Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
DCC004 - Algoritmos e Estruturas de Dados II
Profs. Cristiano Arbex Valle e Gisele L. Pappa

Trabalho Prático 1
Entrega: 11/05/2018
Valor: 15 pontos

Segmentação de Imagens

Breno Claudio de Sena Pimenta
Graduação em Engenharia Elétrica
Matrícula: 2017074424

1 Introdução

A segmentação de imagens é o processo de particionar uma imagem digital em múltiplos segmentos, ou seja, conjuntos de pixels, também conhecidos como super-pixels. O objetivo da segmentação é simplificar ou alterar a representação de uma imagem em algo que seja mais significativo e mais fácil de analisar. Cada um dos pixels de uma região segmentada é semelhante em relação a alguma característica como cor, intensidade ou textura. A segmentação é geralmente o primeiro estágio em qualquer tentativa de analisar ou interpretar uma imagem automaticamente no ramo da Visão Computacional. No entanto, uma segmentação confiável e precisa de uma imagem é em geral muito difícil de ser obtida por meios puramente automáticos. Esse processo é normalmente usado para localizar objetos seus limites em imagens. É utilizado para realizar inspeções industriais, reconhecimento ótico de caracteres, rastrear um objeto em uma sequência de imagens, detecção e medição de ossos, tecidos, em imagens médicas, entre outras aplicações.

Neste TP, será implementado um algoritmo de segmentação denominado algoritmo de crescimento de regiões. Este algoritmo define um grupo a partir de um pixel aleatório inicial, denominado semente, e expande o grupo para os vizinhos do pixel semente de acordo com um critério baseado na intensidade do pixel. Para um dado pixel P(u, v), sua vizinhança 4-conexa é dada pelos pixels P(u, v + 1), P(u + 1, v), P(u, v - 1) e P(u - 1, v). Se a diferença de intensidade entre dois pixels vizinhos for pequena, o grupo se propaga. Por exemplo, se $|P(u, v) - P(u; v+1)| \leq T$, então (u, v) e (u, v + 1) pertencem ao mesmo grupo (região), no qual T é um valor de limiar predefinido.

2 Implementação

De maneira geral, existem dois principais algoritmos para gerar as imagens de saída: um interativo com sementes pré-definidas e outro recursivo e com sementes aleatórias. Essa documentação estará focada em analisar o primeiro algoritmo. A segmentação iterativa utiliza o conceito de fila.

2.1 Estrutura de Dados

Para o desenvolvimento do programa, foram implementados alguns tipos abstratos de dados. Esses tipos são listados a seguir. Foi utilizado o conceito de filas.

```
typedef struct tipo_pixel_cinza {
    int cor;
    bool passei;
}tipo_pixel_cinza;
```

Esse tipo define um pixel do formato de cores de escala de cinza, possuindo um inteiro que armazena o valor de cada cor e também um booleano para verificar se esse pixel já foi visitado durante o crescimento da região. Será usado principalmente para gerar uma matriz para a imagem de saída.

```
typedef struct tipo_pixel_rgb {
    int R, G, B;
}tipo_pixel_rgb;
```

Esse tipo define um pixel do formato de cores RGB, possuindo um inteiro que armazena o valor de cada uma das cores. Será usado principalmente para gerar uma matriz para a imagem de saída.

```
typedef struct tipo_imagem_pgm {
    int linhas, colunas, nivel;
    struct tipo_pixel_cinza **pixels;
}tipo_imagem_pgm;
```

Esse tipo define uma imagem de formato PGM, chamada também de imagem de entrada. Possui informações do número de linhas (linhas), colunas (colunas) e nível de intensidade (nivel) dos pixels em escala de cinza que a compõem (**pixels), criados como uma matriz.

```
typedef struct tipo_imagem_ppm {
    int linhas, colunas, nivel;
    struct tipo_pixel_rgb **pixels;
}tipo_imagem_ppm;
```

Esse tipo define uma imagem de formato PGM, chamada também de imagem de entrada. Possui informações do número de linhas (linhas), colunas (colunas) e nível de intensidade (nivel) dos pixels em RGB que a compõe (**pixels), criados como uma matriz.

```
typedef struct tipo_semente_pai {
    int limiar, num_filhos, linha_inicial, coluna_inicial;
    struct tipo_pixel_rgb cor;
    struct tipo_semente_filho *primeiro;
    struct tipo_semente_filho *ultimo;
}tipo_semente_pai;
```

