, então fazemos uma reassociação.

5. Repetimos os passos 3 e 4 até a convergência[1] ou por um número determinado de vezes.

6. Após o fim do processo, contabilizamos o somatório S da Distância Euclidiana entre cada ponto e seu centroide associado, dada por

$$S = \sum_{x_i \in M} \|x_i - C(x_i)\|^2, \tag{2.1}$$

onde $C(x_i)$ indica o centroide a qual foi associado x_i.

[1]Por convergência, queremos dizer que de um passo para o outro a configuração dos pontos não muda, ou seja, a posição dos centroides continua a mesma e a associação dos pontos a cada centroide também não muda. A partir deste momento a configuração dos pontos no método continua sempre a mesma.

Observação 2.1. No passo 2 é possível que alguma das posições c_k seja tal que nenhum dos pontos de M seja associado a c_k. Logo, das N nuvens formadas com pontos do conjunto M, algumas poderiam ser vazias. Para evitar isso, podemos exigir que as posições iniciais de $c_1, c_2, \ldots, c_N$ sejam escolhidas dentro do próprio conjunto M.

Observação 2.2. No passo 4 podemos convencionar que a reassociação de um ponto de M ao centroide mais próximo só é feita se essa reassociação diminuir estritamente a distância. Isto é, no caso de haver empate, mantemos a associação atual.

Um resultado importante nos diz que o algoritmo k-means sempre converge em um número finito de passos.

Teorema 2.3. *O algoritmo k-means sempre converge em um número finito de passos.*

Demonstração. Para qualquer clusterização com centroides associados a cada nuvem, podemos definir o correspondente valor de S pela equação (2.1).

Inicialmente observamos que ao aplicarmos o passo 3 do nosso algoritmo, o valor de S não aumenta. Isso ocorre porque, ao tomarmos uma nuvem de pontos $z_1, z_2, \ldots, z_m$ e o seu centro de massa

$$\overline{z} = \frac{1}{m}\sum_{k=1}^{m} z_k, \tag{2.2}$$

então temos que

$$\sum_{k=1}^{m} \|z_k - z\|^2 \geq \sum_{k=1}^{m} \|z_k - \overline{z}\|^2, \tag{2.3}$$

para qualquer $z \in \mathbb{R}^3$.

De fato,

$$\begin{aligned}
\sum_{k=1}^{m}\|z_k - z\|^2 &= \sum_{k=1}^{m}\|z_k - \overline{z} + \overline{z} - z\|^2 \\
&= \sum_{k=1}^{m}\left(\|z_k - \overline{z}\|^2 + \|\overline{z} - z\|^2 + 2\langle z_k - \overline{z}, \overline{z} - z\rangle\right) \\
&\geq \sum_{k=1}^{m}\left(\|z_k - \overline{z}\|^2 + 2\langle z_k - \overline{z}, \overline{z} - z\rangle\right) \\
&= \sum_{k=1}^{m}\|z_k - \overline{z}\|^2 + 2\sum_{k=1}^{m}\left(\langle z_k, \overline{z}\rangle - \langle z_k, z\rangle - \langle \overline{z}, \overline{z}\rangle + \langle \overline{z}, z\rangle\right) \\
&= \sum_{k=1}^{m}\|z_k - \overline{z}\|^2 + 2\left(m\langle \overline{z}, \overline{z}\rangle - m\langle \overline{z}, z\rangle - m\langle \overline{z}, \overline{z}\rangle + m\langle \overline{z}, z\rangle\right) \\
&= \sum_{k=1}^{m}\|z_k - \overline{z}\|^2 \qquad (2.4)
\end{aligned}$$

onde na igualdade da quinta linha foi utilizado o fato de que $\overline{z}$ é o centro de massa da nuvem de pontos, dado pela equação (2.2).

Naturalmente, o passo 4 também não aumenta o valor de S, visto que um ponto só muda o seu centroide associado caso o seu novo centroide esteja mais perto que o anterior.

Assim, a iteração dos passos 3 e 4 produz uma sequência de clusterizações, de modo que a sequência dos correspondentes valores de S é não crescente. Como o valor de S é sempre positivo, concluímos que essa sequência numérica converge quando o número de iterações tende a infinito.

Por fim, observamos que o número de possíveis configurações é finito, pois são N^n possíveis clusterizações, e a posição dos centroides a partir do passo 3 é uma função da clusterização. Assim, o conjunto dos possíveis valores para S ao longo do algoritmo é finito. Portanto, sendo convergente, a sequência dos valores de S ao longo do algoritmo é constante a partir de um certo termo. Isso e a Observação 2.2 garantem que o algoritmo converge em um número finito de passos. □

Apesar do Teorema 2.3 garantir que o algoritmo converge em um número finito de passos, ele não necessariamente converge para um mínimo global. A clusterização limite do algoritmo depende da posição inicial dos centroides

$c_1, \ldots, c_N$. É por esse motivo que aplicamos o algoritmo diversas vezes, com posições iniciais aleatórias, e depois escolhemos a clusterização com menor valor de S.

Na Figura 2.1, vemos a evolução do algoritmo com um conjunto aleatório de pontos dentro de $\mathbb{R}^2$. Vemos o seguinte comportamento: (a) Centroides iniciais posicionados aleatoriamente e associação de cada ponto ao centroide mais próximo. (b) Reposicionamento de cada centroide para o centro de massa da respectiva nuvem. (c) Rearranjamento dos pontos. (d) Reposicionamento dos centroides. (e) Rearranjamento dos pontos. (f) Reposicionamento dos centroides. (g) Rearranjamento dos pontos. (h) Reposicionamento dos centroides. Após esse passo a solução converge, ou seja, nenhum ponto muda de nuvem, encerrando o processo.

Ao aplicar o algoritmo k-means em uma imagem, associamos cada pixel a uma entre as N principais cores da mesma. Na Figura 2.2, vemos a aplicação deste processo em uma imagem, utilizando $N = 2, 3, 4, 5, 10, 20, 50$ e 100. Podemos ver que a resolução da imagem melhora conforme aumentamos o valor de N, entretanto o custo computacional também aumenta.

Também é importante ressaltar que, ao realizar a clusterização de uma imagem, perdemos informações e detalhes da imagem. Em vista disso, ao final do processo de transferência de cores realizamos um pós processamento, a fim de recuperarmos alguns desses detalhes (ver seção 2.5).

2.2 Transporte Ótimo Discreto

Ao abordar a transferência de cores entre duas imagens, trabalhamos com um número finito de pixels, ou seja, estamos trabalhando em um espaço discreto. Sendo assim, precisamos realizar algumas adaptações no problema de transporte visto anteriormente. Consideremos X e Y duas listas de pontos com o mesmo número N de elementos[2]:

$$X = (x_1, x_2, \ldots, x_N), \quad Y = (y_1, y_2, \ldots, y_N), \tag{2.5}$$

[2]No nosso problema específico de transferência de cores, as listas X e Y são formadas, respectivamente, pelas N cores das imagens base e modelo depois do processo de clusterização

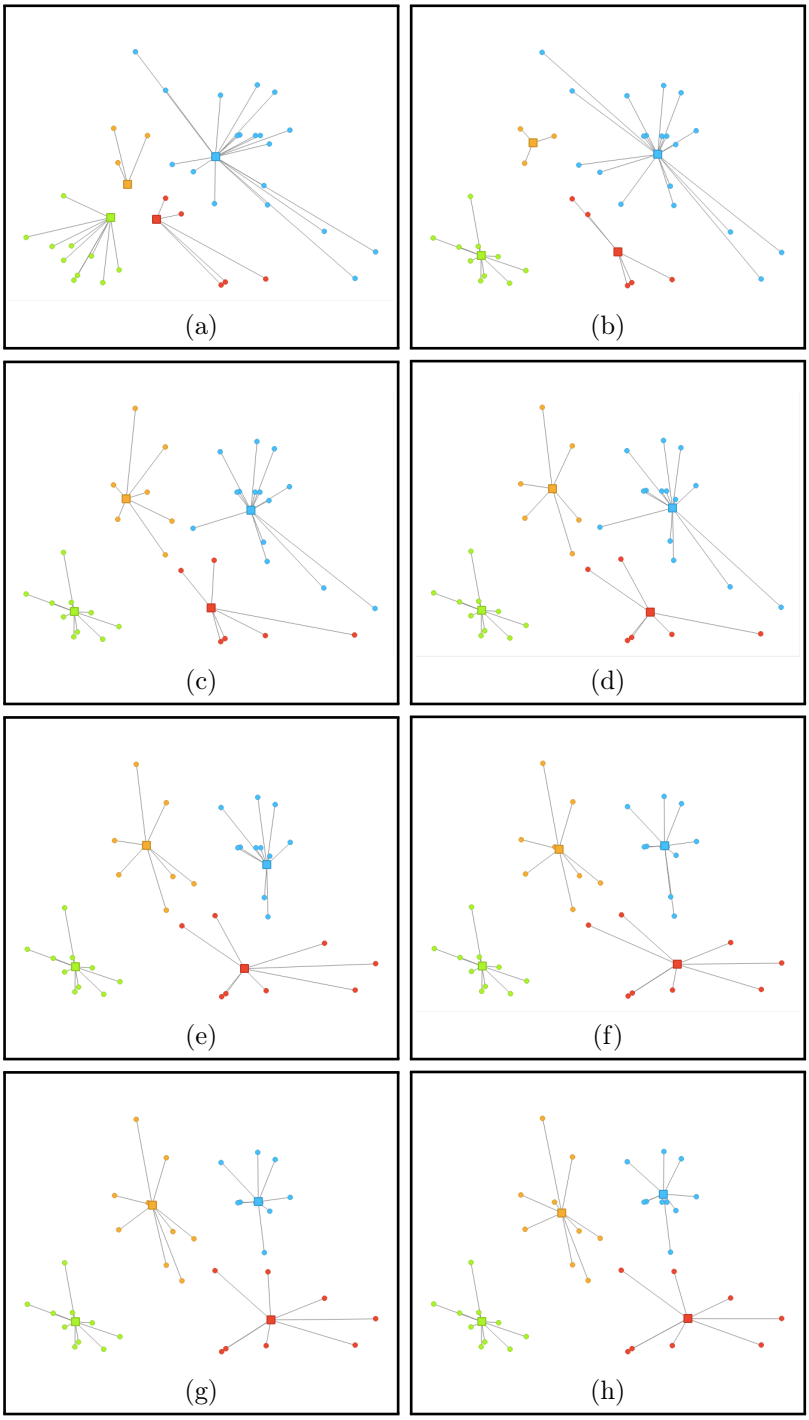


Figura 2.1: Evolução do algoritmo k-means em um conjunto de pontos aleatórios.

(a) Original (b) $N = 2$ (c) $N = 3$

(d) $N = 4$ (e) $N = 5$ (f) $N = 10$

(g) $N = 20$ (h) $N = 50$ (i) $N = 100$

Figura 2.2: Aplicação do algoritmo de clusterização em uma imagem. Comparamos a imagem original com as suas clusterizações usando $N = 2, 3, 4, 5, 10, 20, 50$ e 100.

onde $x_i, y_j \in \{0, 1, 2, \ldots, 255\}^3$. Para efeitos práticos, consideramos X e Y como vetores coluna com N componentes:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \quad \text{e} \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2.6}$$

Assumimos ainda as medidas associadas ao problema de transporte dão a mesma quantidade de massa aos respectivos pontos, ou seja,

$$\mu_X = \frac{1}{N}\sum_{i=1}^{N}\delta_{x_i}, \quad \mu_Y = \frac{1}{N}\sum_{i=1}^{N}\delta_{y_i}, \tag{2.7}$$

onde δ_{x_i} é a medida de Dirac no ponto $x_i \in \mathbb{R}^3$.

Neste contexto, o transporte T entre X e Y, na formulação original de Monge, é uma função bijetiva dada por $T(x_i) = y_{\sigma(i)}$, onde σ é uma permutação conveniente do conjunto $\{1, 2, \ldots, N\}$. Também podemos descrever σ como sendo uma matriz $\pi \in \mathbb{R}^{N\times N}$, de forma que

$$\pi_{i,j} = \begin{cases} 1, \text{ se } j = \sigma(i), \\ 0, \text{ se } j \neq \sigma(i). \end{cases} \tag{2.8}$$

Com essa notação matricial, podemos escrever T como

$$T(x_i) = (\pi Y)_i, \text{ para } i = \{1, 2, \ldots, N\}. \tag{2.9}$$

Também podemos escrever a função custo como uma matriz $C_{X,Y} \in (\mathbb{R}^+)^{N\times N}$ tal que

$$(C_{X,Y})_{i,j} = c(x_i, y_j), \quad \text{para} \quad i, j \in \{1, 2, \ldots, N\}. \tag{2.10}$$

No nosso problema de transferência de cores, utilizamos como função custo a distância euclidiana das cores no espaço $\mathbb{R}^3$, ou seja, $c(x_i, y_j) = \|x_i - y_j\|_2$.

