

Quantização de imagens como um problema de otimização discreta

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

10 de novembro de 2024

Resumo

Este texto descreve o problema de quantização de cores. Inicialmente defino o problema e discuto sobre representação de cores 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. Esses três valores estão associados a 3 canais: 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 a manter em boa qualidade. Essa necessidade surge em diversos contextos distintos, alguns são

1. **Compreensão 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), como discutimos cada pixel possui três valores referentes aos três canais de cores do formato, portanto uma imagem pode ser representada por três matrizes de dimensão $m \times n$. 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, 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. Pode 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 imagem 0 em f 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 imagens para imagens. Mais precisamente, mapeia cada pixel de uma imagem em valores de uma paleta P de k cores. O modo como definiremos que cor da paleta que 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

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

Ou seja, o pixel na posição ij receberá a cor P_k 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 a menor imagem 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 imagem 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 jaz na cara da busca 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 a paleta de cores ser propriamente selecionada;
2. O tamanho k da paleta.

Na verdade já passo para o GRASP essa função com ambos os argumentos fixados, então o GRASP não altera nada nela durante a 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 selecionando k pixels aleatoriamente da imagem com probabilidades ponderadas, as cores mais recorrentes possuem maior probabilidade de serem escolhidas. E a coloração é feita como discutido anteriormente, tomando a cor mais próxima de cada pixel.

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. Necessária para a seleção de novas cores para as paletas vizinhas;
3. Solução gulosa. A matriz colorida com a paleta obtida da na busca gulosa;
4. A paleta;
5. Número de cores a serem alteradas na paleta;
6. Número de vizinhos a serem visitados. Determina quantos vizinhos, paletas distintas, serão testadas.

Desses argumentos, a função de custo, a matriz original, o número de cores para serem alteradas e o número de vizinhos são fixados antes de iniciar o GRASP, portanto os únicos argumentos são a solução gulosa e a 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 selecionando posições aleatórias distintas da paleta original e trocando seu valor por um novo pixel aleatório da imagem. A troca de cores é direta: cores que não existem mais na paleta são substituídas pela nova cor que tomou sua posição.

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` e `Images`. A implementação foi feita de maneira concorrente.

Devido à uma iteração do GRASP ser completamente independente da outra, exceto a parte final de comparar a valor da função da imagem obtida com a melhor imagem obtida até o momento, é um algoritmo facilmente paralelizável. Essencialmente, se utilizo p processos, cada processo executava uma iteração única e após o término das p execuções fazia uma comparação das 12 imagens obtidas e a melhor imagem atual. Após isso, disparava mais p iterações. O algoritmo era executado 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. 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.

Executei o algoritmo algumas vezes com a configuração de $k = 64$, número de mudanças = 16 e número de vizinho = 20. A Figura 2 mostra o resultado.

Note que o chute inicial já é bem bom e o algoritmo mudou a melhor imagem somente uma vez, na segunda iteração, apesar de terem sido feitas 24 iterações. A Figura 3 compara os resultados.



(a) Imagem obtida pelo GRASP na iteração 1. A função de custo para essa imagem tem valor de 285.13462842011273.



(b) Imagem obtida pelo GRASP na iteração 2. A função de custo para essa imagem tem valor de 167.39929160441153.

Figura 2: Resultados da execução do GRASP na imagem da Figura 1. Considerei $k = 64$, número de mudanças = 16 e número de vizinhos = 20. Foram feitas 24 iterações.



(a) Melhor imagem obtida pelo GRASP com $k = 64$ e 24 iterações. A função de custo para essa imagem tem valor de 167.39929160441153.



(b) Imagem original.

Figura 3: Comparação de resultados do GRASP e da imagem original. A Figura 3a é a imagem obtida pelo GRASP com 64 cores e a Figura 3b é a imagem original.

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 $k = 56$. Para os outros parâmetros defini número de mudanças = 10, número de vizinhos = 20 e 60 iterações. A imagem original é mostrada na Figura 4, os resultados do GRASP aparecem na Figura 5 e a Figura 6 mostra a comparação da imagem original com dois resultados do GRASP.



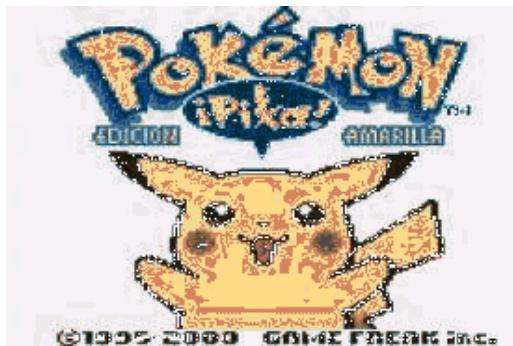
Figura 4: Imagem de capa do jogo eletrônico ”Pokémon Yellow” para o console portátil da Nintendo GameBoy Color.



(a) Imagem obtida pelo GRASP na iteração 1. A função de custo para essa imagem tem valor de 80.99121429846922.



(b) Imagem obtida pelo GRASP na iteração 2. A função de custo para essa imagem tem valor de 58.242257192489134.