Esse tipo define uma semente inicial que será usada para armazenar os pixels que fazem parte de uma mesma região. Como é o ponto de partida do crescimento, esse tipo recebe o nome Pai e contém inteiros que especificam o limiar (diferença mínima entre cores de pixel para segmentação), número de filhos, linha e coluna iniciais, um apontador do tipo pixel RGB para armazenar as cores da região na imagem de saída. Além disso, como a segmentação usa crescimento por meio de fila, o Pai contém um apontador para o primeiro e último filhos da fila.

```
typedef struct tipo_semente_filho {
    int linha, coluna;
    struct tipo_semente_filho *irmao;
}tipo_semente_filho;
```

Esse tipo define uma semente denominada filho, que basicamente é uma célula de uma fila. Como é gerado a partir de uma semente inicial chamada Pai, esse tipo recebe o nome Filho e contém inteiros que especificam a linha e a coluna (coordenadas na matriz imagem), e um apontador para o próximo filho da lista, denominado Irmão.

2.2 Funções e Procedimentos

O trabalho desenvolvido possui as seguintes funções principais descritas a seguir.

`void testar_alocacao (void *ponteiro, int tipo):` Essa função testa se um ponteiro foi alocado corretamente ou se ele existe. O principal objetivo é realizar testes após a alocação dinâmica e abertura de arquivos. A entrada da função é um ponteiro genérico e o tipo de teste que será realizado (há o tipo MEMORIA e ARQUIVO, ambas são constantes definidas no programa). A diferença entre os tipos de teste é a mensagem de erro que irá ser exibida para o usuário. Caso o teste for mal-sucedido, a função termina o programa, porém, caso for bem-sucedido o programa é continuado normalmente. A função não retorna valores.

`FILE *abrir_arquivo (char *nome, char *formato, char *tipo_acesso):` Essa função abre um arquivo para o usuário. É uma variação da função fopen, nativa da biblioteca stdio.h com a diferença que recebe o nome do arquivo, o formato do arquivo e o tipo de acesso conforme a especificação da fopen. A função retorna um ponteiro para o arquivo solicitado pelo usuário.

`tipo_imagem_pgm *armazenar_imagem_entrada (FILE *arquivo):` Essa função cria um ponteiro para um tipo_imagem_pgm que irá conter os dados do arquivo de imagem de entrada. Após criar esse ponteiro e armazenar os dados de número de linhas, colunas e limiar, é alocada uma matriz dinâmica do tipo_pixel_cinza para gravar as cores da imagem de entrada. Como entrada da função é necessário passar um ponteiro para um arquivo que contém a imagem PGM. A função fecha o arquivo passado, e retorna um ponteiro para um tipo_imagem_pgm.

`tipo_semente_pai *armazenar_sementes (FILE *arquivo, int *num_sementes):` Essa função cria um vetor de tipo_semente_pai que irá conter os dados do arquivo auxiliar que contém as sementes pré-definidas. Após criar esse ponteiro, para cada semente são armazenados os dados de limiar, coluna inicial, linha inicial e as cores RGB. Como entrada da função é necessário passar um ponteiro para um arquivo que contém as sementes. A função fecha o arquivo passado, e retorna um ponteiro para um vetor de tipo_semente_pai.

`tipo_imagem_ppm *alocar_matriz_saida (tipo_imagem_pgm *imagem_entrada):` Essa função cria um ponteiro tipo_imagem_ppm que irá conter os dados da matriz de saída. Inicialmente a função seta todos os pixels para formar uma imagem idêntica à imagem de entrada, portanto, em escala de cinza. A função recebe como entrada um ponteiro para a matriz que contém a imagem de entrada (pgm) e retorna um ponteiro para a matriz dinâmica do tipo tipo_imagem_ppm.

`void desalocar_imagem_entrada (tipo_imagem_pgm *imagem):` A função desaloca uma matriz dinâmica de imagem de entrada. Recebe um ponteiro para a matriz dinâmica que contém imagem de entrada tipo_imagem_pgm. A função não retorna valores.

`void desalocar_imagem_saida (tipo_imagem_ppm *imagem):` A função desaloca uma matriz dinâmica de imagem de saída. Recebe um ponteiro para a matriz dinâmica que contém imagem de saída tipo_imagem_ppm. A função não retorna valores.

`void desalocar_sementes (tipo_semente_pai *sementes, int num_sementes):` A função desaloca um vetor dinâmico que contém as sementes. Primeiramente testa se a semente pai possui algum filho e desaloca um a um, em seguida desaloca o vetor. Ponteiro para o vetor dinamico tipo_semente_pai, numero de sementes. A função não retorna valores.