Assim, a versão discreta do nosso problema de otimização (1.5) se torna

$$\min_{\pi\in\mathcal{P}} \langle C_{X,Y}, \pi\rangle, \tag{2.11}$$

onde $\mathcal{P} = \{\pi \in \mathbb{R}^{N\times N} : \pi^*\mathbb{I} = \mathbb{I}, \pi\mathbb{I} = \mathbb{I}, \pi_{i,j} \in \{0, 1\}\}$. Na nossa notação, temos que $\mathbb{I}$ é o vetor coluna $(1, \ldots, 1)^* \in \mathbb{R}^N$, A^* significa a matriz adjunta

de A (como estamos trabalhando apenas com números reais, esta é a matriz transposta) e

$$\langle C_{X,Y}, \pi \rangle := \sum_{i,j=1}^{N} c(x_i, y_j)\pi_{i,j}.$$

Podemos interpretar essas características de $\mathcal{P}$ da seguinte maneira:

- $\pi^*\mathbb{I} = \mathbb{I}$ significa que a soma dos elementos de cada coluna de π sempre vale 1. Como os elementos de π só podem ser 0 ou 1, temos que apenas um elemento de cada coluna vale 1 e os outros todos valem 0. Dessa forma, cada x_i envia toda a sua massa para um único y_j.

- $\pi\mathbb{I} = \mathbb{I}$ significa que a soma dos elementos de cada linha sempre vale 1. Assim como no item anterior, temos que apenas um elemento vale 1 e os outros valem 0. Dessa forma, cada y_j recebe a sua massa total de um único x_i.

Ao adaptar o problema para a versão de Kantorovich, ocorrem algumas mudanças nas restrições: agora os elementos de π podem ser qualquer valor real entre 0 e 1, não apenas os extremos. Sendo assim, nosso problema se torna

$$\min_{\pi \in \mathcal{S}_1} \langle C_{X,Y}, \pi \rangle, \tag{2.12}$$

onde $\mathcal{S}_1 = \{\pi \in \mathbb{R}^{N \times N} : \pi^*\mathbb{I} = \mathbb{I}, \pi\mathbb{I} = \mathbb{I}, \pi_{i,j} \in [0,1]\}$. Sobre as características de $\mathcal{S}_1$, temos o seguinte:

- Como agora cada componente de π pode ser um valor real entre 0 e 1, a restrição $\pi\mathbb{I} = \mathbb{I}$ implica que a soma dos elementos de cada linha seja exatamente 1. Entretanto, temos que uma componente $\pi_{i,j}$ de π indica qual é a quantidade de massa que y_j irá receber de x_i. Assim, a soma total de uma linha i indica qual é o total de massa transportada de x_i. Como esse valor vale 1 para qualquer linha, temos que toda a massa de x_i deve ser transportada, mesmo que dividida entre os elementos de Y.

- Da mesma maneira, temos que a restrição $\pi^*\mathbb{I}$ indica que a soma de cada coluna j de π sempre deve valer 1, ou seja, cada y_j sempre deve receber toda a massa possível, vinda de diversos elementos de X.

Temos que o problema de Kantorovich discreto (2.12) é computacionalmente mais simples de resolver do que o de Monje (2.11), visto que métodos

mais simples resolvem (2.12) (como por exemplo o Método Simplex), enquanto precisamos de métodos de resolução mais complexos para resolver (2.11).

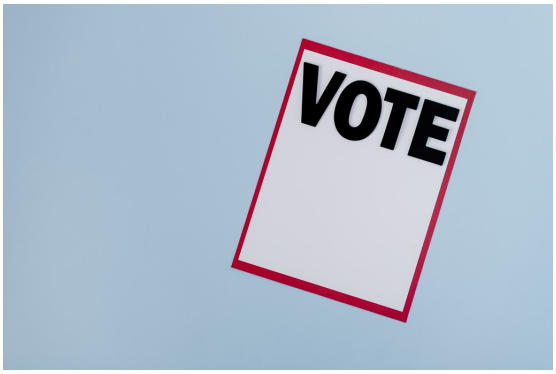
2.3 Relaxamento do Transporte

É possível realizar a transferência de cores resolvendo o problema (2.12), entretanto nesse contexto relaxar a condição de conservação de massa é crucial para obtermos melhor associação entre as cores mais representativas em cada imagem. Dada uma imagem, podemos separá-la em diversos pedaços com cores muito semelhantes em cada pedaço (por exemplo o fundo de uma imagem), cada um destes pedaços terá uma respectiva massa, dada pelo número de centroides localizados nessa região. Em geral, queremos que a solução do problema de transferência de cores mantenha essa distribuição, ou seja, que pixels reunidos em um mesmo pedaço na imagem inicial continuem reunidos em um mesmo pedaço na imagem final. Entretanto, caso as imagens utilizadas não tenham o mesmo número de pedaços ou caso os seus pedaços não tenham massas parecidas, nossa solução irá misturar os pedaços, gerando uma imagem não tão satisfatória. Um exemplo disso pode ser visto na Figura 2.3a: O retângulo preto na imagem base possui um certo número de centroides associados a ele e, como as proporções de cores das duas imagens são diferentes, não há na imagem modelo um conjunto de centroides com cores semelhantes que possam ser relacionados aos centroides do retângulo preto. Assim, ao realizar a transformação, precisamos utilizar mais centroides da imagem modelo (e com cores diferentes) para preencher o mesmo espaço antes ocupado por uma única cor (de fato, vemos uma mistura de preto e vermelho onde antes havia apenas preto).

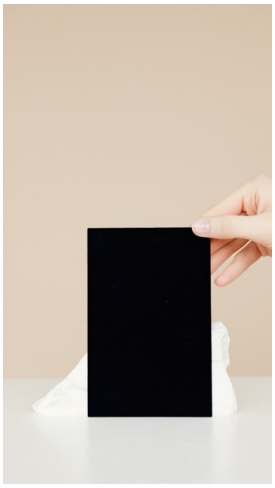
Para melhorar o nosso resultado, podemos realizar um relaxamento no nosso problema de transporte: substituímos as restrições de que cada linha e que cada coluna tenham soma total igual a 1 e permitimos que estejam em uma faixa de valores. Mais especificamente, definimos quatro parâmetros $k_X, K_X, k_Y, K_Y \in \mathbb{R}^+$ e substituimos as restrições $\pi\mathbb{I} = \mathbb{I}$ e $\pi^*\mathbb{I} = \mathbb{I}$ por

$$k_X\mathbb{I} \leq \pi\mathbb{I} \leq K_X\mathbb{I} \quad \text{e} \quad k_Y\mathbb{I} \leq \pi^*\mathbb{I} \leq K_Y\mathbb{I}. \tag{2.13}$$

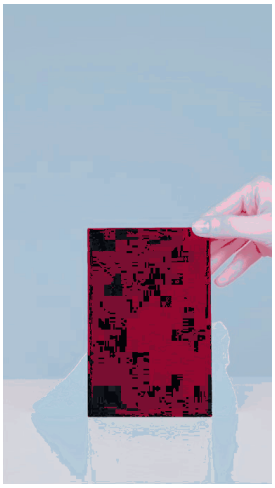
O relaxamento destas restrições faz com que a massa transportada entre os pontos não seja conservada completamente, ou seja, cada x_i pode trans-



(a) Imagem Modelo.



(b) Imagem Base.



(c) Resultado.

Figura 2.3: Sem o relaxamento na condição de conservação de massa, podemos ter um resultado insatisfatório.

portar mais ou menos massa do que o seu total, assim como cada y_j pode receber mais ou menos massa do que o seu total.

Apesar desse relaxamento na conservação de massa pontualmente, ainda queremos exigir a conservação de massa total, ou seja, exigimos que toda a massa de X seja transportada para Y. Fazemos esta restrição exigindo que $\mathbb{I}^*\pi\mathbb{I} = N$, ou seja, a soma de todas as componentes de π deve valer exatamente N. Além disso, como $\pi \in [0,1]^{N\times N}$, temos que qualquer conexão entre dois pontos x_i e y_j deve valer no máximo 1.

Com isso, o nosso problema se torna encontrar

$$\min_{\pi\in\mathcal{S}_k} \langle C_{X,Y}, \pi\rangle, \tag{2.14}$$

onde

$$\mathcal{S}_k = \left\{\pi \in [0,1]^{N\times N} \,|\, k_X\mathbb{I} \leq \pi\mathbb{I} \leq K_X\mathbb{I},\ k_Y\mathbb{I} \leq \pi^*\mathbb{I} \leq K_Y\mathbb{I},\ \mathbb{I}^*\pi\mathbb{I} = N\right\}.$$

Observação 2.4. O problema (2.14) tem solução se, e somente se, $k_X \leq 1 \leq K_X$ e $k_Y \leq 1 \leq K_Y$. A ideia para provar a ida é supor que alguma das desigualdades não vale e chegar em um absurdo. Começamos supondo que $k_X > 1$, então $\mathbb{I}^*\pi\mathbb{I} \geq \mathbb{I}^*k_X\mathbb{I} > \mathbb{I}^*\mathbb{I} = N$, absurdo. Supondo que $K_X < 1$, temos que $\mathbb{I}^*\pi\mathbb{I} \leq \mathbb{I}^*K_X\mathbb{I} < \mathbb{I}^*\mathbb{I} = N$, absurdo. Temos que $\mathbb{I}^*\pi\mathbb{I} = \mathbb{I}^*\pi^*\mathbb{I}$, pois ambos representam a soma de todos os elementos de π. Vamos supor então que $k_Y > 1$, então $\mathbb{I}^*\pi\mathbb{I} = \mathbb{I}^*\pi^*\mathbb{I} \geq \mathbb{I}^*k_Y\mathbb{I} > \mathbb{I}^*\mathbb{I} = N$, absurdo. Por fim, supondo que $K_Y < 1$, temos que $\mathbb{I}^*\pi\mathbb{I} = \mathbb{I}^*\pi^*\mathbb{I} \leq \mathbb{I}^*K_Y\mathbb{I} < \mathbb{I}^*\mathbb{I} = N$, absurdo.

Agora passamos a provar a volta. Temos que o problema (2.14) já possui solução quando $(k_X, K_X, k_Y, K_Y) = (1,1,1,1)$, pois se trata de um problema clássico de transporte de Kantorovich. Como estamos relaxando algumas restrições, o problema continua tendo solução. □

Ao relaxar o transporte e não exigir a conservação de massa pontualmente, temos que uma solução π do problema (2.14) pode não definir uma aplicação bijetiva. Mais ainda, um mesmo ponto x_i pode ser levado em mais do que um y_j. Para contornar isso podemos, a partir de π, definir uma função $T_\pi : X \to \mathbb{R}^3$ tal que

$$T_\pi(x_i) = \frac{\sum_{j=1}^N \pi_{i,j} y_j}{\sum_{j=1}^N \pi_{i,j}}. \tag{2.15}$$

O papel que T_π faz na prática é levar cada ponto x_i no baricentro ponderado dos pontos y_j com pesos $\pi_{i,j}$, $j = 1, \ldots, N$. Podemos representar T_π

em notação matricial como $T_\pi(x_i) = (Z_\pi)_i$, onde $Z_\pi = (\text{diag}(\pi\mathbb{I}))^{-1}\pi Y$, $(Z_\pi)_i$ representa a i-ésima linha de Z_π e $\text{diag}(v)$ significa uma matriz diagonal em $\mathbb{R}^{N\times N}$ tal que a sua diagonal é o vetor $v \in \mathbb{R}^N$. Também podemos definir uma função $T'_\pi : Y \to \mathbb{R}^3$ dada por

$$T'_\pi(y_j) = \frac{\sum_{i=1}^{N} \pi^*_{i,j} x_i}{\sum_{i=1}^{N} \pi^*_{i,j}}, \tag{2.16}$$

que na sua versão matricial pode ser escrita como $T'_\pi(y_j) = (Z'_\pi)_j$, onde $Z'_\pi = (\text{diag}(\pi^*\mathbb{I}))^{-1}\pi^* X$. Pelas construções de T_π e de T'_π, temos que necessariamente k_X e k_Y serão valores não nulos, pois não podemos ter uma linha ou coluna de π com soma zero.

Em alguns momentos, é mais fácil trabalhar com a forma matricial de T_π e de T'_π, em outros momentos trabalhamos com a versão original.

Temos que esse relaxamento se mostra necessário no nosso contexto de transferência de cores, melhorando o resultado em diversas situações. Na Figura 2.4, temos o resultado dessa regularização: quando não realizamos o relaxamento, o fundo da imagem é dividido em diversas cores que não se misturam muito bem, sendo que na imagem base ele é relativamente uniforme. Ao realizar o relaxamento utilizando $(k_X, K_X, k_Y, K_Y) = (0.9, 1.1, 0.9, 1.1)$, podemos ver uma suavização deste efeito.

2.4 Regularização do Transporte

Após realizarmos o relaxamento do problema de transporte e da conservação de massa, podemos realizar uma regularização no transporte, a fim de diminuir artefatos na imagem resultante. Entretanto, problemas de regularização geralmente se baseiam na variação dos parâmetros em uma vizinhança de cada ponto (o que não é o nosso caso, pois estamos trabalhando com nuvens discretas de pontos).

Para contornar este problema, construímos a vizinhança de cada ponto em uma nuvem de pontos utilizando uma estrutura de grafos direcionados e aplicamos o operador gradiente neste grafo. Assim, a vizinhança de um ponto é formada pelos pontos ligados a ele pelo grafo. Isso é feito da seguinte forma: dada uma nuvem de pontos X, definimos um grafo $\mathcal{G}_X = (X, E_X, W_X)$, onde $E_X \subseteq \{1, 2, \ldots, N\}^2$ indica o conjunto de arestas do grafo (se $(i, j) \in E_X$, então x_i se conecta a x_j) e $W_X = (w_{i,j})_{i,j=1}^N : \{1, 2, \ldots, N\}^2 \to \mathbb{R}^+$ indica

(a) Imagem base

(b) Resultado sem relaxamento

(c) Imagem modelo

(d) Resultado com relaxamento, utilizando $(k_X, K_X, k_Y, K_Y) = (0.9, 1.1, 0.9, 1.1)$

Figura 2.4: Resultado do relaxamento nas condições de transporte.

o conjuntos dos pesos dados a cada aresta. Na nossa construção, definimos que $(w_{i,j}) = 0 \iff (i,j) \notin E_X$.

