Alquimia Digital
Trabalho 1 – Programação de Baixo Nivel

Fernanda Franceschini, Luana Sostisso

Escola Politécnica— PUCRS

27 de abril de 2024

Resumo

Este trabalho visa apresentar uma solução para o problema proposto na disciplina de Programação de Baixo Nível no terceiro semestre. O problema consiste em desenvolver um programa que execute a transmutação de imagens, reordenando os pixels de uma imagem de origem para produzir uma imagem desejada, com o objetivo de criar uma imagem que se assemelhe o máximo possível à segunda imagem fornecida. Através dessa narrativa, destaca-se a implementação do programa, mostrando as habilidades em programação de baixo nível e algoritmos de otimização. Além disso, foi realizado um teste de caso, a fim de garantir a eficácia do código.

1. Introdução

A transmutação de imagens é um problema que envolve a reorganização dos pixels de uma imagem de origem para criar uma nova imagem que assemelhe o máximo possível a uma imagem desejada. Inspirado na antiga prática alquímica de transformar elementos, este problema busca manipular digitalmente representações visuais, como cores, durante o processo de reordenação dos pixels para alcançar o resultado desejado. Esse processo envolve o mapeamento/manipulação dos pixels da imagem de origem de modo a reproduzir a aparência da imagem desejada.

Nesta apresentação detalhada, examinaremos os desafios inerentes à transmutação de imagens, os conceitos fundamentais envolvidos, as abordagens algorítmicas comumente empregadas para resolver esse problema e os resultados.

2. Processo de Solução

Para resolver o problema apresentado nesse relatório foi utilizada uma técnica de refinamentos sucessivos, isto é, a partir de uma tentativa inicial, refina-se o resultado até que determinada condição seja satisfeita.

2.1 Conceitos Fundamentais para realizar o algoritmo:

- Compreensão de como as imagens são representadas digitalmente através de matrizes de pixels, onde cada pixel possui informações sobre suas componentes de cor (por exemplo, RGB).
- Implementação de técnicas para comparar as cores dos pixels e determinar se uma troca entre pixels resultará em uma melhoria visual.
- Desenvolvimento de critérios para avaliar a qualidade dos resultados obtidos. Isso pode envolver a comparação visual entre a imagem resultante e a imagem desejada, bem como métricas objetivas de similaridade de cores.
- Estabelecer mecanismo para controlar o processo de refinamento, incluindo critérios de parada quando atingir um número máximo de iterações ou não obter melhorias significativas após um número específico de tentativas.
- Utilização de algoritmos de otimização para encontrar soluções progressivamente melhores.

2.2 Passo a Passo do código:

Inicialmente, para resolver o problema em um programa em C, é essencial representar as imagens, onde cada pixel é caracterizado por uma combinação de valores de vermelho, verde e azul. Essa representação é alcançada utilizando uma estrutura denominada “RGBpixel”, que é composta por três componentes de cor: “r” para vermelho, “g” para verde e “b” para azul. Adicionalmente, a estrutura “Img” proporciona uma forma organizada de manipular imagens, permitindo o acesso às suas propriedades, como largura e altura, bem como aos pixels individuais que compõem a imagem.

```
typedef struct
{
    unsigned char r, g, b;
} RGBpixel;

// Uma imagem RGBpixel
typedef struct
{
    int width, height;
    RGBpixel *pixels;
} Img;
```

Adicionalmente, as medidas da imagem de saída são configuradas para serem iguais às da imagem desejada e, também, a imagem de origem seja copiada para a imagem de saída, garantindo que todos os pixels a serem utilizados já estejam presentes.

```
pic[SAIDA].width = pic[DESEJ].width;
pic[SAIDA].height = pic[DESEJ].height;
pic[SAIDA].pixels = malloc(pic[DESEJ].width * pic[DESEJ].height * 3);


unsigned int tam = pic[ORIGEM].width * pic[ORIGEM].height;

memcpy(pic[SAIDA].pixels, pic[ORIGEM].pixels, sizeof(RGBpixel) * tam);
```

Conforme demonstrado no método anterior, a função “malloc” é usada para alocar espaço na memória para armazenar os pixels da imagem de saída. Para determinar quanto espaço precisamos, multiplicamos a largura pela altura da imagem desejada. Como cada pixel possui três componentes de cor (RGB), multiplicamos esse resultado por 3 para garantir espaço suficiente para todas as cores dos pixels.

