

Quantização de imagens como um problema de otimização discreta

Matheus do Ó Santos Tiburcio
{mathh2899@gmail.com, matheusost@ic.ufrj.br}

11 de janeiro de 2025

Resumo

Este texto descreve o problema de quantização de cores. Inicialmente defino o problema e discuto sobre representação de cores RGB em dispositivos eletrônicos. Após isso introduzo a meta-heurística GRASP e comento de minha implementação. Termino demonstrando alguns resultados que obtive. Esse artigo junto do notebook podem ser encontrados em minha página pessoal do github: <https://github.com/Mth0/Color-Quantization>.

1 Introdução

O formato de representação de cores em aparelhos eletrônicos como computadores mais conhecido é o **RGB**. Neste formato cada pixel é representado por três valores inteiros que vão de 0 a 255. Cada um desses três valores está associado a um canal: vermelho(**Red**), verde (**Green**) e azul **Blue**. A cor de um pixel é determinada pela combinação dos três valores desses canais. Desse modo, por exemplo, a cor vermelha seria (255, 0, 0), verde (0, 255, 0) e azul (0, 0, 255).

Observe que, como se podem assumir 256 valores em cada canal, um total de $256^3 = 16.777.216$ cores possíveis podem ser descritas por esse formato. Isso é bem mais do que o número de cores que nós, humanos, somos capazes de discernir. Em muitos momentos podemos não necessitar de toda essa variedade e até preferimos nem possuir. A ideia da **quantização de cores**[10] é justamente reduzir essa variedade de cores na imagem, mas buscando manter o máximo de qualidade possível da imagem original. Essa necessidade surge em diversos contextos distintos, alguns são

1. **compressão de imagens.** Reduzir o tamanho de imagens muito grandes em aplicações pesadas se torna importante.
2. **Adaptadores de vídeo** que suportam bem menos cores que 256^3 . Nesse contexto não faz sentido manter todas as cores e seria desperdício de espaço.
3. Formatos como o **GIF** e até imagens em navegadores são limitados a um número bem menor de cores.
4. Imagens **PNG** são limitadas a 24-bits para representar cores, número inferior a 256^3 , mas esse número pode ser ainda mais reduzido.

Além disso, a quantização já ocorre naturalmente na passagem de imagens de câmeras fotográficas para sua digitalização. Mesmo o **RGB** não engloba todas as possíveis cores.

Sendo assim, como fazer uma boa quantização é um problema bem corriqueiro e bem estudado. Repare que a motivação para fazê-la tende a estar envolvida com reduzir espaço ocupado, discutiremos isso. Antes é interessante discutir como uma imagem é representada neste formato no computador.

Dada uma imagem de resolução $m \times n$ (m pixels por n pixels) sua representação matricial pode ser feita utilizando três matrizes $m \times n$. Como discutimos, temos que cada pixel possui três valores referentes aos três canais de cores do formato, portanto cada matriz dessas será referente a um canal diferente. Uma imagem completa então seria uma matriz de dimensão $3 \times m \times n$. Neste padrão, estão sendo armazenadas $3mn$ entradas no computador que, a depender do tamanho da imagem, pode ser bem grande. Uma solução seria fatorar cada uma das três matrizes, por exemplo, em uma matriz C de cores e uma matriz Id de índices. Se a imagem possui k cores, C seria de dimensão $3 \times k$ e Id uma matriz $m \times n$ com valores entre 0 e k , representando que cor cada pixel possui. Essa representação faz com que $3k + mn$ espaços sejam armazenados. Porém k pode assumir valores gigantescos, visto que há 256^3 possibilidades, então para k muito grande essa representação pode sofrer problemas. No geral buscaremos k pequenos e bem menores que 256^3 .

Para um exemplo numérico, suponha uma imagem 1920×1080 e $k = 256$. Na representação original, teríamos $3 \times 1920 \times 1080 = 6.220.800$ entradas e a reduzida usando k seria $3 \times 256 + 1920 \times 1080 = 2.074.368$ entradas. Uma redução de 3 vezes.

2 Problema de quantização

Quantizar uma imagem é fixar uma paleta de tamanho k e colorir toda a imagem somente com esta paleta. Note que essa definição é bem solta, k é livre e não discuto como escolhemos a paleta, nem como colorimos. Isso é proposital! E, de fato, existem muitos algoritmos para quantização. Definirei este problema de uma maneira mais precisa, mas ressalto que não é o único modo de definir ele. Começando pela função de custo.

Seja I a matriz $3 \times m \times n$ que representa a imagem original, queremos \hat{I} que minimize

$$f(\hat{I}) = \|I - \hat{I}\|_F^2 \quad (1)$$

Onde $\|\cdot\|_F$ é a norma de Frobenius cuja definição é

$$\|A_{m \times n}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2} \quad (2)$$

A definição não abrange matrizes com três índices, mas farei a norma de Frobenius em cada um dos três canais e somarei os resultados. Podemos ver esta norma como um modo de generalizar a norma de vetores. Informalmente, queremos reduzir a distância entre as duas matrizes. Note que essa função de custo é quadrática, portanto existe um único mínimo global que pode ser encontrado analiticamente. Ocorre que, para k livre, o mínimo é exatamente a imagem original e, portanto, não nos interessa. Pode haver a possibilidade de acertarmos exatamente quantas cores a imagem tem, obter $f(\hat{I}) = 0$ e ainda assim reduzir o espaço (caso $k << 256^3$), mas não possuímos essa informação.

Como o valor de k é crucial, seria mais justo formalizar a função de custo em função de k . Vamos definir uma função p_k de imagens para imagens. Mais precisamente, p_k mapeia cada pixel de uma imagem em valores de uma paleta P de k cores. O modo como definiremos que cor da paleta cada pixel deve receber será avaliando a distância euclidiana do pixel para cada cor na paleta. Lembre que pixels possuem três valores e podemos vê-los como pontos em \mathbb{R}^3 , a cor da paleta que o pixel receberá será a cor mais próxima, na paleta, deste pixel. Essa função dependerá então de uma imagem e da paleta e sua definição será

$$p_k(I, P)_{:ij} = \arg \min_l \|I_{:ij} - P_l\| \quad (3)$$

Aqui trate : em : ij como um acesso aos 3 canais da matriz. Ou seja, o pixel na posição ij receberá a cor P_l que esteja mais próxima dele. Reescrevendo a função de custo, agora em função da paleta, temos

$$f(P) = \|I - p_k(I, P)\|_F^2 \quad (4)$$

Deste modo a única quantidade desconhecida é P , a paleta.