Para definir as arestas de $\mathcal{G}_X$, definimos um parâmetro $k \in \mathbb{N}$ e utilizamos o critério de que x_i se conecta a x_j se e somente se x_j é um dos k pontos mais próximos de x_i. Temos, quanto maior for o valor de k, mais conexões cada um dos pontos realiza, aumentando assim a regularização. Ao mesmo tempo, ao aumentarmos o valor de k, também aumentamos o custo computacional, pois ficamos com um número maior de variáveis e de restrições no nosso problema de programação linear, a ser visto em (2.23).

Na Figura 2.5, vemos o papel de k na regularização do problema. Em todas as imagens utilizamos os parâmetros $N = 200, (k_X, K_X, k_Y, K_Y) = (1,1,1,1)$ e $\lambda_X = \lambda_Y = 0.05$[3], variando apenas o valor de k. Na imagem (a) utilizamos $k = 1$, vemos que foram criados diversos artefatos na imagem, como por exemplo a aparição de um bloco verde no meio do céu. Na imagem (b) foi utilizado o valor de $k = 2$ e já conseguimos ver uma melhora no resultado visual da imagem: a mancha verde sumiu, mas ainda vemos algumas diferenças de cores no céu da imagem. Na imagem (c) utilizamos o valor de $k = 4$ e podemos observar que não há mudanças bruscas de cores no céu da imagem e já temos um resultado visual satisfatório. Na imagem (d), utilizamos o valor de $k = 10$ e não observamos grandes melhoras visuais em relação ao resultado obtido com $k = 4$, inclusive verificamos um resultado levemente inferior na resolução do casco do barco à direita da imagem.

Para a escolha dos pesos de $\mathcal{G}_X$, utilizamos o inverso da distância euclidiana

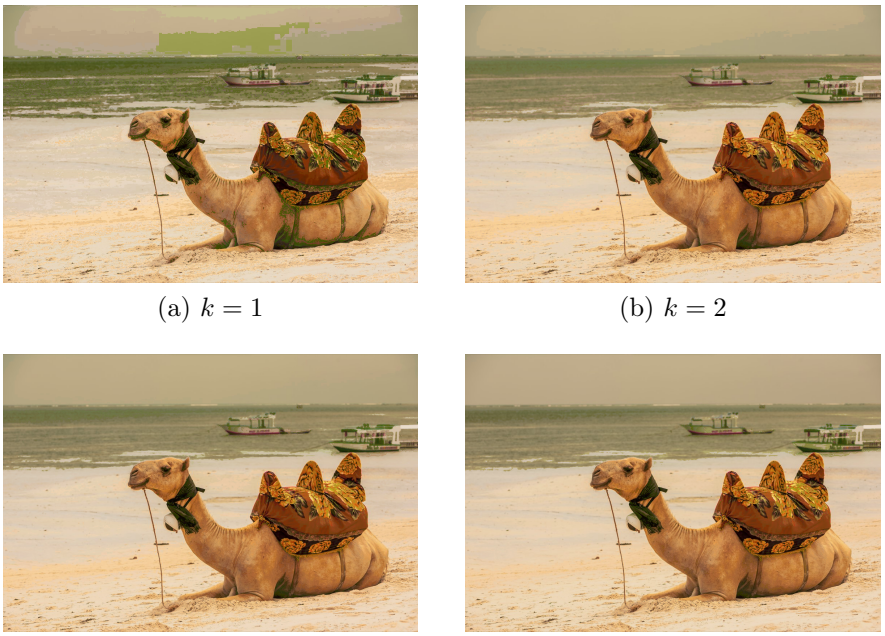
$$w_{i,j} = \|x_i - x_j\|^{-1} = \frac{1}{\sqrt{(x_i^1 - x_j^1)^2 + (x_i^2 - x_j^2)^2 + (x_i^3 - x_j^3)^2}}, \tag{2.17}$$

onde x_i^1 representa a primeira coordenada de x_i, x_i^2 representa a segunda, e x_i^3 a terceira. Essa escolha para os pesos valoriza aquelas arestas que ligam pontos mais próximos.

Então, definimos o operador gradiente em $\mathcal{G}_X$, que será dado por $G_X : \mathbb{R}^{N\times 3} \to \mathbb{R}^{P\times 3}$, onde $P = \|E_X\|$ é o número de arestas de $\mathcal{G}_X$. Temos que G_X é tal que, dada uma nuvem de pontos $V = (V_i)_{i=1}^N \in \mathbb{R}^{N\times 3}$,

$$G_X(V) = (w_{i,j}(V_i - V_j))_{(i,j)\in E_X} \in \mathbb{R}^{P\times 3}. \tag{2.18}$$

[3]Temos que λ_X e λ_Y são parâmetros, a serem definidos, que controlam a regularização do nosso resultado.



(a) $k = 1$ (b) $k = 2$

(c) $k = 4$ (d) $k = 10$

Figura 2.5: Papel do parâmetro k na regularização do resultado final.

Mais detalhes podem ser vistos em [5].

Para medir a regularidade da nuvem de pontos V, definimos uma norma em $G_X(V)$, dada por

$$J(G_X(V)) = \sum_{(i,j)\in E_X} \|w_{i,j}(V_i - V_j)\|_1, \tag{2.19}$$

onde, para $v = (v_1, v_2, v_3) \in \mathbb{R}^3$, temos que $\|v\|_1 = |v_1| + |v_2| + |v_3|$.

Na prática, J calcula a soma dos módulos de todos os vetores computados por G_X. Assim, temos que J é um indicador do quão irregular é a nossa nuvem de pontos, ou seja, quanto maior o valor de $J(G_X(V))$, mais irregular é V.

No nosso contexto, nós queremos regularizar a nuvem de pontos $\hat{V}(\pi) = X - T_\pi(X)$, dada pela diferença entre a cor original de cada pixel e a sua cor após T_π ser aplicada. Na forma matricial, $\hat{V}$ é dado por $X - (\text{diag}(\pi\mathbb{I}))^{-1}\pi Y$. Entretanto, como temos um termo $(\text{diag}(\pi\mathbb{I}))^{-1}$, este mapa não é convexo, gerando problemas na otimização. Para contornar isso, multiplicamos os dois termos por $\text{diag}(\pi\mathbb{I})$ e realizamos a regularização da nuvem de pontos $V(\pi) = \text{diag}(\pi\mathbb{I})X - \pi Y$.

Ao realizar essa multiplicação por $\text{diag}(\pi\mathbb{I})$, temos que a nossa regularização passa a dar mais importância aos pontos que estão transferindo mais massa, ou seja, cujas linhas de π tem maiores valores.

Da mesma forma, também queremos realizar a regularização em $Y - T'_\pi(Y)$, portanto também devemos regularizar a nuvem de pontos $V'(\pi) = \text{diag}(\pi^*\mathbb{I})Y - \pi^* X$.

Para simplificar a notação, definimos as seguintes funções

$$\Delta_{X,Y}(\pi) = \text{diag}(\pi\mathbb{I})X - \pi Y \tag{2.20}$$

e

$$\Delta_{Y,X}(\pi) = \text{diag}(\pi^*\mathbb{I})Y - \pi^* X. \tag{2.21}$$

Assim, para realizar a formulação do problema de transporte entre X e Y, construímos os grafos $\mathcal{G}_X$ e $\mathcal{G}_Y$, calculamos os gradientes[4] $G_X \in \mathbb{R}^{P_X \times 3}$ e $G_Y \in \mathbb{R}^{P_Y \times 3}$ e resolvemos um problema de otimização da forma

$$\min_{\pi\in\mathcal{S}_k}\Big(\langle \pi, C_{X,Y}\rangle + \lambda_X J(G_X(\Delta_{X,Y}(\pi)) + \lambda_Y J(G_Y(\Delta_{Y,X}(\pi))\Big), \tag{2.22}$$

[4]Temos que P_X e P_Y são, respectivamente, o número de arestas de $\mathcal{G}_X$ e de $\mathcal{G}_Y$.

(a) $\lambda_X = \lambda_Y = 0$ (b) $\lambda_X = \lambda_Y = 0.001$

(c) $\lambda_X = \lambda_Y = 0.2$ (d) $\lambda_X = \lambda_Y = 2$

Figura 2.6: Papel desempenhado por λ_X e λ_Y no resultado final.

onde $(\lambda_X, \lambda_Y) \in (\mathbb{R}^+)^2$ são parâmetros que controlam a regularidade desejada nas nuvens de pontos X e Y, respectivamente. Esse termos adicionais colocados no problema de minimização podem ser vistos como penalizações proporcionais às irregularidades de cada nuvem. Os parâmetros λ_X e λ_Y são então os pesos dessas penalizações, portanto conseguimos controlar a regularidade desejada através deles.

Podemos observar que, quando $(\lambda_X, \lambda_Y) = (0, 0)$, o problema não exige nenhuma regularização. Ainda, se $(\lambda_X, \lambda_Y) = (0, 0)$ e $k = (1, 1, 1, 1)$, então temos um problema padrão de transporte ótimo (Kantorovich), sem relaxamento e sem regularização.

Na Figura 2.6, vemos o papel desempenhado por λ_X e λ_Y na regularização dos resultados. Temos que quanto maior o valor dos parâmetros, mais suave é o nosso resultado. Ao mesmo tempo, utilizar um valor muito grande para λ_X e λ_Y também faz com que a distribuição de cores não seja tão parecida com a da imagem modelo (podemos ver na imagem (d) que há uma presença muito pequena da cor verde, o que não está de acordo com a imagem modelo, vista na Figura 2.4c). Em todas as imagens utilizamos os parâmmetros $N = 200$, $k = 4$ e $(k_X, K_X, k_Y, K_Y) = (1, 1, 1, 1)$.

Para realizar a transferência de cores computacionalmente, queremos visualizar o problema (2.22) como um problema de programação linear. Entretanto, temos que dentro de J foi utilizada a função módulo, o que não é aceito em problemas de programação linear.

Precisamos então adaptar o nosso problema utilizando duas variáveis auxiliares: $U_X \in (\mathbb{R}^+)^{P_X \times 3}$ e $U_Y \in (\mathbb{R}^+)^{P_Y \times 3}$. Com essas duas variáveis, podemos moldar o nosso problema da seguinte forma:

$$\begin{aligned} &\min_{\pi, U_X, U_Y} \left(\langle C_{X,Y}, \pi \rangle + \lambda_X \langle U_X, \mathbb{I} \rangle + \lambda_Y \langle U_Y, \mathbb{I} \rangle \right) \\ &\text{sujeito a} \quad \begin{cases} -U_X \leq G_X(\Delta_{X,Y}(\pi)) \leq U_X \\ -U_Y \leq G_Y(\Delta_{Y,X}(\pi)) \leq U_Y, \\ \pi \in \mathcal{S}_k. \end{cases} \end{aligned} \tag{2.23}$$

Os problemas (2.22) e (2.23) são de fato equivalentes, pois U_X e U_Y devem assumir os menores valores positivos possíveis, assumindo assim valores iguais aos módulos dos valores de $G_X(\Delta_{X,Y}(\pi))$ e $G_Y(\Delta_{Y,X}(\pi))$, respectivamente. A vantagem de visualizarmos o problema desta maneira é que conseguimos resolvê-lo através de métodos usuais de programação linear, como o Método Simplex.

É importante observar que o problema (2.23) possui uma região admissível R não-vazia. De fato, se tomarmos $\pi'_{i,j} = 1/N$, para todo i, j, temos que $\pi' \in \mathcal{S}_k$ e que os elementos da matriz $G_X(\pi' Y - \text{diag}(\pi' \mathbb{I}) X)$ têm um valor absoluto máximo (que podemos chamar de M_X), assim como os elementos da matriz $G_Y(\pi'^* X - \text{diag}(\pi'^* \mathbb{I}) Y)$ também têm um valor absoluto máximo (que chamamos M_Y). Temos assim que π', U'_X, U'_Y com

$$\begin{aligned} \pi'_{i,j} &= \frac{1}{N}, \\ U'_{Xi,j} &= M_X, \\ U'_{Yi,j} &= M_Y, \end{aligned} \tag{2.24}$$

é um elemento (não-minimizante, a princípio) que satisfaz as restrições descritas em (2.23).

Ainda, as variáveis pertencentes a π estão todas limitadas no intervalo $[0, 1]$, enquanto as variáveis pertencentes a U_X e U_Y estão limitadas inferiormente por 0 e possuem coeficientes positivos. Sendo assim, mesmo que R não seja necessariamente limitada, ela possui uma solução minimizante.

2.5 Interpolação

Após resolver o problema (2.23), nos resta apenas realizar um pós processamento na nossa imagem final, a fim de retomar alguns detalhes que perdemos ao fazer a clusterização das imagens base e modelo. Realizamos então uma interpolação, através da função $T^0 : \mathbb{R}^3 \to \mathbb{R}^3$ dada por

$$T^0(x) = T((x_{i(x)}) + x - x_{i(x)}, \tag{2.25}$$

onde T é definida como em (2.15) e $i(x) = \underset{1 \leq i \leq N}{\arg\min} \|x - x_i\|$.