(c) Imagem obtida pelo GRASP na iteração 18. A função de custo para essa imagem tem valor de 57.30290990234069.



(d) Imagem obtida pelo GRASP na iteração 40. A função de custo para essa imagem tem valor de 56.67720005228859.

Figura 5: Resultados da execução do GRASP na imagem da Figura 4. Considerei $k = 56$, número de mudanças = 10 e número de vizinhos = 20. Foram feitas 60 iterações.



(a) Imagem da iteração 40 pelo GRASP com $k = 56$ e 60 iterações. A função de custo para essa imagem tem valor de 56.67720005228859.



(b) Imagem original.

Figura 6: Comparação de resultados do GRASP e da imagem original. A Figura 6a é o melhor global obtido pelo GRASP com 56 cores e a Figura 6b é a imagem original.



Figura 7: Imagem de dois gatos dormindo.

O último exemplo é uma foto de dois gatos dormindo juntos. Aqui o algoritmo performou bem e os avanços no valor do mínimo global são não tão perceptíveis. A Figura 8 e 9 mostram as imagens obtidas pelo GRASP com 64 cores durante 24 iterações. A Figura 10c mostra uma comparação da primeira imagem obtida pelo GRASP, a última e a imagem original. Para os outros parâmetros defini número de mudanças = 16 e número de vizinhos = 20.



(a) Imagem obtida pelo GRASP na iteração 1.
A função de custo para essa imagem tem valor de 219.01116331697574.



(b) Imagem obtida pelo GRASP na iteração 2.
A função de custo para essa imagem tem valor de 217.1276495016827.



(c) Imagem obtida pelo GRASP na iteração 4.
A função de custo para essa imagem tem valor de 163.87169768565428.

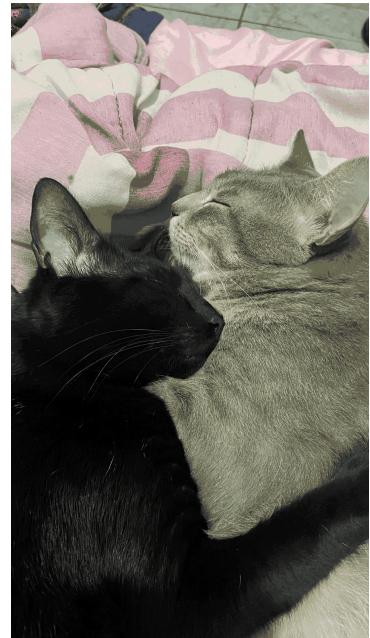


(d) Imagem obtida pelo GRASP na iteração 9.
A função de custo para essa imagem tem valor de 136.58205319455345.

Figura 8: Resultados da execução do GRASP na imagem da Figura 7. Considerei $k = 64$, número de mudanças = 16 e número de vizinhos = 20. Foram feitas 24 iterações.



(a) Imagem obtida pelo GRASP na iteração 14.
A função de custo para essa imagem tem valor
de 102.37296378415313.



(b) Imagem obtida pelo GRASP na iteração 16.
A função de custo para essa imagem tem valor
de 99.81685266203975.

Figura 9: Resultados da execução do GRASP na imagem da Figura 7. Considerei $k = 64$, número de mudanças = 16 e número de vizinhos = 20. Foram feitas 24 iterações. Repare na pouco diferença das imagens atualizadas pelo GRASP.



(a) Primeira imagem, da iteração 1, obtida pelo GRASP com $k = 64$ e 24 iterações. A função de custo para essa imagem tem valor de 219.01116331697574.



(b) Última imagem, da iteração 16, obtida pelo GRASP com $k = 64$ e 24 iterações. A função de custo para essa imagem tem valor de 99.81685266203975.



(c) Imagem original.

Figura 10: Comparação de resultados do GRASP e da imagem original. A Figura 10a é a primeira imagem obtida pelo GRASP com 64 cores, a Figura 10b é a última imagem obtida pelo GRASP com 64 cores e a Figura 10c é a imagem original.

6 Conclusão e outras implementações

O GRASP no geral se saiu muito bem nas imagens de teste e as execuções, no geral, 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 compreensão de imagem costumam endereçar 2 ou 4 cores para blocos 4×4 de pixels. Exemplos de algoritmos de compreensão 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, a 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 com o GRASP com esses outros métodos, mas certamente é algo que desejo fazer em algum momento. O tema é bem interessante e foi divertido implementar esse algoritmo.

7 Trabalhos futuros

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

1. Atualmente pondero as probabilidades de cores a serem escolhidas na busca gulosa levando em conta o quão frequente a cor é. O algoritmo de GRASP é bem sensível à boas soluções gulosas, então tornar mais esperta essa determinação de soluções nessa etapa me soa importante. Para isso poderia usar algoritmos como o *median cut* que citei na conclusão ou fazer alguma análise em cima da distribuição de cores da imagem[6] em que o GRASP está tentando quantizar. Acredito que isso não só melhoraria os resultados em um âmbito geral, mas aceleraria a convergência também.
2. 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.
3. Reduzir o espaço utilizado durante a coloração das imagens. Na função de busca local, para uma imagem $3 \times m \times n$ outras três 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 enclopédia livre, 2024.