Repare que esse problema não é trivial de se resolver. Na verdade, se a paleta possui k cores, existem

$$\binom{256^3}{k} = \frac{(256^3)!}{k!(256^3 - k)!}$$

paletas possíveis. Esse número para $k = 256$, um valor comum e utilizado em GIFs, por exemplo, já é enorme! Isso torna o problema NP-Completo se buscamos a solução exata e é um exemplo de otimização discreta. Meu trabalho tenta utilizar da meta-heurística **GRASP**[7][4][3][5] para obter uma solução aproximada deste problema.

Essencialmente o GRASP vai buscar no espaço de soluções a solução aproximada que tenha o menor valor na função de custo. Mas note que há duas funções de custo citadas, sendo assim você pode interpretar o espaço de busca de duas formas. Se pensar na função de custo da Equação 1, o espaço de soluções são imagens e buscar uma solução é transicionar de imagem em imagem buscando a que possua menor valor na função. Se pensar na função de custo da Equação 4, o espaço de soluções são paletas e a busca da solução é dada pela transição entre paletas procurando a que tenha menor valor na função.

3 Meta-heurística escolhida

A meta-heurística que decidi utilizar foi o GRASP. O GRASP é uma meta-heurística conhecida por empenhar bem em problemas de otimização discreta. É simples e facilmente paralelizável. Existem outras alternativas para otimização discreta, como o ACO (*Ant Colony Optimization*)[9][2], mas por possuir uma maior complexidade, optei pelo GRASP que já trouxe resultados bem satisfatórios.

O GRASP é uma meta-heurística de ***multi-start*** ou **múltiplos começos**. Recebe esse nome porque a cada iteração gera uma nova solução para o problema de maneira randomizada e gulosa. Após essa geração, efetua uma busca local a fim de melhorar essa solução gulosa gerada. Na busca local soluções vizinhas são visitadas e por "soluções vizinhas" considere soluções parecidas com a gulosa. Essa noção varia de aplicação para aplicação e detalharei melhor em meu caso. Esse processo se repete até que a condição de parada seja atingida. Sua estrutura é bem simples e pode ser vista no algoritmo abaixo. Nele uso o número máximo de iterações como critério de parada.

Algoritmo 1: GRASP para minimização e adaptação ao problema

Entrada: f , busca_gulosa, busca_local, max_iter
Saída: x^*

```
1  $x^* \leftarrow NULL$ 
2  $iter \leftarrow 1$ 
3 enquanto  $iter \leq max\_iter$ 
4    $x \leftarrow busca\_gulosa()$ 
5    $x \leftarrow busca\_local(x)$ 
6   se  $f(x) < f(x^*)$  ou  $x^* = NULL$  então
7      $x^* \leftarrow x$ 
8    $iter \leftarrow iter + 1$ 
9 retorno  $x^*$ 
```

Note então que é uma heurística de pouquíssimos parâmetros a princípio e sua complexidade em como construímos as buscas gulosa e local. No contexto deste problema, tanto a busca gulosa, quanto a local que escolhi recebem mais argumentos.

3.1 Busca gulosa

A busca gulosa que utilizei recebe dois argumentos:

1. A imagem original $3 \times m \times n$. É necessária para que a nova imagem gerada tenha as mesmas dimensões e para também que a paleta de cores seja propriamente selecionada;
2. O tamanho k da paleta.

Ambos os argumentos são fixados no início do GRASP e não se alteram durante toda a sua execução. A saída é

1. A nova imagem $3 \times m \times n$, repintada com a nova paleta;
2. A paleta. Que será utilizada na busca local.

Um pseudo código é mostrado abaixo.

Algoritmo 2: Busca gulosa

Entrada: I, k
Saída: X^*, P

```
1  $X^* \leftarrow NULL$ 
2  $P \leftarrow gera\_paleta(I, k)$ 
3 para cada pixel  $ij$  de  $I$ 
4    $X_{ij}^* \leftarrow p_k(I, P)_{:ij}$ 
5 retorno  $X^*, P$ 
```

A geração da paleta é feita de uma maneira não tão direta. Primeiro faço uma contagem de todas as cores que aparecem na imagem. Após isso utilizo o método da roleta, levando em conta o quanto cada cor aparece. Deste modo, cores mais frequentes possuem maior probabilidade de serem escolhidas. Para uma paleta de tamanho k são selecionadas k cores utilizando o método da roleta citado acima. Selecionar cores desta forma faz com que a busca gulosa, crucial para este algoritmo, alcance menos na escolha de cores e obtenha um bom pontapé inicial para a busca local, que é completamente baseada na solução gulosa. Aí vemos a importância de uma boa solução gulosa.

3.2 Busca local

A busca local explora um dado número de vizinhos da solução gulosa e armazena a melhor solução entre os vizinhos e a solução gulosa. Nesse meu contexto, vizinhos de uma solução são paletas com algumas cores trocadas. Deixo o número de cores trocadas como um argumento livre, mas nos testes procurei manter ele como exatamente $\frac{1}{4}$ do tamanho da paleta. Essa busca tem muitos argumentos, sendo eles

1. Função de custo. Necessária para avaliar potenciais melhorias na imagem final;
2. Matriz original da imagem. Necessária para a seleção de novas cores para as paletas vizinhas;
3. Solução gulosa. A matriz colorida com a paleta obtida da busca gulosa;
4. A paleta;
5. Número de cores a serem alteradas na paleta (c);
6. Número de vizinhos a serem visitados (N). Determina quantos vizinhos, paletas distintas, serão testadas.

Desses argumentos, a função de custo, a matriz original da imagem e o número de vizinhos são fixados antes de iniciar o GRASP, portanto os únicos argumentos são a solução gulosa, a paleta e o número de cores a serem alteradas na paleta. A saída é a melhor imagem.

Algoritmo 3: Busca local

Entrada: f, I, \hat{I}, P, c, N
Saída: X^*

```

1  $X^* \leftarrow \hat{I}$ 
2 para  $i$  de 1 até  $N$ 
3    $\hat{P} \leftarrow \text{troca_paleta}(I, P, c)$ 
4    $aux \leftarrow \text{troca_cores}(\hat{I}, P, \hat{P})$ 
5     Se  $f(aux) < f(X^*)$ 
6        $X^* \leftarrow aux$ 
7  retorno  $X^*$ 
```

A troca de paletas é feita da mesma forma feita na busca gulosa, seleciona novas cores distintas da paleta original via método da roleta. A troca de cores é direta: cores que não existem mais na paleta são substituídas pela nova cor que tomou sua posição. Faço isso ao invés de recolorir toda a imagem por questões de eficiência de código. Recolorir, de fato, parece o ideal a se fazer, mas os resultados finais não diferem tanto e, portanto, optei por fazer desta forma. O número de cores mudadas se altera ao decorrer do algoritmo, mas discutirei isso mais detalhadamente em outra seção.