Essa função T^0 adiciona a T o termo $x - x_{i(x)}$, que é justamente a diferença entre a cor original do pixel e a sua cor após a clusterização. Na Figura 2.7, vemos o resultado dessa aplicação com diferentes valores para o parâmetro N. Temos que com um N pequeno há uma grande melhora no resultado final. Quando utilizamos um valor maior para N, há apenas uma pequena melhora na resolução nos cascos dos barcos e no banco do camelo, o restante da imagem permanece praticamente igual. Isso se deve justamente ao fato de que quanto maior o número de cores utilizadas, menor será a distância $x - x_{i(x)}$, diminuindo o efeito visual do pós-processamento.

(a) Resultado sem a interpolação com $N = 15$.

(b) Resultado com a interpolação com $N = 15$.

(c) Resultado sem a interpolação com $N = 200$.

(d) Resultado com a interpolação com $N = 200$.

Figura 2.7: Papel do pós-processamento no resultado final da transferência de cores.

Capítulo 3

Aplicação do Método em Python

Neste capítulo, mostramos em detalhes a construção do código para realizar a transferência de cores entre duas imagens. Todo o código foi construído em Python, uma linguagem de programação orientada a objeto e de código aberto. Detalhes sobre as principais funções e funcionalidades dessa linguagem podem ser encontrados em [7]. Na Seção 3.1, criamos uma função que transforma qualquer imagem em uma lista, indicando a cor de cada pixel da imagem. Na Seção 3.2, realizamos a construção de duas funções que, juntas, realizam a clusterização de uma imagem em N cores diferentes. Na Seção 3.3, construímos as funções $\Delta_{X,Y}$ e $\Delta_{Y,X}$, definidas na Seção 2.4. Na Seção 3.4, criamos a função que cria um grafo a partir de uma nuvem de pontos. Na Seção 3.5, criamos a função gradiente G_X, definida na Seção 2.4. Na Seção 3.6, criamos a função que resolve o problema de programação linear definido em (2.23). Na Seção 3.7, construímos a função que gera a imagem final após a resolução do programa linear. Na Seção 3.8, construímos a função que junta todas as outras funções e realiza a transferência de cores entre as duas imagem.

Para conseguir utilizar as funções, primeiro necessitamos importar diversos pacotes, módulos e funções, são eles os seguintes:

```
from PIL import Image
import os
from sklearn.cluster import KMeans
from pulp import LpMaximize, LpProblem, LpStatus, lpSum,
LpVariable, LpMinimize
from math import sqrt
```

O módulo Image, dentro do pacote PIL, foi importado justamente para

que pudéssemos trabalhar com imagens no nosso código. Esse módulo possui algumas funções essenciais para o nosso trabalho, como por exemplo importar imagens externas, obter as cores de um determinado pixel da imagem, salvar uma imagem e obter as dimensões de altura e largura de uma imagem.

O módulo "os" nos ajuda a trabalhar com elementos relacionados ao sistema operacional, como por exemplo acessar as pastas onde armazenamos as nossas imagens.

A função Kmeans, dentro do módulo sklearn.cluster, serve para realizarmos o algoritmo k-means e clusterizar as nossas imagens.

As funções LpMaximize, LpProblem, LpStatus, lpSum, LpVariable e LpMinimize, localizadas dentro do módulo pulp, foram necessárias para conseguirmos aplicar os algoritmos de programação linear utilizados.

A função sqrt, dentro do módulo math, foi utilizada para realizarmos operações com raízes quadradas.

3.1 Decodificação de uma imagem

Uma maneira de trabalharmos com imagens no Python é através de listas. Podemos "traduzir" uma imagem com n pixels para uma lista $M = (x_1, x_2, \ldots, x_n)$ onde cada pixel $x_i \in \{0, 1, \ldots, 255\}^3$ da lista possui 3 informações, correspondentes às suas coordenadas na escala RGB[1].

Para fins de organização, ordenamos os elementos da lista conforme a ordem de leitura dos pixels da imagem (da esquerda para a direita, de cima para baixo). Além disso, é preciso ressaltar que, dentro do Python, as posições dos pixels não seguem exatamente as orientações do plano cartesiano. Na verdade, a coordenada x cresce para a direita, mas a coordenada y cresce para baixo, como podemos ver na Figura 3.1. Assim, o pixel com posição $(0, 0)$[2] de uma imagem será sempre aquele localizado no canto superior esquerdo da imagem.

O código da função que traduz uma imagem para uma lista é o seguinte:

```
def decodificador(img):
    M = []
```

[1]Durante o restante deste capítulo, quando nos referirmos às coordenadas de um pixel, estamos nos referindo às coordenadas da cor deste pixel na escala RGB. Caso queiramos nos referir à posição do pixel dentro da imagem, identificamos isso explicitamente.

[2]Na linguagem Python a contagem sempre inicia no valor 0, assim uma lista com 3 elementos terá as posições $0, 1$ e 2.

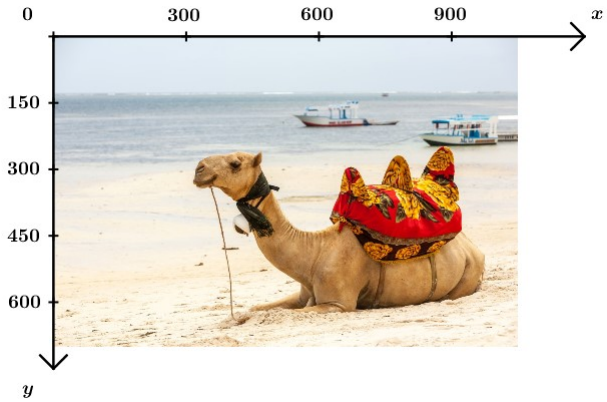


Figura 3.1: Posição dos pixels de uma imagem em Python.

```
L, A = img.size

for y in range(A):
    for x in range(L):
        pxl = img.getpixel((x,y))
        M.append([pxl[0],pxl[1],pxl[2]])

return [M, L]
```

Inicialmente, criamos uma lista vazia M onde serão armazenadas as coordenadas dos pixels, depois extraímos a largura L e a altura A da imagem. Finalmente, fazemos uma varredura pixel a pixel na imagem, identificando as suas coordenadas e as adicionando em M na ordem definida anteriormente.

Por fim, retornamos uma lista contendo a lista M juntamente com a largura L da imagem (a informação de L é importante pois posteriormente precisamos reconstruir a imagem, e sem a informação da largura não temos como definir as dimensões da imagem).

Através da decodificação das imagens, conseguimos trabalhar facilmente com os dados de cada pixel individualmente, pois a sua posição na imagem e na lista M estão relacionadas através de uma bijeção. De fato, se a posição de um pixel na imagem é dada por (x, y), então a sua posição na lista pode ser obtida pelo comando $M[L * y + x]$.

3.2 Clusterização de uma imagem

Após transformar uma imagem em uma lista, podemos agora realizar a clusterização da nossa imagem através da lista obtida com a função decodificador.

Para isso, criamos a função k_means, que a partir de uma imagem e de um número natural N, nos retorna uma lista com 3 informações: a lista M com as coordenadas dos pixels da imagem, a largura da imagem e uma lista com as cordenadas dos N centroides obtidos. O código da função é o seguinte:

```
def k_means(img, N):

    lista = decodificador(img)

    kmeans = KMeans(
        init = "random",
        n_clusters = N,
        n_init = 10,
        max_iter = 30)

    kmeans.fit(lista[0])

    C = kmeans.cluster_centers_
    for i in range(N):
        for j in range(3):
            C[i][j] = int(C[i][j])

    return(lista[0],lista[1],C)
```

Inicialmente aplicamos a função decodificador na nossa imagem para transformá-la em uma lista. Em seguida, utilizamos a função KMeans, que foi extraída do módulo "sklearn.cluster", essa função que aplica o algoritmo de clusterização no nosso conjunto de pixels, já vimos como funciona esse algoritmo na Seção 2.1. Temos os seguintes parâmetros para a função KMeans:

- init: neste parâmetro definimos como serão escolhidas as coordenadas iniciais dos centroides, escolhemos esse parâmetro como aleatório.
- n_clusters: aqui definimos o número de centroides (e consequentemente de nuvens) que queremos, colocamos N para esse parâmetro.

- n_init: esse parâmetro diz respeito a quantas vezes vamos repetir o processo. Por padrão, escolhemos 10 para este parâmetro.

- max_iter: aqui definimos o número máximo de iterações que cada processo irá realizar, colocamos 30 para esse parâmetro.

Após, realizamos o "encaixe" do KMeans em nossa lista para obter as coordenadas dos centroides, nomeamos a lista com as coordenadas RGB dos centroides de C. Como as coordenadas dos centroides podem ser dadas por valores decimais, precisamos realizar o arredondamento para valores inteiros[3]. Por fim, retornamos uma lista com 3 informações: a lista de pixels M, a largura da imagem e a lista de centroides C.

Após aplicar o algoritmo k-means, podemos recriar a nossa imagem mudando a cor de cada pixel para a do centroide associado, fazemos isso através da função criador. Essa função tem como entrada uma lista com três informações (M, L e C) e tem como saída a mesma lista, mas com cada pixel associado ao centroide respectivo, além da imagem clusterizada.

Abaixo vemos o código da função criador.

```
def criador(lista):

    n = len(lista[0])
    N = len(lista[2])
    L = lista[1]
    A = n//L
    C = lista[2]
    M = lista[0]

    img = Image.new("RGB", (L,A), (0,0,0))

    for y in range(A):
        for x in range(L):
            pxl = M[L*y + x]
            mi = 0
            d0 = -1

            for j in range(N):
                d1 = (pxl[0] - C[j][0])**2
                + (pxl[1] - C[j][1])**2
```

[3]Na prática, o que fazemos é apenas remover a parte decimal de cada coordenada, não necessariamente arredondando o valor. Isso não causa grandes alterações, pois as coordenadas variam de 0 a 255.

```
                + (pxl[2] - C[j][2])**2

                if d0 == -1:
                    d0 = d1
                if d1 < d0:
                    d0 = d1
                    mi = j

            M[L*y + x].append(mi)

            img.putpixel((x,y),(int(C[mi][0]),
            int(C[mi][1]),int(C[mi][2])))

    img.show()

    return([M,L,C],img)
```

Inicialmente, obtemos os valores n (número de pixels), N (número de centroides), L (largura da imagem), A (altura da imagem), C (lista de centroides) e M (lista de pixels). Em seguida, criamos uma imagem totalmente preta com as dimensões da imagem desejada.

Iniciamos então o processo de descobrir a qual centroide cada pixel está associado[4]. Para isso, dado um pixel $pxl_{x,y}$, calculamos a distância euclidiana entre as suas coordenadas RGB e as de cada centroide C_j e escolhemos aquele com a menor distância encontrada, ou seja, procuramos

$$\arg\min_{0\leq j<N} (pxl_{x,y}[0] - C_j[0])^2 + (pxl_{x,y}[1] - C_j[1])^2 + (pxl_{x,y}[2] - C_j[2])^2. \quad (3.1)$$

Para encontrar o centroide com a menor distância de cada pixel, criamos as notações mi, d_0 e d_1: dado um pixel $pxl_{x,y}$, d_{1_j} indica o quadrado da distância de $pxl_{x,y}$ a C_j, d_{0_j} indica a menor distância encontrada ao testarmos de C_0 a C_j e mi_j indica o índice do centroide mais próximo de $pxl_{x,y}$ ao testarmos de C_0 até C_j. Ao fim do processo, temos que mi nos dá a posição do centroide mais próximo de $pxl_{x,y}$ e d_0 nos dá a distância do pixel até este centroide:

$$\begin{aligned} d_{1_j} &= \|pxl_{x,y} - C_j\|_2^2 \\ d_{0_j} &= \min_{0\leq k\leq j} d_{1_k} \\ mi_j &= \arg\min_{0\leq k\leq j} d_{1_k}. \end{aligned} \quad (3.2)$$

[4]Esse processo pode otimizado utilizando algoritmos mais sofisticados, como por exemplo [6].

Após descobrir o centroide mais próximo de cada pixel, adicionamos essa informação na lista de pixels M, fazendo com que cada ponto tenha 4 informações: suas 3 coordenadas e o índice do centroide associado.

A seguir, trocamos a cor do pixel correspondente à posição (x, y) na imagem em preto pela cor do centroide associado a $pxl_{x,y}$. Por fim, exibimos a imagem e retornamos a lista modificada e a imagem.

Com essa função, conseguimos realizar normalmente a clusterização de imagens, que é a parte inicial do nosso processo.

3.3 Funções Delta

Agora, precisamos criar as funções necessárias para realizar o processo de transferência de cores, ou seja, precisamos construir as ferramentas necessárias para resolver (2.23).