`void enfileirar (tipo_semente_pai *pai, int i, int j):` A função adiciona à fila pai um novo filho que irá conter os parâmetros i e j que indicam qual é a linha e coluna desse pixel a ser colorido. Essa fila é de itens a serem processados, ou seja, testar se a sua vizinhança faz parte da mesma região. Recebe um ponteiro para um tipo_semente_pai, coordenadas da semente na matriz de imagem. A função não retorna valores.

`void desenfileirar (tipo_semente_pai *pai):` A função remova da fila Pai o primeiro Filho caso ela não esteja vazia. Essa função recebe um ponteiro para uma semente pai que define a fila. A função não retorna valores.

`bool verificar_condicoes (tipo_imagem_pgm *imagem, int limiar, int cor, int i, int j):` Essa função verifica um conjunto de condições para que um pixel seja adicionado a uma região. A posição da linha (i) deve existir, portanto deve verificar se 0 ≤ i < altura da imagem, a posição da coluna (j) deve existir, portanto deve verificar se 0 ≤ j < largura da imagem. Além disso a função também verifica se o pixel a ser comparada já não foi analisado. Por fim, a função verifica se o módulo da diferença da cor de dois pixels é menor ou igual ao limiar (máximo aceitável). Caso todas essas condições sejam positivas, a função acessa o pixel em análise na matriz de entrada, e o marca como “passei”, para evitar que seja analisado novamente em outro ciclo. A função recebe como entrada um ponteiro para a imagem de entrada, o valor do limiar da semente pai, valor da cor de comparação, número da linha e da coluna do pixel em análise. A função retorna um valor booleano para habilitar o enfileiramento da semente (true) ou para pular o enfileiramento (false).

`void testar_vizinhos (tipo_imagem_pgm *imagem, tipo_semente_pai *pai):` Essa função testa a vizinhança 4-conexa do primeiro pixel de uma fila dada por P (u, v + 1), P (u + 1, v), P (u, v - 1) e P (u - 1, v), por meio da chamada da função verificar_condições. Inicialmente, a imagem de entrada é marcada no ponto P (u, v) como visitada. Em seguida, existem quatro comparadores if que testam se os pixels das direções cima, baixo, esquerda e direita podem ser enfileirados, ou seja, pertencem à mesma região. A função recebe um ponteiro para a imagem de entrada e um ponteiro para a semente pai. Caso nenhum dos pixels da vizinhança pertençam a região, a função não faz nada. A função não retorna valores.

`void colorir_imagem_saida (tipo_imagem_ppm *imagem_saida, tipo_pixel_rgb *cor_rgb, int i, int j):` A função colore a imagem de saída com os dados de cores de um pixel RGB. Recebe um ponteiro para a imagem de saída, um ponteiro para um pixel RGB e a posição (i, j) que deve ser colorida. A função não retorna valores.

`void resetar_matriz_passei (tipo_imagem_pgm *imagem):` A função redefine o verificador "passei" de todos os pixels da matriz dinâmica que contém a imagem de entrada (.pgm) para false. Recebe um ponteiro para a imagem de entrada. A função não retorna valores.

`void segmentar_regioes (tipo_imagem_pgm *imagem_entrada, tipo_imagem_ppm* imagem_saida, tipo_semente_pai *pai, int num_sementes):` A função contém o algoritmo iterativo para segmentar a imagem de entrada a partir de uma semente inicial. A função possui dois ciclos, o primeiro ciclo (tipo for) salva a semente de partida como o primeiro filho, isso irá se repetir de acordo com o número de sementes que a imagem possui. Em seguida, inicia-se o segundo ciclo (tipo while contido no primeiro) em que a função testar_vizinhos é chamada para verificar se a região 4-conexa do primeiro filho faz parta da região, caso algum faça, eles são adicionados à fila. A imagem de saída é colorida com base nas cores do Pai e coordenadas do primeiro Filho. Em seguida, é chamada a função desenfileirar visto que já foi analisado o primeiro ocupante da fila. Quando o número de filhos do pai seja igual zero, o segundo

ciclo se encerra, significando que a região já foi completamente segmentada. Após isso, a matriz de posições visitadas é resetada a fim de que se houver mais de uma semente para a mesma região, a semente que irá prevalecer seja a última chamada. A função recebe um ponteiro para a imagem de entrada tipo_imagem_pgm, ponteiro para a imagem de saída tipo_imagem_ppm, ponteiro para o vetor de sementes pai tipo_semente_pai. A função não retorna valores.