4 Implementação e paralelização do algoritmo

Toda a implementação foi feita em Julia[1]. Utilizei algumas bibliotecas para meu auxílio, todas para auxiliar coisas como leitura de arquivo, plots e suporte a programação paralela. As que utilizei foram `ImageView`, `Plots`, `ImageCore`, `Distributed`, `FileIO`, `Images`, dentre outras.

Devido à uma iteração do GRASP ser completamente independente da outra, exceto a parte final em que verificamos se podemos atualizar o minimizador atual, é um algoritmo facilmente paralelizável e por este motivo implementei o algoritmo de maneira concorrente. Essencialmente, se utilizo p processos, cada processo executa uma iteração única e após o término das p execuções é feita uma comparação da melhor imagem obtida por cada processo e a melhor imagem atual. Após isso, são disparadas mais p iterações. Ou seja, o algoritmo é executado "em rajada", de p em p iterações. Minha implementação pode ser encontrada em minha página pessoal do github: <https://github.com/Mth0/Color-Quantization>.

5 Resultados

Tentei fazer experimentos com diferentes imagens de diferentes fontes: Desenhos, jogos, fotos reais, justamente para avaliar a capacidade do modelo de aprender uma boa paleta em casos distintos. As subseções seguintes mostram os resultados. Em todos os casos utilizei 12 processos.

5.1 Naruto - Time 7

O primeiro teste foi com uma imagem do mangá de Naruto[13] representada na Figura 1.



Figura 1: Imagem do mangá de Naruto. Obra Japonesa criada por Masashi Kishimoto.

Primeiramente foi feita uma análise na distribuição de cores da imagem. Podemos ver o resultado na Figura 2. Repare que muitas cores parecem semelhantes e seus valores RGB distoam pouco. Isso explica que, de fato, efetuar uma redução no número de cores não parece absurdo.

As Figuras 3 e 4 mostram os resultados de uma execução com $k = 64$, 60 iterações, 50 vizinhos explorados e 16 mudanças de cores e 12 processos. O tempo levado para esta execução foi de 116.169647 segundos e as dimensões da imagem são 640×805 pixels. Por fim, a última imagem na Figura 3 é a imagem original a critério de comparação.

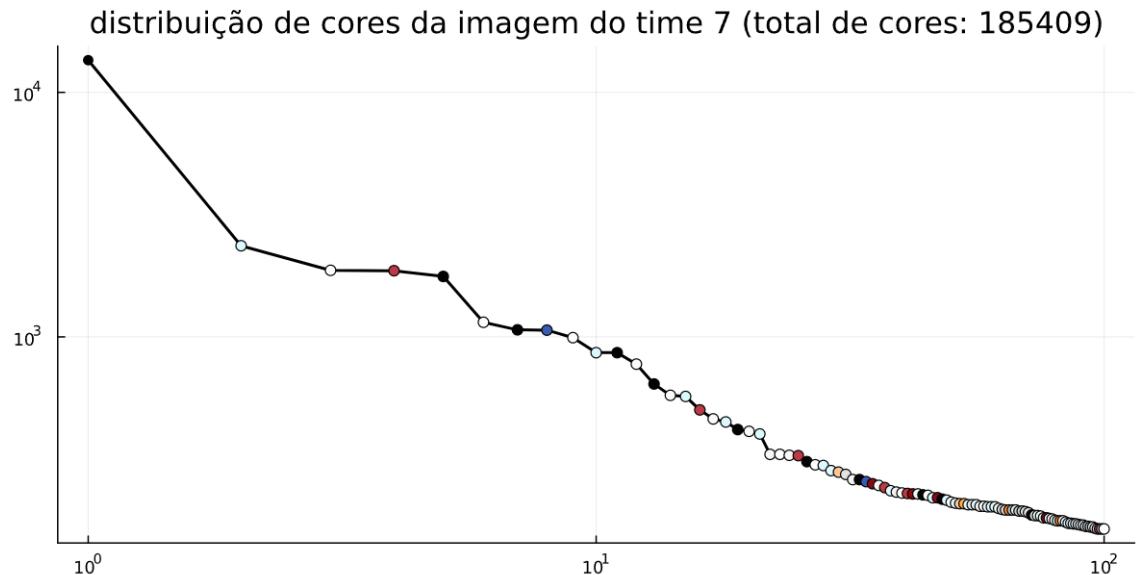
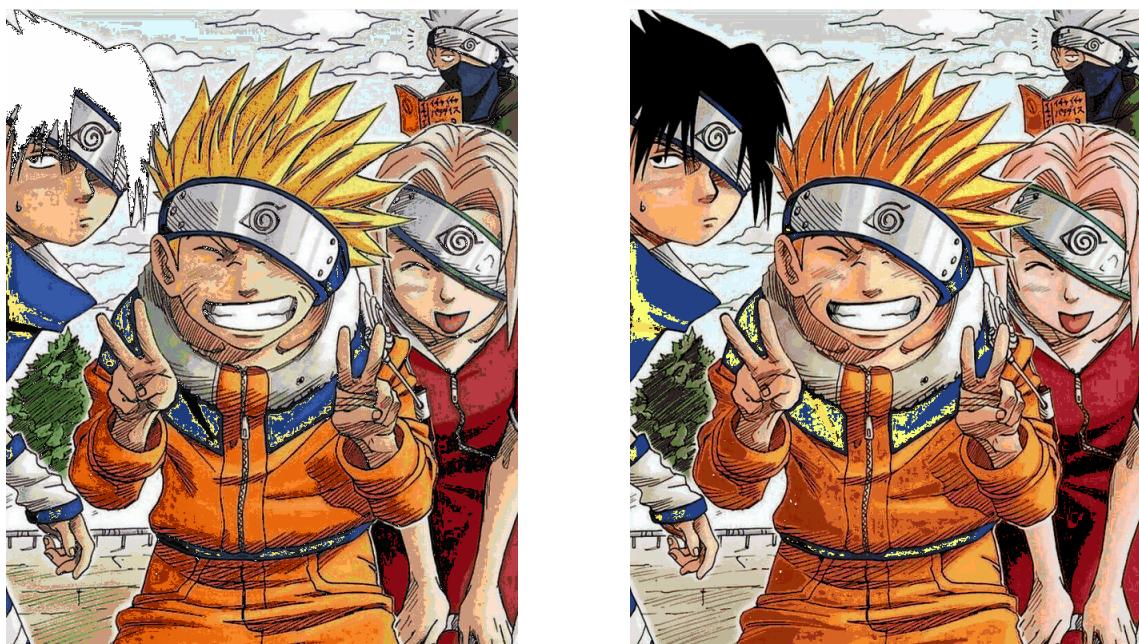


Figura 2: Distribuição de cores da imagem do time 7 de naruto. Cada ponto está colorido com a cor que representa. O eixo x é o índice da cor (1 para a que mais aparece, 2 para a segunda e assim em diante) e o eixo y , em escala log 10, é o número de aparições da cor.