Os primeiros elementos que construímos são as funções Δ, começando por $\Delta_{X,Y}$:

$$\Delta_{X,Y}(\pi) = \text{diag}(\pi\mathbb{I})X - \pi Y. \tag{3.3}$$

No início, realizamos a construção matricial dessa equação. Devemos observar que $\pi \in \mathbb{R}^{N\times N}$, $X \in \mathbb{R}^{N\times 3}$ e $Y \in \mathbb{R}^{N\times 3}$. Ainda, para facilitar a escrita, vamos usar as seguintes notações:

$$\begin{aligned} l_i &= \pi_{i1} + \pi_{i2} + \cdots + \pi_{iN} \\ c_j &= \pi_{1j} + \pi_{2j} + \cdots + \pi_{Nj} \\ \pi_i &= (\pi_{i1}, \pi_{i2}, \ldots, \pi_{iN}) \\ \pi_j^* &= (\pi_{1j}, \pi_{2j}, \ldots, \pi_{Nj}) \\ X_j &= (X_{1j}, X_{2j}, \ldots, X_{Nj}) \\ Y_j &= (Y_{1j}, Y_{2j}, \ldots, Y_{Nj}). \end{aligned} \tag{3.4}$$

Temos que l_i é simplesmente a soma dos elementos da linha i de π, c_j é a soma dos elementos da coluna j, π_i é um vetor dado pela linha i de π, π_j^* é um vetor dado pela coluna j de π, X_j é um vetor dado pela coluna j de X e Y_j é um vetor dado pela coluna j de Y.

Começamos então construindo $\text{diag}(\pi\mathbb{I})$:

$$\begin{aligned}\text{diag}(\pi\mathbb{I}) &= \text{diag}\left(\begin{bmatrix}\pi_{11} & \pi_{12} & \dots & \pi_{1N}\\ \pi_{21} & \pi_{22} & \dots & \pi_{2N}\\ \vdots & \vdots & \ddots & \vdots\\ \pi_{N1} & \pi_{N2} & \dots & \pi_{NN}\end{bmatrix}\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}\right)\\ &= \begin{bmatrix}l_1 & 0 & \dots & 0\\ 0 & l_2 & \dots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \dots & l_N\end{bmatrix}. \end{aligned} \tag{3.5}$$

$$\begin{aligned}\text{diag}(\pi\mathbb{I})X &= \begin{bmatrix}l_1 & 0 & \dots & 0\\ 0 & l_2 & \dots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \dots & l_N\end{bmatrix}\begin{bmatrix}X_{11} & X_{12} & X_{13}\\ X_{21} & X_{22} & X_{23}\\ \vdots & \vdots & \vdots\\ X_{N1} & X_{N2} & X_{N3}\end{bmatrix}\\ &= \begin{bmatrix}l_1X_{11} & l_1X_{12} & l_1X_{13}\\ l_2X_{21} & l_2X_{22} & l_2X_{23}\\ \vdots & \vdots & \vdots\\ l_NX_{N1} & l_NX_{N2} & l_NX_{N3}\end{bmatrix}\end{aligned} \tag{3.6}$$

Também queremos construir πY.

$$\begin{aligned}\pi Y &= \begin{bmatrix}\pi_{11} & \pi_{12} & \dots & \pi_{1N}\\ \pi_{21} & \pi_{22} & \dots & \pi_{2N}\\ \vdots & \vdots & \ddots & \vdots\\ \pi_{N1} & \pi_{N2} & \dots & \pi_{NN}\end{bmatrix}\begin{bmatrix}Y_{11} & Y_{12} & Y_{13}\\ Y_{21} & Y_{22} & Y_{23}\\ \vdots & \vdots & \vdots\\ Y_{N1} & Y_{N2} & Y_{N3}\end{bmatrix}\\ &= \begin{bmatrix}\langle\pi_1, Y_1\rangle & \langle\pi_1, Y_2\rangle & \langle\pi_1, Y_3\rangle\\ \langle\pi_2, Y_1\rangle & \langle\pi_2, Y_2\rangle & \langle\pi_2, Y_3\rangle\\ \vdots & \vdots & \vdots\\ \langle\pi_N, Y_1\rangle & \langle\pi_N, Y_2\rangle & \langle\pi_N, Y_3\rangle\end{bmatrix}.\end{aligned} \tag{3.7}$$

Para terminar a construção, basta realizar a subtração e temos que $\Delta_{X,Y}$

equivale a

$$\begin{bmatrix} l_1X_{11} - \langle \pi_1, Y_1 \rangle & l_1X_{12} - \langle \pi_1, Y_2 \rangle & l_1X_{13} - \langle \pi_1, Y_3 \rangle \\ l_2X_{21} - \langle \pi_2, Y_1 \rangle & l_2X_{22} - \langle \pi_2, Y_2 \rangle & l_2X_{23} - \langle \pi_2, Y_3 \rangle \\ \vdots & \vdots & \vdots \\ l_NX_{N1} - \langle \pi_N, Y_1 \rangle & l_NX_{N2} - \langle \pi_N, X_Y \rangle & l_NX_{N3} - \langle \pi_N, X_Y \rangle \end{bmatrix}. \tag{3.8}$$

Passamos agora para a construção de $\Delta_{X,Y}(\pi)$ na linguagem Python, o que fazemos utilizando o seguinte código:

```
def delta_XY(listax, listay, pi):
    N = len(listax)
    piI = []

    for a in range(N):
        s = 0
        for b in range(N):
            s += pi[a*N + b]
        piI.append(s)

    piIX = []

    for a in range(N):
        piIX.append([piI[a]*listax[a][0],
        piI[a]*listax[a][1],piI[a]*listax[a][2]])

    piY = []

    for a in range(N):
        s0 = 0
        s1 = 0
        s2 = 0
        for b in range(N):
            s0 += pi[N*a + b]*listay[b][0]
            s1 += pi[N*a + b]*listay[b][1]
            s2 += pi[N*a + b]*listay[b][2]
        piY.append([s0,s1,s2])

    deltaXY = []

    for a in range(N):
        deltaXY.append([piIX[a][0] - piY[a][0],
        piIX[a][1] - piY[a][1],piIX[a][2] - piY[a][2]])

    return(deltaXY)
```

A ideia é muito parecida com a da construção matricial: começamos criando uma lista vazia piI, que terá o papel de diag($\pi\mathbb{I}$). Como só nos interessam os valores da diagonal principal, construímos piI como sendo uma lista com N entradas, cada uma sendo o valor da soma da respectiva linha de π.

Após, para realizar a criação de diag($\pi\mathbb{I}$)X, nos baseamos na matriz de (3.6), apenas multiplicamos cada elemento de X (listax[a][i]) pelo seu respectivo em diag($\pi\mathbb{I}$) (piI[a]).

Ainda, para construirmos πY (piY), também vamos nos basear pela construção matricial em (3.7). Para obter o elemento da posição (i, j), devemos realizar o produto escalar entre a linha i de π e a coluna j de X. Realizamos esse produto escalar em todos os 3 elementos de cada linha conjuntamente, através das variáveis s_0, s_1 e s_2.

Por fim, resta apenas a criação da lista deltaXY, que terá em cada posição i uma lista com 3 entradas, dadas pelas 3 coordenada da linha i de (3.8). Essa construção é relativamente fácil, pois precisamos apenas subtrair os valores de piIX por piY.

De maneira semelhante, podemos realizar a construção de $\Delta_{Y,X}$, que é dada por

$$\Delta_{Y,X}(\pi) = \text{diag}(\pi^*\mathbb{I})Y - \pi^*X. \tag{3.9}$$

Novamente vamos primeiro realizar a construção matricial: começamos pela criação de diag($\pi^*\mathbb{I}$):

$$\begin{aligned}\text{diag}(\pi^*\mathbb{I}) &= \text{diag}\left(\begin{bmatrix}\pi_{11} & \pi_{21} & \dots & \pi_{N1}\\ \pi_{12} & \pi_{22} & \dots & \pi_{N2}\\ \vdots & \vdots & \ddots & \vdots\\ \pi_{1N} & \pi_{2N} & \dots & \pi_{NN}\end{bmatrix}\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}\right)\\ &= \begin{bmatrix}c_1 & 0 & \dots & 0\\ 0 & c_2 & \dots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \dots & c_N\end{bmatrix}.\end{aligned} \tag{3.10}$$

Após isso, construimos $\text{diag}(\pi^*\mathbb{I})Y$:

$$\text{diag}(\pi^*\mathbb{I})Y = \begin{bmatrix} c_1 & 0 & \dots & 0 \\ 0 & c_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & c_N \end{bmatrix} \begin{bmatrix} Y_{11} & Y_{12} & Y_{13} \\ Y_{21} & Y_{22} & Y_{23} \\ \vdots & \vdots & \vdots \\ Y_{N1} & Y_{N2} & Y_{N3} \end{bmatrix}$$
$$= \begin{bmatrix} c_1Y_{11} & c_1Y_{12} & c_1Y_{13} \\ c_2Y_{21} & c_2Y_{22} & c_2Y_{23} \\ \vdots & \vdots & \vdots \\ c_NY_{N1} & c_NY_{N2} & c_NY_{N3} \end{bmatrix}. \tag{3.11}$$

Construimos também a matriz π^*X:

$$\pi^*X = \begin{bmatrix} \pi_{11} & \pi_{21} & \dots & \pi_{N1} \\ \pi_{12} & \pi_{22} & \dots & \pi_{N2} \\ \vdots & \vdots & \ddots & \vdots \\ \pi_{1N} & \pi_{2N} & \dots & \pi_{NN} \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ \vdots & \vdots & \vdots \\ X_{N1} & X_{N2} & X_{N3} \end{bmatrix}$$
$$= \begin{bmatrix} \langle \pi_1^*, X_1 \rangle & \langle \pi_1^*, X_2 \rangle & \langle \pi_1^*, X_3 \rangle \\ \langle \pi_2^*, X_1 \rangle & \langle \pi_2^*, X_2 \rangle & \langle \pi_2^*, X_3 \rangle \\ \vdots & \vdots & \vdots \\ \langle \pi_N^*, X_1 \rangle & \langle \pi_N^*, X_2 \rangle & \langle \pi_N^*, X_3 \rangle \end{bmatrix}. \tag{3.12}$$

Para terminar, basta realizarmos a subtração e vermos que $\Delta_{Y,X}(\pi)$ equivale a

$$\begin{bmatrix} c_1Y_{11} - \langle \pi_1^*, X_1 \rangle & c_1Y_{12} - \langle \pi_1^*, X_2 \rangle & c_1Y_{13} - \langle \pi_1^*, X_3 \rangle \\ c_2Y_{11} - \langle \pi_2^*, X_1 \rangle & c_2Y_{12} - \langle \pi_2^*, X_2 \rangle & c_2Y_{13} - \langle \pi_2^*, X_3 \rangle \\ \vdots & \vdots & \vdots \\ c_NY_{11} - \langle \pi_N^*, X_1 \rangle & c_NY_{12} - \langle \pi_N^*, X_2 \rangle & c_NY_{13} - \langle \pi_N^*, X_3 \rangle \end{bmatrix}. \tag{3.13}$$

O código que criamos para $\Delta_{Y,X}(\pi)$ é o seguinte:

```
def delta_YX(listax, listay, pi):
    N = len(listax)
    pi_conjugI = []

    for a in range(N):
        s = 0
        for b in range(N):
```

```
            s += pi[a + b*N]
        pi_conjugI.append(s)

    pi_conjugIY = []

    for a in range(N):
        pi_conjugIY.append([pi_conjugI[a]*listay[a][0],
        pi_conjugI[a]*listay[a][1],pi_conjugI[a]*listay[a][2]])

    pi_conjugX = []

    for a in range(N):
        s0 = 0
        s1 = 0
        s2 = 0
        for b in range(N):
            s0 += pi[a + N*b]*listax[b][0]
            s1 += pi[a + N*b]*listax[b][1]
            s2 += pi[a + N*b]*listax[b][2]
        pi_conjugX.append([s0,s1,s2])

    deltaYX = []

    for a in range(N):
        deltaYX.append([pi_conjugIY[a][0] - pi_conjugX[a][0],
        pi_conjugIY[a][1] - pi_conjugX[a][1],
        pi_conjugIY[a][2] - pi_conjugX[a][2]])

    return(deltaYX)
```

Começamos criando uma lista pi_conjugI com N entradas, sendo cada uma delas a soma da respectiva coluna de π, de acordo com (3.10).

Em seguida, criamos a matriz $\text{diag}(\pi^* I)Y$, em que simplesmente cada elemento de Y foi multiplicado pela soma dos elementos de uma coluna de π, de acordo com (3.11).

Após, também construímos a matriz $\pi^* X$. Podemos ver por (3.12) que um elemento na posição (i, j) é dado pelo produto escalar entre a coluna i de π e a coluna j de X. Realizamos então essa operação e seguimos para a parte final da função.

Por fim, basta criarmos a matriz $\Delta_{Y,X}(\pi)$ realizando a subtração entre as entradas de $\text{diag}(\pi^* \mathbb{I})Y$ e $\pi^* X$.

3.4 Grafo de uma nuvem de pontos

Nesta subseção, nós realizamos a construção da função que nos permite transformar uma lista de pixels em um grafo direcionado. A função construída foi a função graf, que tem como entrada uma lista de centroides e um valor k, que indica o número de ligações que cada vértice do grafo faz. Como saída, ela nos fornece o grafo direcionado dado por uma lista com 3 informações: a lista de pontos, a lista de arestas e a lista de pesos das arestas.