Já a função “memcpy” realiza uma cópia byte a byte dos dados de uma área de memória para outra. Os parâmetros passados pelo método são: o array de pixels da imagem de saída, o array de pixels da imagem de origem e o tamanho da área a ser copiado em bytes (calcula o tamanho de um pixel “sizeof (RGBpixel)” multiplicado pelo número total de pixels na imagem “tam”).

Posteriormente, é necessário realizar trocas entre dois pixels da imagem de saída de forma iterativa, em que cada troca é avaliada para determinar se produz um resultado visualmente melhor que a configuração atual.

```
unsigned int counter = 0;
    while (counter < 5000) {
        unsigned long randomA = genrand64_int64() % (tam);
        unsigned long randomB = genrand64_int64() % (tam);

        RGBpixel pixelDesejadoA = pic[DESEJ].pixels[randomA];
        RGBpixel pixelDesejadoB = pic[DESEJ].pixels[randomB];
```

```
        RGBpixel pixelCorA = pic[SAIDA].pixels[randomA];
        RGBpixel pixelCorB = pic[SAIDA].pixels[randomB];

        double distPCA_PDA = euclidean_distance (pixelDesejadoA, pixelCorA);

        double distPCA_PDB = euclidean_distance (pixelDesejadoB, pixelCorA);

        double distPCB_PDA = euclidean_distance (pixelDesejadoA, pixelCorB);

        if(distPCA_PDB<distPCB_PDA && distPCB_PDA<distPCA_PDA){
            pic[SAIDA].pixels[randomA] = pixelCorB;
            pic[SAIDA].pixels[randomB] = pixelCorA;
            counter = 0;
        }
        else{
            counter++;
        }
    }
```

Esse trecho de código implementa um algoritmo de troca de pixels na imagem de saída (pic[SAIDA].pixels) com base na comparação das distâncias euclidianas entre pixels desejados (da imagem desejada) e pixels atuais (da imagem de saída). O primeiro passo é gerado dois índices aleatórios matriz “genrand64_int64() % (tam)”, que representa a posição dos pixels na imagem desejada e na imagem de saída.

Logo em seguida, é calculado as distâncias euclidianas entre os pixels da imagem desejada (“pixelDesejadoA e “pixelDesejadoB”) e os pixels correspondentes na imagem de saída (“pixelCorA” e “pixelCorB”). Com base nessas distâncias, é aplicado o critério de aceitação. Isso envolve comparar as distâncias para determinar se a troca de pixels resultará em uma melhoria na semelhança entre a imagem de saída e a imagem desejada. Para isso, o bloco “if” verifica se a distância do pixel da imagem desejada A com o pixel da imagem de saída B é menor do que o que está posicionado originalmente na imagem de saída e se a distância do pixel da imagem desejada B com o pixel da imagem de saída A for menor do que o que está posicionado orginalmente na imagem de saída. Caso essa premissa seja verdadeira o algoritmo troca os pixels da imagem de saída.

Além disso, neste mesmo trecho de código, é implementada a condição de parada, que determina quando o processo de refinamento deve ser encerrado. Essa condição é baseada na ausência de trocas vantajosas. Quando não há mais trocas vantajosas, o loop "while" é encerrado. Para controlar isso, uma variável denominada "counter" é utilizada para contar quantas vezes o algoritmo é executado sem ocorrer uma troca de pixel, para este caso foi determinado um valor de 5000 vezes. Cada vez que uma troca é realizada, o contador é reiniciado para zero, caso contrário é incrementado +1 no valor da variável. Essa abordagem visa evitar que o algoritmo gaste um tempo excessivo tentando encontrar melhorias marginais.