(a) Imagem obtida pelo GRASP na iteração 1. A função de custo para essa imagem tem valor de 304.11997281033734.

(b) Imagem obtida pelo GRASP na iteração 3. A função de custo para essa imagem tem valor de 172.3178241231102.

Figura 3: Resultados da execução do GRASP na imagem da Figura 1. Considerei $k = 64$, 60 iterações, 50 vizinhos explorados e 16 mudanças de cores.



(a) Imagem obtida pelo GRASP na iteração 17. A função de custo para essa imagem tem valor de 166.67099177765115.



(b) Imagem obtida pelo GRASP na iteração 32. A função de custo para essa imagem tem valor de 156.07527891181076



(c) Imagem obtida pelo GRASP na iteração 44. A função de custo para essa imagem tem valor de 151.81322277634916.



(d) Imagem original.

Figura 4: Resultados da execução do GRASP na imagem da Figura 1. Considerei $k = 64$, 60 iterações, 50 vizinhos explorados e 16 mudanças de cores. A critérios de comparação, a Figura 4d é a imagem original.

5.2 Pokémon Yellow e a limitação do Game Boy Color

O teste seguinte foi com a capa do jogo eletrônico "Pokémon Yellow"[15] para GameBoy Color[11], um console portátil da Nintendo[17]. O GameBoy Color só suportava 56 cores simultâneas na tela, então testei uma quantização com $k = 56$. Assim como feito no caso do Naruto, uma análise na distribuição de cores também foi feita e pode ser vista na Figura 6. Note novamente a aparição de inúmeras cores bem semelhantes. Em especial vemos vários tons de amarelo e branco, o que faz pleno sentido dada a imagem original, representada na Figura 5.



Figura 5: Imagem de capa do jogo eletrônico "Pokémon Yellow" para o console portátil da Nintendo GameBoy Color.

As Figura 7 e 8 mostram os resultados de uma execução com $k = 56$, 180 iterações, 100 vizinhos explorados, 10 mudanças de cores e 12 processos. O tempo levado para esta execução foi de 20.076898 segundos e as dimensões da imagem são 275×183 pixels, consideravelmente menor que a anterior e por este motivo que consegui fazer mais iterações e manter um tempo bom de execução. Em especial a Figura 8 faz um comparativo do resultado final do GRASP e a imagem original.

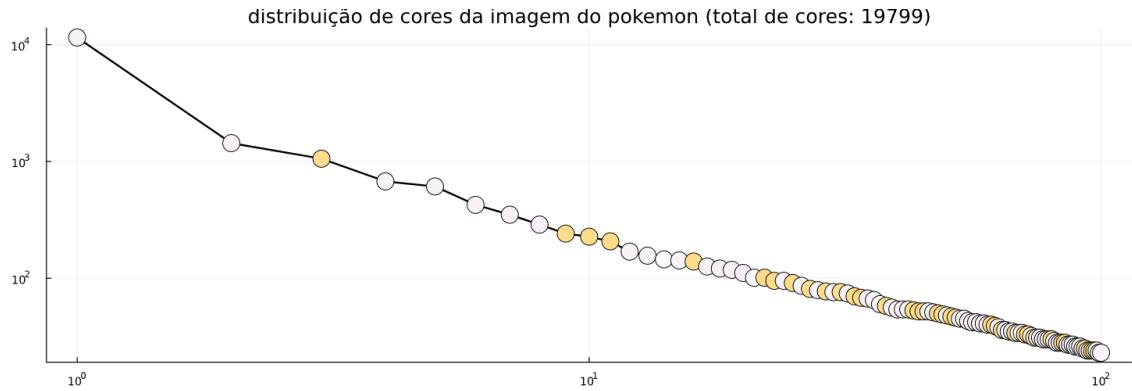
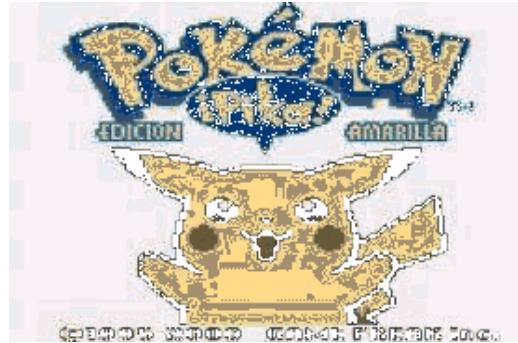


Figura 6: Distribuição de cores da imagem da capa do jogo "Pokémon Yellow". Cada ponto está colorido com a cor que representa. O eixo x é o índice da cor (1 para a que mais aparece, 2 para a segunda e assim em diante) e o eixo y , em escala log 10, é o número de aparições da cor.



(a) Imagem obtida pelo GRASP na iteração 1. A função de custo para essa imagem tem valor de 91.69039399323141.



(b) Imagem obtida pelo GRASP na iteração 2. A função de custo para essa imagem tem valor de 88.55345375236402.