O código utilizado foi o seguinte:

```
def graf(lista, k):

    N = len(lista)
    Grafo = [lista,[],[]]

    for i in range(N):
        dist = []

        for j in range(N):
            Grafo[1].append(0)
            Grafo[2].append(0)
            dist.append((Grafo[0][i][0]-Grafo[0][j][0])**2
            + (Grafo[0][i][1]-Grafo[0][j][1])**2
            + (Grafo[0][i][2]-Grafo[0][j][2])**2)
        dist2 = dist[:]
        dist2.sort()

        for s in range(1,k+1):
            a = dist.index(dist2[s])
            Grafo[1][N*i + a] = 1
            Grafo[2][N*i + a] =
            1/sqrt((Grafo[0][i][0]-Grafo[0][a][0])**2
            + (Grafo[0][i][1]-Grafo[0][a][1])**2
            + (Grafo[0][i][2]-Grafo[0][a][2])**2)
            dist[a] = 0

    return(Grafo)
```

Começamos definindo N como sendo o número de pontos e também criamos uma lista (Grafo) com 3 informações: na primeira entrada a lista de pontos (que já está pronta), na segunda as arestas do grafo e na terceira os pesos dados a cada aresta.

Para descobrir as arestas do nosso grafo precisamos, para cada ponto

da nossa lista, encontrar quais são os seus k pontos mais próximos. Para isso criamos, para cada ponto, uma lista com N entradas que indica todas as distâncias possíveis entre ele e os outros pontos. Já as listas Grafo[1] (arestas) e Grafo[2] (pesos) têm $N \times N$ entradas, sendo as N primeiras relativas às distâncias relacionadas ao primeiro centroide, as próximas N relacionadas ao segundo, e assim por diante.

Após realizarmos a criação da lista dist para um centroide e calcular todas as distâncias, criamos uma cópia dessa lista chamada dist2, essa lista é então reordenada da menor distância para a maior. Com isso, basta buscarmos os índices em dist que correspondem às k primeiras entradas de dist2.[5]

Ao identificar a quais outros pontos um vértice está ligado, trocamos o valor correspondente a esta ligação para 1 em Grafo[1] e para o inverso da Distância Euclidiana em Grafo[2].

Por fim, trocamos o valor dessa ligação na lista dist por 0. Fazemos isso para que, caso tenhamos dois pontos com a mesma distância do vértice inicial, então não seja considerado o mesmo ponto duas vezes.

Ao terminar as iterações, retornamos o grafo relativo à nossa nuvem de pontos.

3.5 Gradiente de um grafo

Vamos agora construir a função G, dada por $G_X(V) = (w_{i,j}(V_i - V_j))_{(i,j)\in E_X}$. Como entrada para a nossa função temos três informações: a lista de pontos X, a lista de pontos V e o parâmetro k, que indica o número de ligações de cada vértice do grafo. O código que utilizamos é o seguinte:

```
def G(listaX, listaV, k):
    N = len(listaX)
    x = graf(listaX, k)
    Gx = []

    for i in range(N):
        for j in range(N):
            if x[1][N*i + j] != 0:
                Gx.append(
                [x[2][N*i + j]*(listaV[i][0] - listaV[j][0]),
                x[2][N*i + j]*(listaV[i][1] - listaV[j][1]),
```

[5] Na verdade não consideramos a primeira entrada de dist, pois esse valor vale 0 e representa a distância do ponto até ele mesmo.

```
                x[2][N*i + j]*(listaV[i][2] - listaV[j][2])])

    return(Gx)
```

Começamos definindo N como o número de vértices no nosso grafo e construindo o grafo relativo a estes vértices. Após, construímos uma lista Gx que, a cada vez que tivermos uma aresta de posição (i, j) no grafo de X, armazena um valor dado por $w_{i,j}(V_i - V_j)$.

3.6 Resolução do problema de programação linear

Passamos agora para a construção da função que resolve o problema de programação linear dado em (2.23). Ela tem 9 entradas: uma lista relativa à imagem base (pixels, largura e centroides), uma lista relativa à imagem modelo (pixels, largura e centroides), os parâmetros $k_X, K_X, k_Y, K_Y, \lambda_X, \lambda_Y$ e k. Como saída, temos a matriz resultante π. Na construção da função prog_linear, utilizamos as funções LpProblem, presente no módulo pulp. Abaixo, vemos o código utilizado para construir essa função:

```
def prog_linear(lista1, lista2, k_X, K_X, k_Y, K_Y,
lambdax, lambday, k):
    N = len(lista1[2])
    P = N*k
    def C(a,b):
        return sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2
        + (a[2]-b[2])**2)

    modelo = LpProblem(name = "transporte_otimo",
    sense =LpMinimize)
    variaveis = {i: LpVariable(name=f"pi{i}",
    lowBound=0, upBound=1) for i in range(N**2)}

    for i in range(P):
        for j in range(3):
            variaveis[N**2 + 3*i + j]
            = LpVariable(name=f'UX_{i},{j}',lowBound=0)

    for i in range(P):
        for j in range(3):
            variaveis[N**2 + 3*P + 3*i + j]
            = LpVariable(name=f'UY_{i},{j}', lowBound=0)
```

```
x = G(lista1[2], delta_XY(lista1[2], lista2[2],
variaveis), k)
y = G(lista2[2], delta_YX(lista1[2], lista2[2],
variaveis),k)

for i in range(P):
    for j in range(3):
        modelo += x[i][j] <= variaveis[N**2 + 3*i + j]
        modelo += -variaveis[N**2 + 3*i + j] <= x[i][j]
        modelo += y[i][j] <=
        variaveis[N**2 +3*P + 3*i + j]
        modelo += -variaveis[N**2 +3*P + 3*i + j] <=
        y[i][j]

for i in range(N):
    modelo += (lpSum(variaveis[j] for j in
    range(i*N, i*N + N)) <= K_X)
    modelo += (lpSum(variaveis[j] for j in
    range(i*N, i*N + N)) >= k_X)
    modelo += (lpSum(variaveis[N*j+i] for j in
    range(N)) <= K_Y)
    modelo += (lpSum(variaveis[N*j+i] for j in
    range(N)) >= k_Y)
    modelo += (lpSum(variaveis[j] for j in
    range(N**2)) == N)

constantes = []
for j in range(N**2):
    constantes.append(
    C(lista1[2][j//N], lista2[2][j % N]))
for j in range(3*P):
    constantes.append(lambdax)
for j in range(3*P):
    constantes.append(lambday)

z = lpSum(variaveis[j]*constantes[j] for j in
range(N**2+6*P))

modelo += z
modelo.solve()

resultado = []
for var in variaveis.values():
    resultado.append(var.value())
```

```
    return resultado
```

Começamos definindo N como sendo o número de centroides utilizados na imagem modelo e também definindo uma constante $P = N \cdot k$ dada pelo total de arestas no grafo da imagem modelo.

Em seguida, definimos a função custo, que é dada por

$$C(a, b) = \sqrt{(a[0] - b[0])^2 + (a[1] - b[1])^2 + (a[2] - b[2])^2}.$$

O próximo passo é definir o nosso modelo de programação linear, criamos ele com o nome "transporte ótimo" e indicamos que é um problema de minimização.

Passamos então para a parte onde definimos as variáveis do nosso problema. Por (2.23), vemos que as variáveis são dadas pelas entradas de π e pelas entradas nas variáveis auxiliares U_X e U_Y. Temos que existem N^2 variáveis relativas à matriz π, que podem assumir valores reais entre 0 e 1. Também temos $3P$ variáveis relativas a U_X e $3P$ relativas a U_Y, isso porque essas variáveis são utilizadas para limitar $G_X(\delta_{X,Y}(\pi)), G_Y(\delta_{Y,X}(\pi)) \in \mathbb{R}^{P\times 3}$. Temos que essas variáveis só precisam ser não-negativas, sem outro tipo de limitação.

Continuando, definimos $x = G_X(\Delta_{X,Y}(\pi))$ e $y = G_Y(\Delta_{Y,X}(\pi))$.

Após, passamos a definir as restrições do problema, que são várias:

$$\begin{array}{ccc} x \leq U_X & -U_X \leq x & y \leq U_Y \\ -U_Y \leq y & \overline{\pi}\mathbb{I} \leq K_X & \overline{\pi}\mathbb{I} \geq k_X \\ \overline{\pi}^*\mathbb{I} \leq K_Y & \overline{\pi}^*\mathbb{I} \geq k_Y & \mathbb{I}\overline{\pi}\mathbb{I} = N. \end{array} \tag{3.14}$$

Em seguida, criamos uma lista (constantes) que será dada pelos valores que multiplicam as variáveis dentro do nosso modelo, ou seja, nas N^2 primeiras entradas temos os valores da função custo aplicada nos centroides da imagem modelo, nas $3P$ entradas seguintes temos o valor de λ_X e nas $3P$ variáveis restantes temos o valor de λ_Y.

Então, construímos a função $z = \langle C_{X,Y}, \overline{\pi}\rangle + \lambda_X\langle U_X, \mathbb{I}\rangle + \lambda_Y\langle U_Y, \mathbb{I}\rangle$ e a colocamos dentro do problema. Por fim, resolvemos o programa linear utilizando o Método Simplex e retornamos os valores das variáveis através de uma lista com $N^2 + 6P$ elementos.

3.7 Transformação final

A próxima função que construimos é aquela que, a partir da matriz obtida com a função prog_linear, constrói a nossa imagem final. Como valores de entrada da nossa função temos os seguintes: lista relativa à imagem base (pixels, largura e centroides), lista relativa à imagem modelo (pixels, largura e centroides), parâmetros $k_X, K_X, k_Y, K_Y, \lambda_X, \lambda_Y$ e k. Como saída, temos a imagem final, resultado da transformação de transferência de cores. O código utilizado foi o seguinte:

```
def transformacao(lista1, lista2, k_X, K_X, k_Y, K_Y,
lambdax,lambday,k):

    n = len(lista1[0])
    L = lista1[1]
    A = n//A
    N = len(lista1[2])

    img = Image.new("RGB", (L,A), (0,0,0))
    J = prog_linear(lista1, lista2, k_X, K_X, k_Y, K_Y,
    lambdax,lambday,k)

    T = []
    for i in range(N):
        s10 = 0
        s11 = 0
        s12 = 0
        s2 = 0
        for j in range(N):
            s10 += J[N*i + j]*lista2[2][j][0]
            s11 += J[N*i + j]*lista2[2][j][1]
            s12 += J[N*i + j]*lista2[2][j][2]
            s += J[N*i + j]
        T.append([int(s10/s), int(s11/s), int(s12/s)])

    for y in range(A):
        for x in range(L):
            p = lista1[0][L*y + x][3]
            img.putpixel((x,y),(int(T[p][0]),
            int(T[p][1]),int(T[p][2])))

    img.save(out_file(f"camelo-k={k}-N={N}-ks
␣␣␣␣=({k_X}-{K_X}-{k_Y}-{K_Y})-lambdas=({lambdax}-{lambday})
␣␣␣␣-sem-interpolacao.png"))
```

```
    for y in range(A):
        for x in range(L):
            p = lista1[0][L*y + x][3]
            img.putpixel((x,y),
            (int(T[p][0] + lista1[0][L*y+x][0]
            - lista1[2][p][0]), int(T[p][1]
            + lista1[0][L*y+x][1] - lista1[2][p][1]),
            int(T[p][2] + lista1[0][L*y+x][2]
            - lista1[2][p][2])))

    img.save(out_file(f"camelo-k={k}-N={N}-ks=({k_X}-{K_X}
␣␣␣␣-{k_Y}-{K_Y})-lambdas=({lambdax}-{lambday}).png"))
```

Começamos definindo n como o número total de pixels da imagem base (precisamos necessariamente obter essa informação da imagem base porque a nossa imagem final tem o mesmo formato que ela. Também obtemos a largura L e a altura A da imagem base, além do número de centroides N.

Depois, criamos uma imagem totalmente em preto com as dimensões obtidas anteriormente e em seguida aplicamos a função prog_linear nas listas e parâmetros selecionados, obtendo a lista J, cujas N^2 primeiras entradas representam a matriz π.

Após isso, queremos criar a transformação T que foi definida em (2.15). Para isso, criamos uma lista T e também variáveis s_{10}, s_{11} e s_{12}, que representam, para cada $i = 0, 1, \ldots, N-1$, o somatório $\sum_{j=1}^{N} \pi_{i,j} y_j$ restrito à coordenada $1, 2$ ou 3 do pixel, respectivamente. Também criamos a variável s, que tem o papel do somatório $\sum_{j=1}^{N} \pi_{i,j}$. Assim, para cada $i = 0, \ldots, N-1$, adicionamos a T o pixel $\left(int\left(\frac{s_{10}}{s}\right), int\left(\frac{s_{11}}{s}\right), int\left(\frac{s_{12}}{s}\right)\right)$, onde $int(x)$ indica a retirada da parte decimal de x, tornando-o um número inteiro. Fazemos esse arredondamento porque as coordenada da cor de um pixel precisam ser números inteiros.

Por fim, realizamos a construção da imagem final, pois sabemos a qual centroide está associado cada pixel (quarta entrada na lista de pixels) e sabemos em qual cor deve ser mandado cada centroide (definido pela lista T). Também salvamos a imagem obtida na pasta Output com um nome que indique todos os parâmetro escolhidos.

Após isso realizamos novamente a transformação, mas dessa vez aplicando

a interpolação, definida em (2.25). Também salvamos a imagem obtida com um nome adequado.