`void criar_imagem_saida (tipo_imagem_ppm *imagem, char *nome_arquivo):` A função cria um arquivo de imagem com as regiões segmentadas para acesso do usuário. Recebe um ponteiro para a imagem de saída tipo_imagem_ppm e o nome desejado do arquivo. A função não retorna valores.

`void segmentar_aleatoriamente (tipo_imagem_pgm *imagem_entrada, tipo_imagem_ppm *imagem_saida):` A função segmenta a imagem de entrada a partir de uma semente inicial de posição e cores aleatórias. Inicialmente a função cria uma variável tipo_pixel_rgb que irá conter as cores aleatórias na imagem de saída. O número de sementes é limitado a um número chamado limite_funcional que é o valor médio entre o número de linhas e colunas. A função recebe um ponteiro para a imagem de entrada tipo_imagem_pgm e um ponteiro para a imagem de saída tipo_imagem_ppm. Após gerar os valores aleatórios, a função testa se a posição gerada já foi segmentada, caso não tenha sido ainda, ela define a posição como visitada e chama a função de testar vizinhos recursivamente. A função não retorna valores.

`void testar_vizinhos_recursivamente (tipo_imagem_pgm *imagem_entrada, tipo_imagem_ppm *imagem_saida, tipo_pixel_rgb *cor_rgb, int i, int j):` A função testa se os vizinhos de cima, baixo, direita e esquerda de um pixel faz parte de uma mesma região a ser segmentada de maneira recursiva. Inicialmente a função colore a imagem de saída com os dados da posição passada para a função. Em seguida, a função testa por meio da função verificar_condições se o pixel de cima pertence a mesma região. Caso positivo, a função define essa posição como visitada e chama ela mesma, agora tendo como pixel de partida o pixel de cima do primeiro ciclo. A recursividade termina caso não exista mais como entrar nos comparadores if. A função recebe um ponteiro para a imagem de entrada tipo_imagem_pgm, ponteiro para a imagem de saída tipo tipo_imagem_ppm, ponteiro para um conjunto de cores RGB, número da linha e da coluna que será testada na imagem de entrada. A função não retorna valores.

2.3 Programa Principal

Inicialmente o programa testa se foi passado um arquivo de imagem, verificando a existência do argv[1] por meio da função testar_alocacao. Como será necessário gerar números aleatórios no programa, a função srand(time(NULL)); é chamada para gerar melhores números. Em seguida é armazenada a imagem de entrada (pgm).

O programa possui duas possibilidades de execução correta, primeiro: caso exista um arquivo de sementes pré-definidas,serão geradas duas saídas uma que utiliza o arquivo de semente auxiliar e outra saída utilizando sementes aleatórias e o algoritmo recursivo. Segundo: caso não exista um arquivo de sementes auxiliares, será gerado somente uma imagem com as sementes aleatórias.

Usando o arquivo de sementes auxiliares, primeiramente é aberto o arquivo e a função armazenar_sementes retorna um ponteiro para um tipo que contém a imagem de entrada. Em seguida, é alocada a matriz de imagem de saída (alocar_matriz_saida) e a imagem é segmentada (segmentar_regioes). Com a imagem de saída pronta, é criado um arquivo para que o usuário possa visualizar o resultado (criar_imagem_saida). O nome do arquivo será o nome passado por argumento de formato “ppm”. Como esse processo finalizou, a imagem de saída e o vetor de sementes são desalocados (desalocar_imagem_saida e desalocar_sementes).

Usando sementes aleatórias, primeiramente é alocada a matriz de imagem de saída (alocar_matriz_saida) e a imagem é segmentada (segmentar_aleatoriamente). Com a imagem de saída pronta, é criado um arquivo

para que o usuário possa visualizar o resultado (criar_imagem_saida). O nome do arquivo será o nome passado por argumento + "_recursivo" de formato "ppm". Como esse processo finalizou, a imagem de saída é desalocada (desalocar_imagem_saida).

Por fim é desalocada a imagem de entrada (desalocar_imagem_entrada) e o programa finaliza com sucesso.

2.4 Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código foi dividido em três arquivos, sendo eles:

- *main.c*: que contém as diretrizes principais do programa.
- *segmentacao-imagens.c*: que contém a definição das funções iterativas e recursivas.
- *segmentacao-imagens.h*: que contém os cabeçalhos das funções iterativas e recursivas.