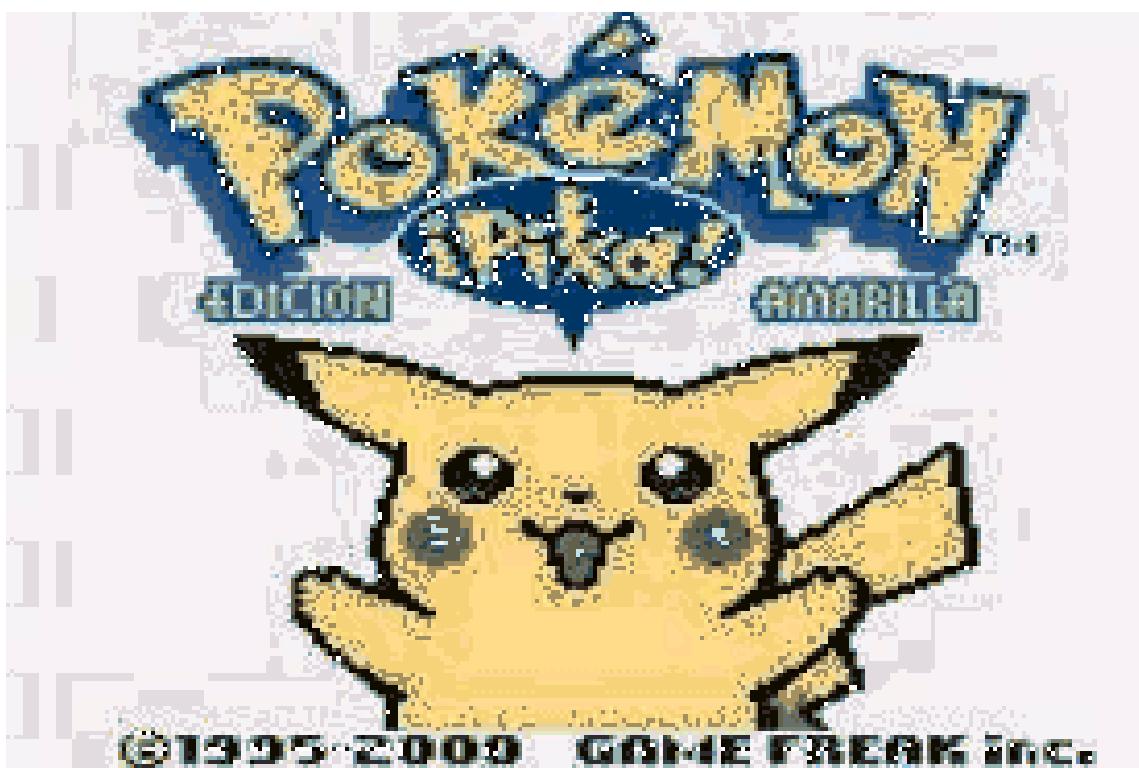


(c) Imagem obtida pelo GRASP na iteração 3. A função de custo para essa imagem tem valor de 75.33908012090257.



(d) Imagem obtida pelo GRASP na iteração 5. A função de custo para essa imagem tem valor de 50.17260618485235.

Figura 7: Resultados da execução do GRASP na imagem da Figura 5. Considerei $k = 56, 180$ iterações, 100 vizinhos explorados e 10 mudanças de cores.



(a) Imagem obtida pelo GRASP na iteração 16. A função de custo para essa imagem tem valor de 44.894054616909216.



(b) Imagem original.

Figura 8: Comparação da última atualização de melhor minimizador feita pelo GRASP e a imagem original. A Figura 8a é a última atualização obtida pelo GRASP e a Figura 8b é a imagem original.

5.3 Imagens reais

Os últimos dois exemplos são referentes à imagens reais (com pessoas e animais reais) e curiosamente foram os melhores resultados. O primeiro exemplo é referente à uma imagem, representada na Figura 10, em homenagem ao primeiro título da CONMEBOL Libertadores da América conquistado pelo clube de futebol carioca Botafogo em 2024. A Figura 9 mostra a distribuição de cores da imagem, na qual podemos reparar tons de preto dominando as primeiras posições. Para o teste do botafogo usei $k = 128$, 48 iterações, 50 vizinhos explorados, 32 mudanças de cores e 12 processos. As dimensões da imagem são 1024×1280 pixels. Devido ao grande tamanho da paleta e às dimensões da imagem, o tempo levado foi disparado o maior de todos os testes. A execução levou cerca de 412.149520 segundos e os resultados podem ser vistos na Figura 11. A Figura 12 traz uma comparação entre o resultado final obtido pelo GRASP e a imagem original. Note que obtemos uma imagem de boa qualidade já na segunda atualização.

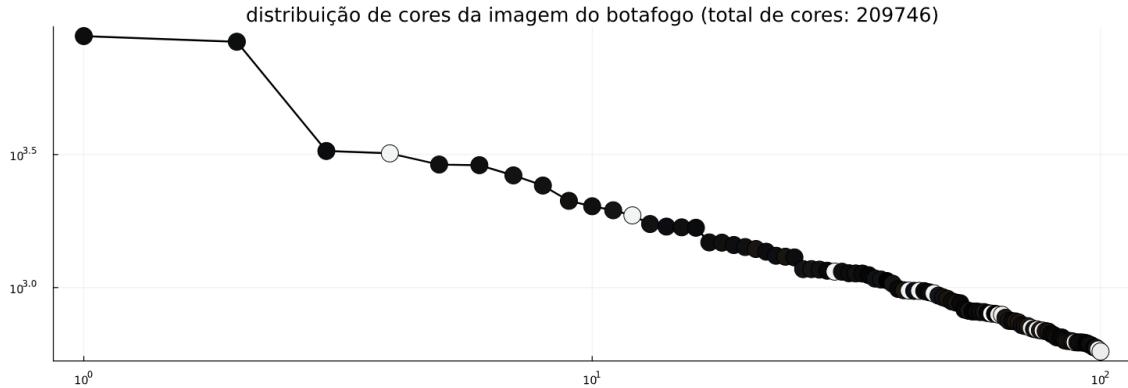


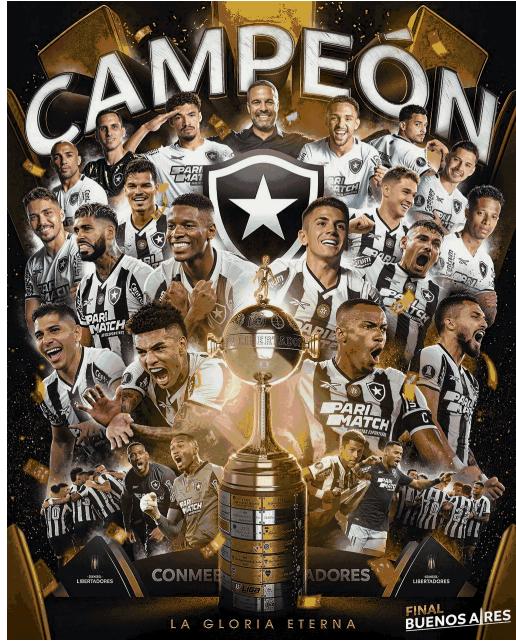
Figura 9: Distribuição de cores da imagem em homenagem ao primeiro título da CONMEBOL Libertadores da América conquistado pelo clube de futebol carioca Botafogo em 2024. Cada ponto está colorido com a cor que representa. O eixo x é o índice da cor (1 para a que mais aparece, 2 para a segunda e assim em diante) e o eixo y , em escala log 10, é o número de aparições da cor.



Figura 10: Imagem em homenagem ao primeiro título da CONMEBOL Libertadores da América conquistado pelo clube de futebol carioca Botafogo em 2024.



(a) Imagem obtida pelo GRASP na iteração 1. A função de custo para essa imagem tem valor de 232.1196690193037.



(b) Imagem obtida pelo GRASP na iteração 3. A função de custo para essa imagem tem valor de 128.686476653032.



(c) Imagem obtida pelo GRASP na iteração 13. A função de custo para essa imagem tem valor de 112.04712271448183.