3.8 Transferência de cores

Para finalizar o nosso algoritmo, criamos uma função que basicamente junta todas as outras funções para realizar a transferência de cores entre duas imagens. Como queremos realizar a transferência de cores com diversos valores diferentes para cada parâmetro, realizamos uma mudança na forma como os parâmetros entram na função. Ao invés de considerarmos cada parâmetro como um número, vamos considerá-lo como uma lista de números, assim conseguimos realizar diversas transformações com apenas um acionamento do código.

Assim, temos que essa função recebe como entrada a imagem base, a imagem modelo e 8 listas com todos os valores de $N, k_X, K_X, k_Y, K_Y, \lambda_X, \lambda_Y$ e k. Como saída temos todas as imagens transformadas, uma para cada combinação de parâmetros escolhida. O código para a nossa função é o seguinte:

```
def transferencia(imagem1,imagem2,N,k_X, K_X, k_Y, K_Y,
lambdax,lambday,k):

    for j in N:
        x1 = k_means(imagem1,j)
        y1 = k_means(imagem2,j)
        x2 = criador(x1)
        y2 = criador(y1)
        for l in k:
            for i in range(len(K_Y)):
                for m in range(len(lambdax)):
                    img = transformacao(
                    x2[0],y2[0],k_X[i], K_X[i], k_Y[i],
                    K_Y[i], lambdax[m], lambday[m], l)
```

No código acima, criamos um laço nos valores de N e realizamos a clusterização das duas imagens com esses valores. Após, para cada um destes valores de N, criamos laços nos outros parâmetros e aplicamos a função transformacao, gerando a imagem final com os valores escolhidos.

Capítulo 4

Resultados

Neste capítulo, apresentamos os principais resultados obtidos com o algoritmo construído anteriormente. Durante a construção do algoritmo, diversos parâmetros foram utilizados: o número N de cores utilizadas na clusterização, os parâmetros de relaxamento k_X, K_X, k_Y e K_Y, os parâmetros de regularização λ_X e λ_Y e também o número k de ligações na construção dos grafo. Cada um destes parâmetros produz efeitos diferentes no resultado final, conforme foi discutido no Capítulo 2.

Para obtermos o melhor resultado visual possível, é necessário aplicar o algoritmo com diversas mudanças nestes parâmetros, a fim de selecionar aqueles com melhores resultados. É importante ressaltar que nem toda escolha de parâmetro produz um resultado satisfatório. Abaixo temos um exemplo de valores utilizados para a construção das imagens.

```
if __name__ == '__main__':

    N = [50, 100, 200]

    k_X = [1,0.1,0.7,0.7,0.9,1,1,1]
    K_X = [1,1,1,1.1,1.1,1.5,2,8]
    k_Y = [1,0.1,0.7,0.7,0.9,1,1,1]
    K_Y = [1,1,1,1.1,1.1,1.5,2,8]

    k = [1,2,4,5,10]

    lambdax = [0,0.001,0.5,0.1,0.2,0.3,0.5,1,1.5,2,10]
    lambday = [0,0.001,0.5,0.1,0.2,0.3,0.5,1,1.5,2,10]

    x0 = Image.open(in_file("camelo.jpg"))
```

```
    y0 = Image.open(in_file("estrela.jpg"))

    transferencia(x0,y0,N,k_X,K_X,k_Y,K_Y,lambdax,lambday,k)
```

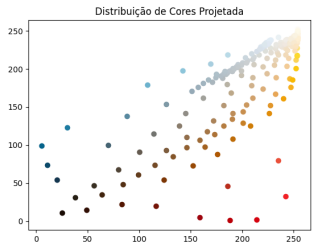
No código acima, escolhemos alguns valores possíveis para N, k_X, K_X, k_Y e K_Y, k, λ_X e λ_Y. Após isso, escolhemos as imagens que utilizamos e aplicamos a função transferencia.

Na Figura 4.1, vemos alguns resultados obtidos com diferentes parâmetros. Nas figuras (a) e (c) temos a imagem base e a sua distribuição de cores projetada no plano $R \times G$, nas figuras (b) e (d) temos a imagem modelo e a sua distribuição de cores. Na figura (e) temos o resultado do transporte ótimo padrão de Kantorovich, ou seja, utilizando $(N, k, k_X, K_X, k_Y, K_Y, \lambda_X, \lambda_Y) = (200, 4, 1, 1, 1, 1, 0, 0)$. Na figura (h) temos a sua distribuição de cores, podemos ver que mesmo sem ter um resultado visual satisfatório, a distribuição de cores ficou muito próxima à distribuição da imagem modelo (d). Nas figuras (f) e (i) temos um resultado relaxado e regularizado e a sua distribuição de cores, utilizando $(N, k, k_X, K_X, k_Y, K_Y, \lambda_X, \lambda_Y) = (200, 4, 0.2, 1, 0.2, 1, 0.05, 0.05)$. Podemos ver que o resultado visual melhora consideravelmente, sem modificar muito o padrão de distribuição de cores. Já nas figuras (g) e (j), temos o resultado utilizando $(N, k, k_X, K_X, k_Y, K_Y, \lambda_X, \lambda_Y) = (200, 4, 0.1, 8, 0.1, 8, 0.2, 0.2)$ e a sua respectiva distribuição de cores. Neste caso podemos observar que, apesar de termos um bom resultado visual (imagem nítida), as cores obtidas não estão de acordo com a imagem modelo, como podemos ver pela distribuição de cores (que não apresenta quase nenhum ponto verde, cor muito relevante na distribuição da imagem modelo). Vemos assim a importância da escolha correta dos parâmetros no método para obter um resultado mais satisfatório.

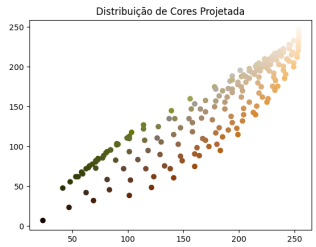
Entretanto, ao escolhermos bons parâmetros, temos resultados realmente satisfatórios. Na Figura 4.2, vemos o resultado da aplicação do método em diferentes conjuntos de imagens. Na coluna da esquerda, temos as imagens base e modelo, já na coluna da direita vemos os resultados obtidos.

(a) Imagem base.

(b) Imagem modelo.



(c) Distribuição da imagem base.

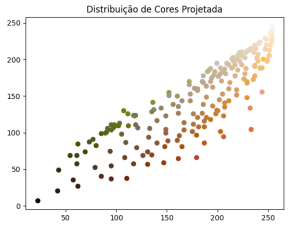


(d) Distribuição da imagem modelo.

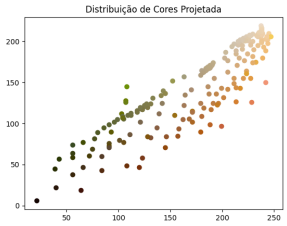
(e) Resultado do problema de Kantorovich.

(f) Resultado relaxado e regularizado satisfatório.

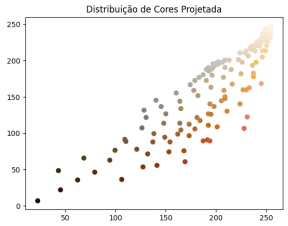
(g) Resultado relaxado e regularizado não satisfatório.



(h) Distribuição de (e).



(i) Distribuição de (f).



(j) Distribuição de (g).

Figura 4.1: Comparação de diferentes resultados ao mudarmos os parâmetros do problema.

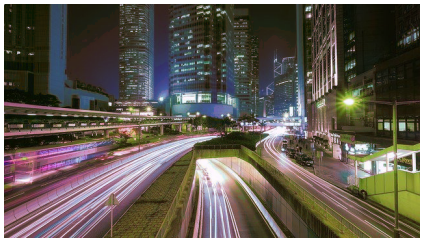
(a) Foto de um lago.

(b) Lago modificado por (c).

(c) Foto de uma floresta.

(d) Floresta modificada por (a).

(e) Foto de uma cidade.



(f) Cidade modificada por (g).

(g) Foto de uma montanha.

(h) Montanha modificada por (e).

Figura 4.2: Resultado do algoritmo aplicado em diferentes imagens.

Créditos das Imagens

Figura 1.1a: Freepik.com, foto de master1305.

Figura 2.3a: Pexels.com, foto de Tara Winstead.

Figura 2.3b: Pexels.com, foto de Artem Podrez.

Figura 2.4c: Freepik.com.

Figura 4.2a: Pixabay.com.

Figura 4.2c: Pixabay.com, foto de Valentin.

Figura 4.2e: Pixabay.com.

Figura 4.2g: Pixabay.com, foto de Bessi.

Apêndice: Tabela com os tempos computacionais

Neste apêndice, construímos algumas tabelas com o tempo levado para realizar diversas aplicações do algoritmo construído, a fim de analisar o papel dos diversos parâmetros no custo computacional do método.

É importante ressaltar que o tempo absoluto do algoritmo varia dependendo da capacidade de processamento de cada máquina. Realizamos todas as execuções do algoritmo no mesmo computador, portanto nosso objetivo é analisar comparativamente os tempos obtidos ao variar os parâmetros $N, k_X, K_X, k_Y, K_Y, k, \lambda_X$ e λ_Y, além de observar a influência do tamanho das imagens base e modelo no tempo computacional. Não é o nosso objetivo analisar a eficiência geral do algoritmo construído.

Nas Tabelas 4.1, 4.2, 4.3, 4.4, 4.5 e 4.6, utilizamos as imagens base e modelo com dimensões 1149×768 e 1148×768 pixels, respectivamente. Na Tabela 4.7, utilizamos ambas as imagens base e modelo com dimensões de 512×512 pixels.

Ao fixarmos todos os parâmetros e variarmos N, vemos que o tempo computacional cresce conforme esse parâmetro aumenta. Afinal o parâmetro N controla o número de cores da clusterização e também influencia o número de variáveis no problema de programação linear.

Também observamos um comportamento parecido quando variamos apenas o parâmetro k. Conforme este parâmetro aumenta, aumenta também o tempo computacional. Esse comportamento é melhor observado para valores maiores de N, pois o parâmetro k controla o número de arestas saindo de cada vértice do grafo, quando N é pequeno o número de arestas também é pequeno, mesmo com um k grande.

Quando analisamos os parâmetros $k_X, K_X, k_Y, K_Y, \lambda_X$ e λ_Y, temos que eles não impactam significativamente no tempo computacional do método.

N	k	(k_X, K_X, k_Y, K_Y)	(λ_X, λ_Y)	Tempo (s)
5	1	(1,1,1,1)	(0,0)	44
5	1	(1,1,1,1)	(0.1,0.1)	43
5	1	(1,1,1,1)	(2,2)	44
5	1	(0.9,1.1,0.9,1.1)	(0,0)	44
5	1	(0.9,1.1,0.9,1.1)	(0.1,0.1)	44
5	1	(0.9,1.1,0.9,1.1)	(2,2)	43
5	2	(1,1,1,1)	(0,0)	43
5	2	(1,1,1,1)	(0.1,0.1)	44
5	2	(1,1,1,1)	(2,2)	41
5	2	(0.9,1.1,0.9,1.1)	(0,0)	42
5	2	(0.9,1.1,0.9,1.1)	(0.1,0.1)	42
5	2	(0.9,1.1,0.9,1.1)	(2,2)	44
5	4	(1,1,1,1)	(0,0)	43
5	4	(1,1,1,1)	(0.1,0.1)	44
5	4	(1,1,1,1)	(2,2)	44
5	4	(0.9,1.1,0.9,1.1)	(0,0)	44
5	4	(0.9,1.1,0.9,1.1)	(0.1,0.1)	42
5	4	(0.9,1.1,0.9,1.1)	(2,2)	42

Tabela 4.1: Tempos computacionais, com $N = 5$.

Um outro fator que influencia significativamente o tempo total do algoritmo é o tamanho das imagens base e modelo: ao utilizarmos imagens com dimensões menores, o tempo computacional necessário é reduzido consideravelmente. Isso se deve ao fato de que o tempo para realizar a clusterização destas imagens é reduzido, pois estamos utilizando um número menor de pixels.

10	1	(1,1,1,1)	(0,0)	82
10	1	(1,1,1,1)	(0.1,0.1)	82
10	1	(1,1,1,1)	(2,2)	81
10	1	(0.9,1.1,0.9,1.1)	(0,0)	82
10	1	(0.9,1.1,0.9,1.1)	(0.1,0.1)	81
10	1	(0.9,1.1,0.9,1.1)	(2,2)	80
10	2	(1,1,1,1)	(0,0)	80
10	2	(1,1,1,1)	(0.1,0.1)	80
10	2	(1,1,1,1)	(2,2)	81
10	2	(0.9,1.1,0.9,1.1)	(0,0)	80
10	2	(0.9,1.1,0.9,1.1)	(0.1,0.1)	81
10	2	(0.9,1.1,0.9,1.1)	(2,2)	81
10	4	(1,1,1,1)	(0,0)	82
10	4	(1,1,1,1)	(0.1,0.1)	81
10	4	(1,1,1,1)	(2,2)	79
10	4	(0.9,1.1,0.9,1.1)	(0,0)	83
10	4	(0.9,1.1,0.9,1.1)	(0.1,0.1)	81
10	4	(0.9,1.1,0.9,1.1)	(2,2)	80
10	8	(1,1,1,1)	(0,0)	79
10	8	(1,1,1,1)	(0.1,0.1)	77
10	8	(1,1,1,1)	(2,2)	80
10	8	(0.9,1.1,0.9,1.1)	(0,0)	78
10	8	(0.9,1.1,0.9,1.1)	(0.1,0.1)	78
10	8	(0.9,1.1,0.9,1.1)	(2,2)	91

Tabela 4.2: Tempos computacionais, com $N = 10$.