2.3 Testes

Para testar a eficácia do código, dois testes foram realizados. O primeiro é o método valida()

```
    void valida()
    {
        int ok = 1;
        int size = pic[ORIGEM].width * pic[ORIGEM].height;

        RGBpixel *aux1 = malloc(size * 3);
        RGBpixel *aux2 = malloc(size * 3);

        memcpy(aux1, pic[ORIGEM].pixels, size * 3);
        memcpy(aux2, pic[SAIDA].pixels, size * 3);
```

```
            for (int i = 0; i < 8; i++)
                printf("[%d %d %d] ", aux1[i].r, aux1[i].g, aux1[i].b);
            printf("\n");
            for (int i = 0; i < 8; i++)
                printf("[%d %d %d] ", aux2[i].r, aux2[i].g, aux2[i].b);
            printf("\n");
            printf("Validando...\n");

            qsort(aux1, size, sizeof(RGBpixel), cmp);
            qsort(aux2, size, sizeof(RGBpixel), cmp);

            for (int i = 0; i < 8; i++)
                printf("[%d %d %d] ", aux1[i].r, aux1[i].g, aux1[i].b);
            printf("\n");
            for (int i = 0; i < 8; i++)
                printf("[%d %d %d] ", aux2[i].r, aux2[i].g, aux2[i].b);
            printf("\n");
            for (int i = 0; i < size; i++)
            {
                if (aux1[i].r != aux2[i].r ||
                    aux1[i].g != aux2[i].g ||
                    aux1[i].b != aux2[i].b)
                {

                    printf("* INVÁLIDO na posição %d *: %02X %02X %02X -> %02X %02X
%02X\n",
                           i, aux1[i].r, aux1[i].g, aux1[i].b, aux2[i].r, aux2[i].g,
aux2[i].b);
                    ok = 0;
                    break;
                }
            }

            free(aux1);
            free(aux2);
            if (ok)
                printf(">>>> TRANSFORMAÇÃO VÁLIDA <<<<<\n");
        }
```

Esta função é responsável por validar a transformação aplicada às imagens. Ela aloca memória para duas arrays, aux1 e aux2, e copia os pixels originais das imagens de origem e de saída para essas arrays. Em seguida, ordena os pixels em ambas as arrays. Após a ordenação, verifica se os pixels em ambas as arrays são idênticos. Se algum pixel for diferente, indica que a transformação foi aplicada incorretamente, isto é, os pixels, em vez de serem mapeados/trocados de lugar, sofrerão alterações.

A função valida() chama o método cmp()**,** responsável pela comparação usada pelo algoritmo qsort() para ordenar os pixels. Ela compara os valores de vermelho, verde e azul de dois pixels e determina a ordem deles para a ordenação.

```
        int cmp(const void *elem1, const void *elem2)
        {
            RGBpixel *ptr1 = (RGBpixel *)elem1;
            RGBpixel *ptr2 = (RGBpixel *)elem2;
            unsigned char r1 = ptr1->r;
            unsigned char r2 = ptr2->r;
            unsigned char g1 = ptr1->g;
            unsigned char g2 = ptr2->g;
            unsigned char b1 = ptr1->b;
            unsigned char b2 = ptr2->b;
            int r = 0;
            if (r1 < r2)
                r = -1;
            else if (r1 > r2)
```

```
            r = 1;
        else if (g1 < g2)
            r = -1;
        else if (g1 > g2)
            r = 1;
        else if (b1 < b2)
            r = -1;
        else if (b1 > b2)
            r = 1;
        return r;
    }
```

Outro teste para garantir que o algoritmo está produzindo o desempenho desejados de maneira precisa e rápida, é realizar testes repetidos com várias imagens diferentes. Isso permite verificar se o algoritmo produz consistentemente os resultados desejados em diferentes situações e condições.

```
// PAR DE IMAGENS DADO <./alchemy ./images/mona.jpg ./images/relogio.jpg>
// PRIMEIRO PAR DE IMAGENS <./alchemy ./images/dog.jpg ./images/cat.jpg>
// SEGUNDO PAR DE IMAGENS <./alchemy ./images/preguica.jpg ./images/girafa.jpg>
// TERCEIRO PAR DE IMAGENS <./alchemy ./images/bolsa.jpg ./images/flor.jpg>
// QUARTO PAR DE IMAGENS <./alchemy ./images/cha_frutas.jpg ./images/cha_gengibre.jpg>
```