(d) Imagem obtida pelo GRASP na iteração 18. A função de custo para essa imagem tem valor de 110.49522893359918.

Figura 11: Resultados da execução do GRASP na imagem da Figura 10. Considerei $k = 128$, 48 iterações, 50 vizinhos explorados, 32 mudanças de cores e 12 processos.



(a) Imagem obtida pelo GRASP na iteração 18. A função de custo para essa imagem tem valor de 110.49522893359918.



(b) Imagem original

Figura 12: Comparação da última atualização de melhor minimizador feita pelo GRASP e a imagem original. A Figura 12a é a última atualização obtida pelo GRASP e a Figura 12b é a imagem original.

O último exemplo, também real, é uma foto de dois gatos dormindo juntos. Para este teste com os gatos usei $k = 64$, 36 iterações, 50 vizinhos explorados, 16 mudanças de cores e 12 processos. As dimensões da imagem são 900×1600 pixels. Aqui o algoritmo performou bem e os avanços no valor do mínimo global são não tão perceptíveis. A Figura 14 e 15 mostram algumas atualizações de melhor minimizador obtidas pelo GRASP. Em especial, a Figura 15 mostra uma comparação do resultado final obtido pelo GRASP e a imagem original.

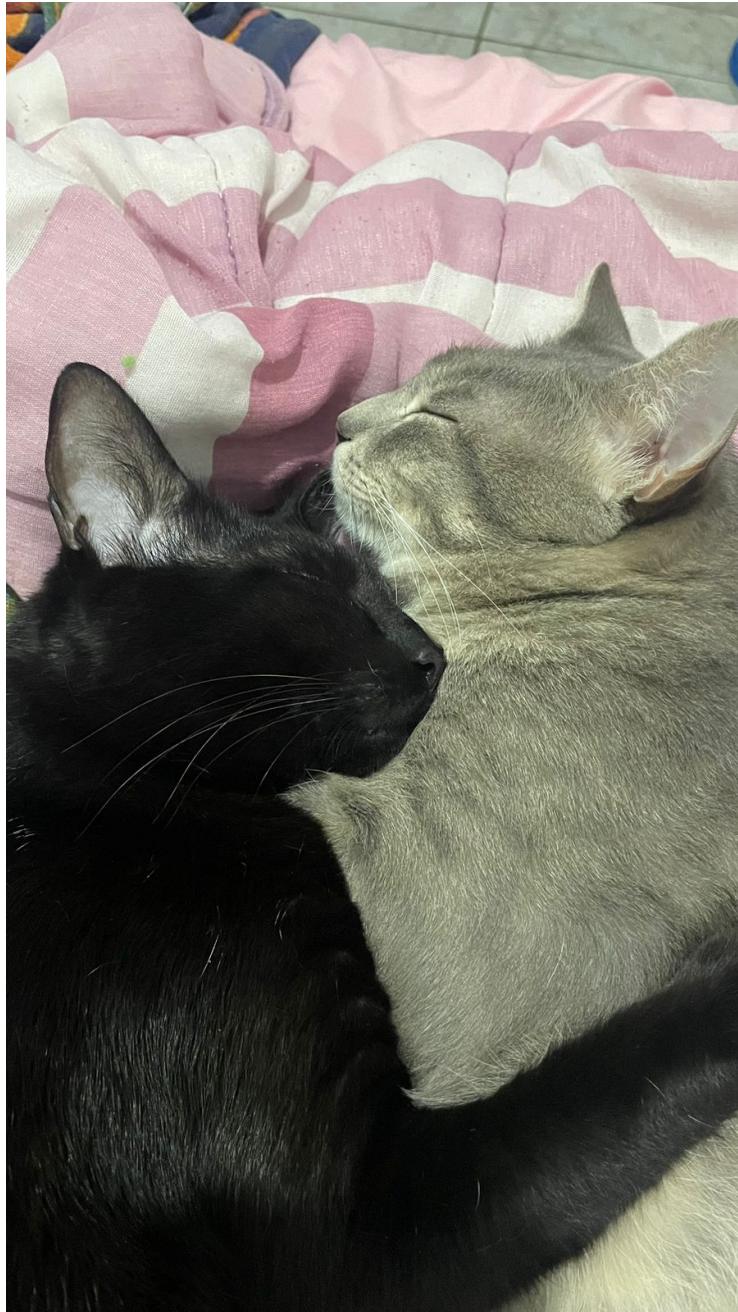


Figura 13: Imagem de dois gatos dormindo.



(a) Imagem obtida pelo GRASP na iteração 1. A função de custo para essa imagem tem valor de 402.30016037848895.



(b) Imagem obtida pelo GRASP na iteração 2. A função de custo para essa imagem tem valor de 287.02321052915016.



(c) Imagem obtida pelo GRASP na iteração 8. A função de custo para essa imagem tem valor de 105.64092409805978.



(d) Imagem obtida pelo GRASP na iteração 13. A função de custo para essa imagem tem valor de 100.31471208638781.

Figura 14: Resultados da execução do GRASP na imagem da Figura 13. Considerei $k = 64$, número de mudanças = 16 e número de vizinhos = 20. Foram feitas 24 iterações.



(a) Imagem obtida pelo GRASP na iteração 28. A função de custo para essa imagem tem valor de 89.23335571617696.



(b) Imagem original

Figura 15: Comparação da última atualização de melhor minimizador feita pelo GRASP e a imagem original. A Figura 15a é a última atualização obtida pelo GRASP e a Figura 15b é a imagem original.

6 Efeito platô

Como podemos notar, todos os exemplos atualizaram pouco o melhor minimizador e isso se deve um pouco pela natureza do GRASP. Como o GRASP é um algoritmo de múltiplos inícios, nenhuma memória de iterações passadas é armazenada exceto o melhor minimizador encontrado. Deste modo, o processo de construção de soluções é feito do ínicio sempre e não há garantias de um novo minimizador surgir e por esse fato o GRASP passa longo períodos com o mesmo melhor minimizador. Isso pode ser notado no gráfico de convergência média da Figura 16. Nessa figura executei 20 vezes o algoritmo GRASP na imagem da Figura 1. Todas as execuções utilizaram $k = 32$, 48 iterações, 20 vizinhos explorados, 8 mudanças de cores. Considere esse experimento para o resto desta seção.