Foi decidido fazer uma alegoria de pais, filhos e irmãos para representar a estrutura do programa para facilitar o entendimento das filas utilizadas. Foi definida a constante LIMIAR de valor 10 para as imagens geradas com cores aleatórias, visto que pelos testes, 10 é uma tolerância razoável na tonalidade dos pixels a serem pintados. A constante MAX_NOME valor 260 foi definida para ser o tamanho máximo para nome de arquivos, padrão do Windows 10.

Para o algoritmo de segmentação iterativa, caso duas sementes pré-definidas estejam na mesma região, a cor que irá prevalecer é a cor da última semente. Porém, no algoritmo recursivo, caso duas sementes aleatórias pertençam à mesma região, a cor que irá prevalecer é a da primeira semente. Isso se deve ao fato do algoritmo recursivo ser mais propenso a causar stack overflow, então a execução do algoritmo deve ser minimizada ao mínimo necessário.

O compilador utilizado foi o ambiente Debian (versão 9.3) instalado no Windows 10 Pro 64 bits (versão 10.0.16299, compilação 16299). Para executá-lo, basta digitar ./tp1, seguido do nome de uma imagem a qual deseja segmentar.

3 Análise de Complexidade

A análise de complexidade de tempo será feita em função da variável n, sua representação será especificada de acordo com a função analisada.

Função testar_alocacao: A função possui custo constante, apenas realiza dois testes if. Portanto, a função é O(1).

Função abrir_arquivo: A função apenas realiza três atribuições de custo constante. Portanto, a função é O(1).

Função armazenar_imagem_entrada: A função possui dois for alinhados que inicializam uma matriz dinâmica e preenchem cada uma das posições (i, j). Sendo i o número de linhas, j o número de colunas da imagem de entrada. Sendo a operação relevante a atribuição de um elemento na matriz, a função é portanto O(ij).

Função armazenar_sementes: A função realiza cinco atribuições relevantes que dependem do número de sementes. Sendo n o número de sementes, a função é O(n).

Função alocar_matriz_saida: A função realiza três atribuições de custo constante, e possui dois for alinhados que realizam três atribuições que dependem do número de linhas (i) e do número de colunas (j) da matriz, portanto a função é O(ij).

Função desalocar_imagem_entrada: A função depende do número de linhas (i) da imagem de entrada, visto que utiliza um for que vai de 0 ate número de linhas (i) para liberar a memória da matriz criada. Portanto, a função é O(i).

Função desalocar_imagem_saida: A função depende do número de linhas (i) da imagem de entrada, visto que utiliza um for que vai de 0 ate número de linhas (i) para liberar a memória da matriz criada. Portanto, a função é O(i).

Função desalocar_sementes: A função depende do número de sementes (n) e do número de filhos (k) de uma semente. Sendo que a operação relevante é a função free, a função, no pior caso O(nk). Porém, isso só ocorre se todos os pais tiverem filhos a serem desalocados. O caso mais comum no programa implementado é apenas realizar o free do vetor de sementes, portanto sendo O(1).

Função enfileirar: A função realiza sete atribuição no pior caso e seis no melhor caso, todas de custo constante. Portanto é O(1).

Função desenfileirar: A função realiza três atribuição no pior caso e uma atribuição no melhor caso, todas de custo constante. Portanto é O(1).

Função verificar_condicoes: A função realiza três atribuição no pior caso, e uma atribuição no melhor caso, todas de custo constante. Portanto é O(1).

Função testar_vizinhos: A função realiza cinco atribuições de custo constante. Portanto é O(1).

Função colorir_imagem_saida: A função realiza três atribuições de custo constante. Portanto é O(1).

Função resetar_matriz_passei: A função possui dois for alinhados que realizam uma atribuição que depende do número de linhas (i) e do número de colunas (j) da matriz. Portanto é O(ij).

Função segmentar_regioes: A função possui dois loops aninhados. O primeiro possui a função enfileirar que o custo unitário é O(1), mas a chamada da função depende do número de sementes (k), assim é O(k). O segundo loop é um while ocorre até que o número de filhos de uma semente pai seja diferente de zero. Considerando o pior caso em que o while precisa esperar que a semente colora a imagem inteira, o custo é O(ij), sendo i o número de linhas e j o número de colunas. Como a matriz é resetada, e considerando ainda o pior caso em que as sementes precisam colorir a imagem inteira o custo depende de i, j e k. Portanto é O(ijk).