Exemplos:

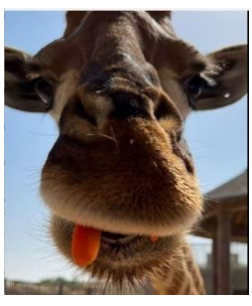
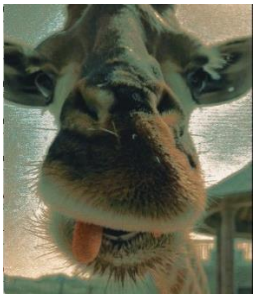
ORIGEM DESTINO SAIDA



ORIGEM DESTINO SAIDA



ORIGEM DESTINO SAIDA

3. Conclusão e Desafios

A conclusão deste trabalho demonstra que, por meio da implementação de um algoritmo de otimização eficaz, foi possível alcançar resultados satisfatórios na transmutação de imagens. Através do processo de refinamento sucessivo, baseado na comparação e troca de pixels entre a imagem de origem e a imagem desejada, foi possível criar imagem que se assemelha ao máximo à segunda imagem fornecida. A validação do processo mostrou que a transformação foi aplicada corretamente, garantindo a precisão e confiabilidade do algoritmo desenvolvido.

Durante a execução do código, deparamo-nos com algumas dificuldades. No início, estávamos usando o método "rand () % tam" para sortear pixels. No entanto, a função "rand ()", em C, gera um número pseudoaleatório que varia de 0 a RAND_MAX, sendo que este último é, no mínimo, 32767. Esse valor é consideravelmente pequeno em comparação com a quantidade total de pixels em uma imagem, o que, consequentemente, fazia com que o resto da divisão nunca chegasse aos valores menores. Como resultado, os pixels localizados mais acima da imagem (com valores menores) nunca eram sorteados e, portanto, não sofriam alterações. Após analisar diversas possibilidades de solução, optamos por importar uma biblioteca que implementa o método "genrand64_int64() % (tam)", o qual proporciona uma distribuição maior de números pseudoaleatórios. Essa abordagem resolveu o problema, pois garante uma seleção mais equilibrada de pixels, permitindo que todos os pixels da imagem tenham a mesma probabilidade de serem selecionados para alteração.

Antes de chegarmos à abordagem apresentada neste relatório, testamos outra solução inicial. Nessa primeira versão, apenas comparávamos 3 pixels: um de referência da imagem 2, outro no mesmo local na imagem de saída e um terceiro pixel aleatório na imagem de saída. Com base nesses pixels, avaliávamos se o pixel aleatório na imagem de saída era mais semelhante ao de referência na imagem de saída do que o do mesmo local na imagem de referência, e assim realizávamos a troca. No entanto, observamos que esse algoritmo resultava em imagens com ruído e tinha um tempo de execução significativo.

Isso pode ser explicado pelo fato de que, em algumas situações, embora a troca fosse vantajosa em comparação com o pixel de referência na imagem 2, poderia ocorrer que a troca afetasse negativamente o cenário do pixel em que foi trocado. Ou seja, enquanto um lado melhorava, o outro poderia piorar, às vezes mais do que a melhoria obtida. Esse comportamento levava o algoritmo a realizar trocas excessivas e, consequentemente, a demorar mais para processar. Levando isto em consideração, desenvolvemos a solução apresentada que melhorou os quesitos acima apresentados.

Outro desafio enfrentado foi ao tentar utilizar imagens de teste, pois todas as imagens baixadas estavam no formato JPEG progressivo. Infelizmente, a biblioteca SOIL utilizada na aplicação do código não é capaz de identificar ou processar imagens JPEG progressivas. Como solução, foi necessário instalar um aplicativo adicional para converter as imagens JPEG progressivas em JPEG padrão. Somente após essa conversão foi possível utilizar as imagens adequadamente no programa.