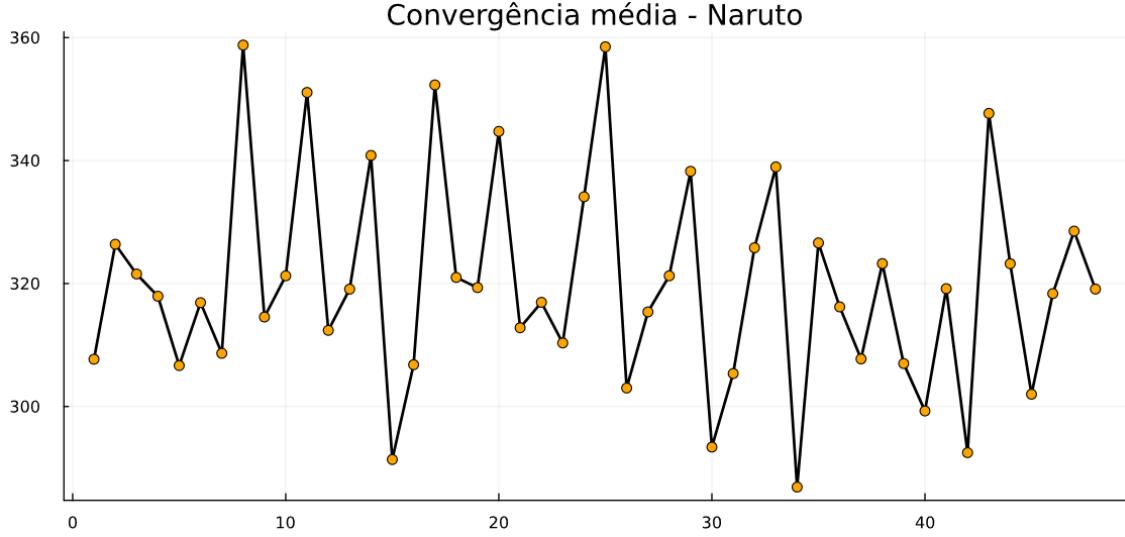


Figura 16: Gráfico de convergência média para 20 execuções do algoritmo GRASP para a imagem da Figura 1. Todas as execuções utilizaram $k = 32$, 48 iterações, 20 vizinhos explorados, 8 mudanças de cores. O eixo x representa as iterações do algoritmo e o eixo y o valor médio das 20 execuções nestas iterações.

Chamo esse problema de **efeito platô** em alusão ao grande tempo em que o algoritmo permanece com o mesmo valor de melhor minimizador. Isso pode ser visto nas 20 iterações individualmente na Figura 17, onde o valor do melhor minimizador de cada iteração é ilustrado para todas as execuções.

Para tentar amenizar este problema eu introduzi uma alteração dinâmica no valor do parâmetro de mudança de cores. Um argumento especial `threshold` é passado ao código e se refere ao número de iterações seguidas com o mesmo melhor minimizador que o GRASP tolerará. Caso esse número de iterações ultrapasse o valor de `threshold`, o número de mudança de cores se torna exatamente a metade do tamanho da paleta e o contador de iterações com o mesmo minimizador é zerado. Para as próximas ultrapassagens é somado $\frac{k}{10}$, k tamanho da paleta, ao valor de mudança atual. Esse incremento é feito até que o número de mudança de cores na paleta ultrapasse o tamanho da paleta ou uma atualização de melhor minimizador seja feita. Em ambos os casos o valor de mudança volta a seu valor original estipulado no ínicio do algoritmo.

Apesar de introduzir isso, não senti grandes diferenças, mas preservei esse comportamento no algoritmo. Para um gráfico de convergência média melhor, vamos considerar somente os melhores minimizadores de cada iteração. Computando o valor médio das 20 execuções destes minimizadores em cada uma das iterações chegamos ao gráfico da figura 18. O interessante é ver que, de fato, parece que estamos convergindo para um mínimo local.

Iterações individuais

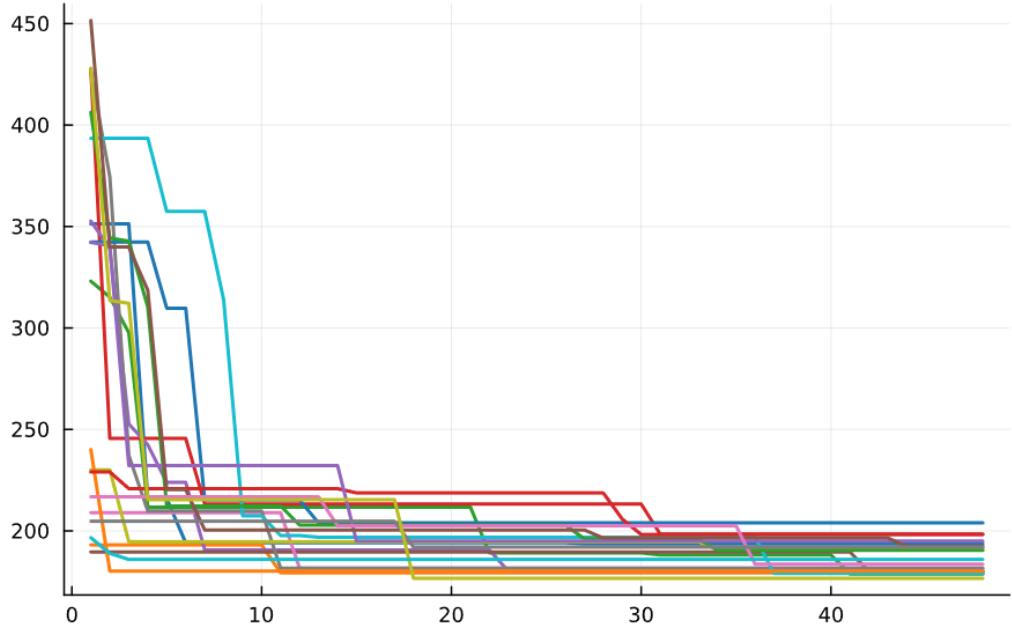


Figura 17: Gráfico representando o valor individual de cada uma das 20 execuções do GRASP em cada iteração. Todas as execuções utilizaram $k = 32$, 48 iterações, 20 vizinhos explorados, 8 mudanças de cores. O eixo x representa as iterações do algoritmo e o eixo y o valor em cada uma dessas iterações. Note o **efeito platô** ocorrendo em todas.