20	1	(1,1,1,1)	(0,0)	144
20	1	(1,1,1,1)	(0.1,0.1)	147
20	1	(1,1,1,1)	(2,2)	147
20	1	(0.9,1.1,0.9,1.1)	(0,0)	152
20	1	(0.9,1.1,0.9,1.1)	(0.1,0.1)	150
20	1	(0.9,1.1,0.9,1.1)	(2,2)	147
20	2	(1,1,1,1)	(0,0)	152
20	2	(1,1,1,1)	(0.1,0.1)	149
20	2	(1,1,1,1)	(2,2)	149
20	2	(0.9,1.1,0.9,1.1)	(0,0)	151
20	2	(0.9,1.1,0.9,1.1)	(0.1,0.1)	150
20	2	(0.9,1.1,0.9,1.1)	(2,2)	149
20	4	(1,1,1,1)	(0,0)	143
20	4	(1,1,1,1)	(0.1,0.1)	150
20	4	(1,1,1,1)	(2,2)	148
20	4	(0.9,1.1,0.9,1.1)	(0,0)	148
20	4	(0.9,1.1,0.9,1.1)	(0.1,0.1)	148
20	4	(0.9,1.1,0.9,1.1)	(2,2)	146
20	8	(1,1,1,1)	(0,0)	145
20	8	(1,1,1,1)	(0.1,0.1)	143
20	8	(1,1,1,1)	(2,2)	148
20	8	(0.9,1.1,0.9,1.1)	(0,0)	151
20	8	(0.9,1.1,0.9,1.1)	(0.1,0.1)	148
20	8	(0.9,1.1,0.9,1.1)	(2,2)	145
20	16	(1,1,1,1)	(0,0)	149
20	16	(1,1,1,1)	(0.1,0.1)	149
20	16	(1,1,1,1)	(2,2)	149
20	16	(0.9,1.1,0.9,1.1)	(0,0)	152
20	16	(0.9,1.1,0.9,1.1)	(0.1,0.1)	148
20	16	(0.9,1.1,0.9,1.1)	(2,2)	148

Tabela 4.3: Tempos computacionais, com $N = 20$.

50	1	(1,1,1,1)	(0,0)	326
50	1	(1,1,1,1)	(0.1,0.1)	334
50	1	(1,1,1,1)	(2,2)	326
50	1	(0.9,1.1,0.9,1.1)	(0,0)	330
50	1	(0.9,1.1,0.9,1.1)	(0.1,0.1)	331
50	1	(0.9,1.1,0.9,1.1)	(2,2)	330
50	2	(1,1,1,1)	(0,0)	333
50	2	(1,1,1,1)	(0.1,0.1)	335
50	2	(1,1,1,1)	(2,2)	335
50	2	(0.9,1.1,0.9,1.1)	(0,0)	335
50	2	(0.9,1.1,0.9,1.1)	(0.1,0.1)	335
50	2	(0.9,1.1,0.9,1.1)	(2,2)	337
50	4	(1,1,1,1)	(0,0)	337
50	4	(1,1,1,1)	(0.1,0.1)	336
50	4	(1,1,1,1)	(2,2)	337
50	4	(0.9,1.1,0.9,1.1)	(0,0)	338
50	4	(0.9,1.1,0.9,1.1)	(0.1,0.1)	333
50	4	(0.9,1.1,0.9,1.1)	(2,2)	347
50	8	(1,1,1,1)	(0,0)	352
50	8	(1,1,1,1)	(0.1,0.1)	388
50	8	(1,1,1,1)	(2,2)	331
50	8	(0.9,1.1,0.9,1.1)	(0,0)	335
50	8	(0.9,1.1,0.9,1.1)	(0.1,0.1)	336
50	8	(0.9,1.1,0.9,1.1)	(2,2)	332
50	16	(1,1,1,1)	(0,0)	338
50	16	(1,1,1,1)	(0.1,0.1)	360
50	16	(1,1,1,1)	(2,2)	367
50	16	(0.9,1.1,0.9,1.1)	(0,0)	340
50	16	(0.9,1.1,0.9,1.1)	(0.1,0.1)	336
50	16	(0.9,1.1,0.9,1.1)	(2,2)	339
50	32	(1,1,1,1)	(0,0)	342
50	32	(1,1,1,1)	(0.1,0.1)	365
50	32	(1,1,1,1)	(2,2)	371
50	32	(0.9,1.1,0.9,1.1)	(0,0)	342
50	32	(0.9,1.1,0.9,1.1)	(0.1,0.1)	370
50	32	(0.9,1.1,0.9,1.1)	(2,2)	363

Tabela 4.4: Tempos computacionais, com $N = 50$.

100	1	(1,1,1,1)	(0,0)	627
100	1	(1,1,1,1)	(0.1,0.1)	626
100	1	(1,1,1,1)	(2,2)	629
100	1	(0.9,1.1,0.9,1.1)	(0,0)	635
100	1	(0.9,1.1,0.9,1.1)	(0.1,0.1)	638
100	1	(0.9,1.1,0.9,1.1)	(2,2)	628
100	2	(1,1,1,1)	(0,0)	624
100	2	(1,1,1,1)	(0.1,0.1)	627
100	2	(1,1,1,1)	(2,2)	621
100	2	(0.9,1.1,0.9,1.1)	(0,0)	617
100	2	(0.9,1.1,0.9,1.1)	(0.1,0.1)	621
100	2	(0.9,1.1,0.9,1.1)	(2,2)	621
100	4	(1,1,1,1)	(0,0)	646
100	4	(1,1,1,1)	(0.1,0.1)	625
100	4	(1,1,1,1)	(2,2)	633
100	4	(0.9,1.1,0.9,1.1)	(0,0)	621
100	4	(0.9,1.1,0.9,1.1)	(0.1,0.1)	623
100	4	(0.9,1.1,0.9,1.1)	(2,2)	628
100	8	(1,1,1,1)	(0,0)	624
100	8	(1,1,1,1)	(0.1,0.1)	623
100	8	(1,1,1,1)	(2,2)	642
100	8	(0.9,1.1,0.9,1.1)	(0,0)	614
100	8	(0.9,1.1,0.9,1.1)	(0.1,0.1)	624
100	8	(0.9,1.1,0.9,1.1)	(2,2)	643
100	16	(1,1,1,1)	(0,0)	623
100	16	(1,1,1,1)	(0.1,0.1)	659
100	16	(1,1,1,1)	(2,2)	698
100	16	(0.9,1.1,0.9,1.1)	(0,0)	624
100	16	(0.9,1.1,0.9,1.1)	(0.1,0.1)	656
100	16	(0.9,1.1,0.9,1.1)	(2,2)	701
100	32	(1,1,1,1)	(0,0)	652
100	32	(1,1,1,1)	(0.1,0.1)	794
100	32	(1,1,1,1)	(2,2)	904
100	32	(0.9,1.1,0.9,1.1)	(0,0)	675
100	32	(0.9,1.1,0.9,1.1)	(0.1,0.1)	820
100	32	(0.9,1.1,0.9,1.1)	(2,2)	873

Tabela 4.5: Tempos computacionais, com $N = 100$.

200	1	(1,1,1,1)	(0,0)	1214
200	1	(1,1,1,1)	(0.1,0.1)	1212
200	1	(1,1,1,1)	(2,2)	1243
200	1	(0.9,1.1,0.9,1.1)	(0,0)	1218
200	1	(0.9,1.1,0.9,1.1)	(0.1,0.1)	1240
200	1	(0.9,1.1,0.9,1.1)	(2,2)	1238
200	2	(1,1,1,1)	(0,0)	1232
200	2	(1,1,1,1)	(0.1,0.1)	1257
200	2	(1,1,1,1)	(2,2)	1301
200	2	(0.9,1.1,0.9,1.1)	(0,0)	1218
200	2	(0.9,1.1,0.9,1.1)	(0.1,0.1)	1254
200	2	(0.9,1.1,0.9,1.1)	(2,2)	1354
200	4	(1,1,1,1)	(0,0)	1347
200	4	(1,1,1,1)	(0.1,0.1)	1311
200	4	(1,1,1,1)	(2,2)	1407
200	4	(0.9,1.1,0.9,1.1)	(0,0)	1234
200	4	(0.9,1.1,0.9,1.1)	(0.1,0.1)	1329
200	4	(0.9,1.1,0.9,1.1)	(2,2)	1398
200	8	(1,1,1,1)	(0,0)	1379
200	8	(1,1,1,1)	(0.1,0.1)	1700
200	8	(1,1,1,1)	(2,2)	1656
200	8	(0.9,1.1,0.9,1.1)	(0,0)	1300
200	8	(0.9,1.1,0.9,1.1)	(0.1,0.1)	1515
200	8	(0.9,1.1,0.9,1.1)	(2,2)	1717
200	16	(1,1,1,1)	(0,0)	2145
200	16	(1,1,1,1)	(0.1,0.1)	2154
200	16	(1,1,1,1)	(2,2)	3038
200	16	(0.9,1.1,0.9,1.1)	(0,0)	2130
200	16	(0.9,1.1,0.9,1.1)	(0.1,0.1)	2828
200	16	(0.9,1.1,0.9,1.1)	(2,2)	3022
200	32	(1,1,1,1)	(0,0)	5132
200	32	(1,1,1,1)	(0.1,0.1)	7235
200	32	(1,1,1,1)	(2,2)	7468
200	32	(0.9,1.1,0.9,1.1)	(0,0)	6314
200	32	(0.9,1.1,0.9,1.1)	(0.1,0.1)	7823
200	32	(0.9,1.1,0.9,1.1)	(2,2)	7886

Tabela 4.6: Tempos computacionais, com $N = 200$.

N	k	(k_X, K_X, k_Y, K_Y)	(λ_X, λ_Y)	Tempo (s)
5	4	(1,1,1,1)	(0,0)	13
5	4	(1,1,1,1)	(0.1,0.1)	13
5	4	(1,1,1,1)	(2,2)	13
10	4	(1,1,1,1)	(0,0)	24
10	4	(1,1,1,1)	(0.1,0.1)	23
10	4	(1,1,1,1)	(2,2)	25
20	4	(1,1,1,1)	(0,0)	44
20	4	(1,1,1,1)	(0.1,0.1)	42
20	4	(1,1,1,1)	(2,2)	42
50	4	(1,1,1,1)	(0,0)	101
50	4	(1,1,1,1)	(0.1,0.1)	106
50	4	(1,1,1,1)	(2,2)	102
100	4	(1,1,1,1)	(0,0)	211
100	4	(1,1,1,1)	(0.1,0.1)	217
100	4	(1,1,1,1)	(2,2)	235
200	4	(1,1,1,1)	(0,0)	478
200	4	(1,1,1,1)	(0.1,0.1)	485
200	4	(1,1,1,1)	(2,2)	570

Tabela 4.7: Tempos computacionais utilizando imagens menores, com dimensões 512×512.

Bibliografia

[1] Robert G Bartle. *The elements of integration and Lebesgue measure*. John Wiley & Sons, 2014.

[2] Stephen Boyd and Jacob Mattingley. Branch and bound methods. *Notes for EE364b, Stanford University*, pages 2006–07, 2007.

[3] Antonin Chambolle and Thomas Pock. An introduction to continuous optimization for imaging. *Acta Numerica*, 25:161–319, 2016.

[4] Sira Ferradans, Nicolas Papadakis, Gabriel Peyré, and Jean-François Aujol. Regularized discrete optimal transport. *SIAM Journal on Imaging Sciences*, 7(3):1853–1882, 2014.

[5] Guy Gilboa and Stanley Osher. Nonlocal linear image regularization and supervised segmentation. *Multiscale Modeling & Simulation*, 6(2):595–630, 2007.

[6] Michael Greenspan and Mike Yurick. Approximate kd tree search for efficient icp. In *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.*, pages 442–448. IEEE, 2003.

[7] John V Guttag. *Introduction to computation and programming using Python*. Mit Press, 2013.

[8] Frederick S Hillier and Gerald J Lieberman. *Introdução à pesquisa operacional*. McGraw Hill Brasil, 2013.

[9] Shiguang Liu and Min Pei. Texture-aware emotional color transfer between images. *IEEE Access*, 6:31375–31386, 2018.

[10] Ludovic Métivier, Romain Brossier, Quentin Merigot, Édouard Oudet, and Jean Virieux. An optimal transport approach for seismic tomography: Application to 3d full waveform inversion. *Inverse Problems*, 32(11):115008, 2016.

[11] John E Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of applied optimization*, 1:65–77, 2002.

[12] Matthew J Saltzman. Coin-or: an open-source library for optimization. In *Programming languages and systems in computational economics and finance*, pages 3–32. Springer, 2002.

[13] Filippo Santambrogio. Optimal transport for applied mathematicians. *Birkäuser, NY*, 55(58-63):94, 2015.

[14] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[15] Cédric Villani. *Optimal transport: old and new*, volume 338. Springer, 2009.

[16] Cédric Villani. *Topics in optimal transportation*, volume 58. American Mathematical Soc., 2021.

[17] Yi Zhang, Tianhao Zhao, Zhipeng Mo, and Wenbo Li. A method of illumination effect transfer between images using color transfer and gradient fusion. In *2013 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, pages 1–6. IEEE, 2013.