Convergência mínima média - Naruto

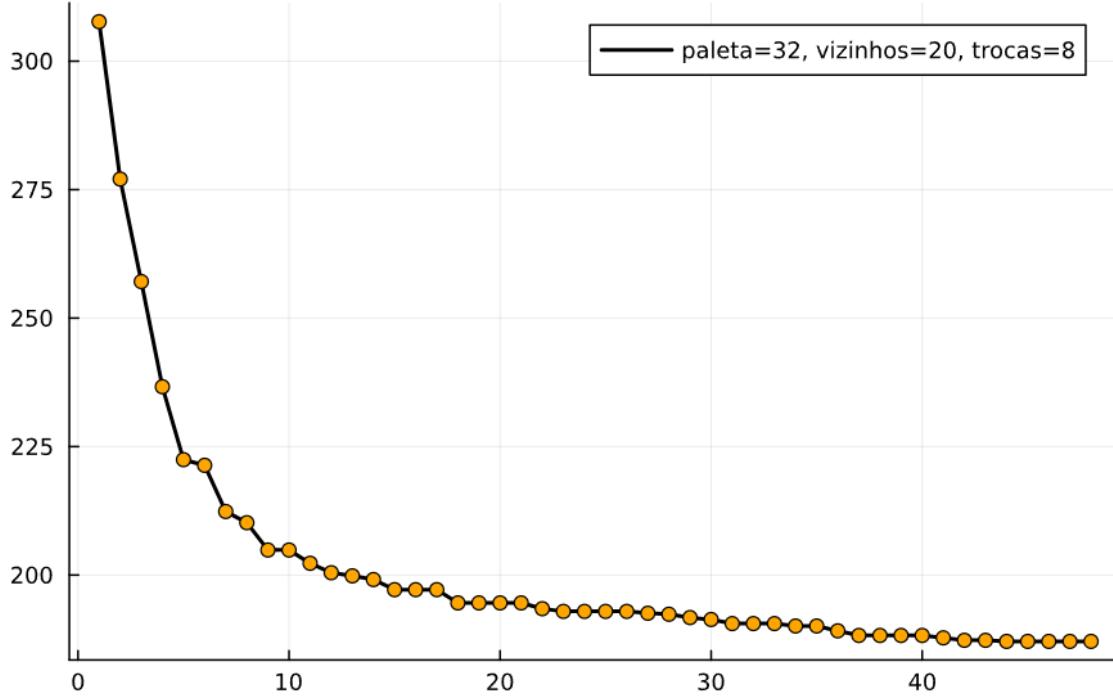


Figura 18: Gráfico de convergência média dos melhores minimizadores para 20 execuções do algoritmo GRASP para a imagem da Figura 1. Todas as execuções utilizaram $k = 32$, 48 iterações, 20 vizinhos explorados, 8 mudanças de cores. O eixo x representa as iterações do algoritmo e o eixo y o valor médio das melhores imagens encontradas por cada uma das 20 execuções em cada iteração.

7 Conclusão e outras implementações

O GRASP no geral se saiu muito bem nas imagens de teste e as execuções levaram em torno de poucos minutos. Como dito inicialmente, esse problema é bem conhecido e inúmeras outras formas de resolvê-lo existem. Uma delas é endereçar cores por blocos de pixels, técnicas de compressão de imagem costumam endereçar 2 ou 4 cores para blocos de pixels de tamanho 4×4 . Exemplos de algoritmos de compressão são BTC, CCC, S2TC e S3TC.

Outro algoritmo bem famoso é o *median cut algorithm*[8]. Nele os pixels são discretizados em um *bin*, em seguida o canal com maior variabilidade é selecionado e os pixels ordenados de acordo com ele. A mediana é tomada (um ou dois pixels) e o *bin* é cortado exatamente aí. Esse processo se repete até que se tenha o número de *bins* exatamente igual ao tamanho da paleta desejado.

Um modo mais moderno de quantizar é utilizando *octrees*, uma estrutura de dados de árvore em que cada nó tem 8 filhos. A ideia é ir dividindo o espaço \mathbb{R}^3 em seus oito octantes. Por último, as redes neurais **SOM** (self-organizing map)[16] também são usadas. Esta é uma técnica de aprendizado não-supervisionado de redução de dimensionalidade. A ideia é fazer essa redução preservando a topologia dos dados.

Além desses métodos existem outros mais. Infelizmente não comparei minha implementação do GRASP com esses outros métodos, mas certamente é algo que desejo fazer em algum momento. O tema é bem interessante e foi divertido implementar este algoritmo.

8 Trabalhos futuros

Apesar do bom desempenho do GRASP, algumas melhorias são possíveis. Listo algumas.

1. Reduzir a complexidade das funções de busca gulosa e local. No momento, a função de busca gulosa tem complexidade $O(kmn)$ para uma paleta de tamanho k e uma imagem de tamanho $3 \times m \times n$. Uma complexidade um pouco menor, mas acima de ordem quadrática ocorre na busca local também. No geral, uma iteração do GRASP nesse contexto é uma operação bem cara para imagens grandes. Uma ideia seria tentar, de maneira esperta, implementar de outro modo essas funções.
2. Reduzir o espaço utilizado durante a coloração das imagens. Na função de busca local, para uma imagem $3 \times m \times n$ outras duas imagens de mesma dimensão são mantidas a todo momento, fora a matriz de índices $m \times n$, importante para tornar a busca local um pouco mais rápida, fazendo com que somente os pixels cuja cor não está mais na paleta sejam recoloridos ao invés de toda a imagem. Gostaria de reduzir essa ocupação toda de espaço.

Referências

- [1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.
- [2] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1:28–39, 12 2006.
- [3] Thomas Feo and Mauricio Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 03 1995.
- [4] Thomas A Feo and Mauricio G.C Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67–71, 1989.
- [5] Mauricio G. C. Resende and Celso C. Ribeiro. *Greedy Randomized Adaptive Search Procedures*, pages 219–249. Springer US, Boston, MA, 2003.
- [6] Wikipedia contributors. Color histogram — Wikipedia, the free encyclopedia, 2023. [Online; accessed 4-November-2024].
- [7] Wikipedia contributors. Greedy randomized adaptive search procedure — Wikipedia, the free encyclopedia, 2023.
- [8] Wikipedia contributors. Median cut — Wikipedia, the free encyclopedia, 2023.
- [9] Wikipedia contributors. Ant colony optimization algorithms — Wikipedia, the free encyclopedia, 2024.
- [10] Wikipedia contributors. Color quantization — Wikipedia, the free encyclopedia, 2024.
- [11] Wikipedia contributors. Game boy color — Wikipedia, the free encyclopedia, 2024.
- [12] Wikipedia contributors. Indexed color — Wikipedia, the free encyclopedia, 2024.
- [13] Wikipedia contributors. Naruto — Wikipedia, the free encyclopedia, 2024.
- [14] Wikipedia contributors. Png — Wikipedia, the free encyclopedia, 2024.
- [15] Wikipedia contributors. Pokémon red, blue, and yellow — Wikipedia, the free encyclopedia, 2024.
- [16] Wikipedia contributors. Self-organizing map — Wikipedia, the free encyclopedia, 2024.
- [17] Wikipédia. Nintendo — wikipédia, a encyclopédia livre, 2024.