Função criar_imagem_saida: A função realiza um fprintf de custo constante e possui dois for alinhados que realizam um fprintf que acessa três variáveis, depende do número de colunas (i) e do número de colunas (j). Portanto é O(ij).

Programa principal (main): A função main possui 15 atribuições e as funções mais relevantes são: segmentar_regioes e segmentar_aleatoriamente. Como a função recursiva será ignorada nessa análise, a complexidade do programa principal é dada pela ordem de complexidade da função segmentar_regioes, sendo portanto O(ijk). Sendo i e j o número de linhas e colunas de uma matriz, e k o número de sementes do arquivo auxiliar.

4 Testes

Os testes do programa foram feitos em um computador com o Windows 10 Pro 64 bits (versão 10.0.16299, compilação 16299) utilizando o utilizando o Valgrind (versão 3.13.0) para executá-lo no ambiente Debian (versão Debian 9.3). Processador Intel Core i7-4790K CPU 4.00GHz, 4001 Mhz, 4 núcleos, 8 procesadores

lógicos, memória física (RAM) instalada de 16,0 GB. Foram utilizados quatro arquivos de teste para o algoritmo iterativo e recursivo, o resultado foi o observado a seguir:

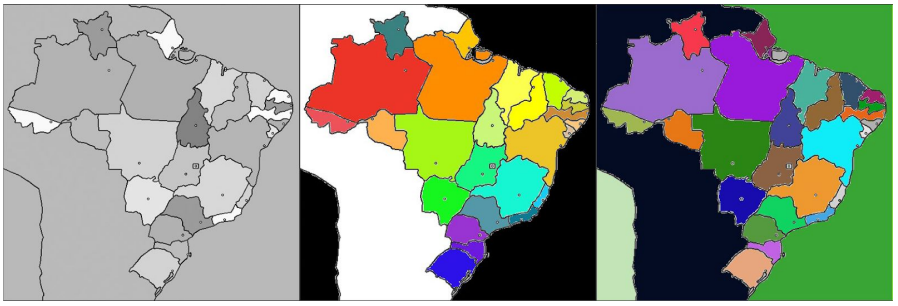

FIGURA 1: Utilizando a imagem brasil. Da esquerda para direita: imagem original, imagem gerada com sementes pré-definidas, imagem gerada com sementes aleatórias

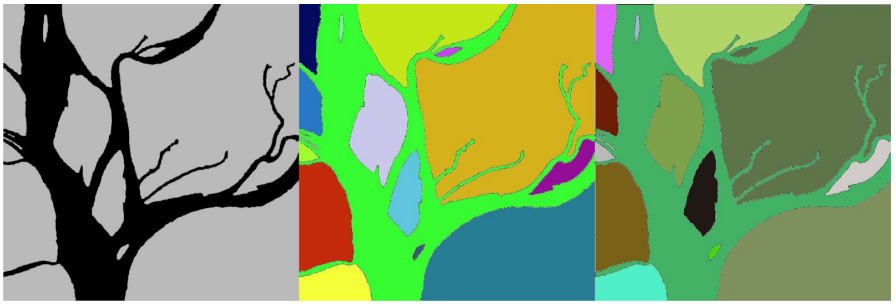

FIGURA 2: Utilizando a imagem amazon_river. Da esquerda para direita: imagem original, imagem gerada com sementes pré-definidas, imagem gerada com sementes aleatórias

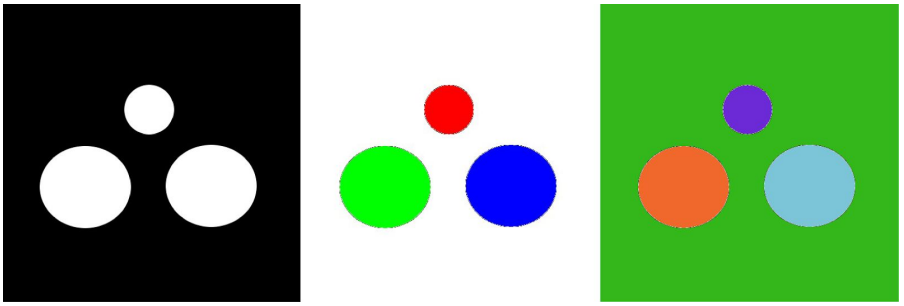

FIGURA 3: Utilizando a imagem split_2. Da esquerda para direita: imagem original, imagem gerada com sementes pré-definidas, imagem gerada com sementes aleatórias

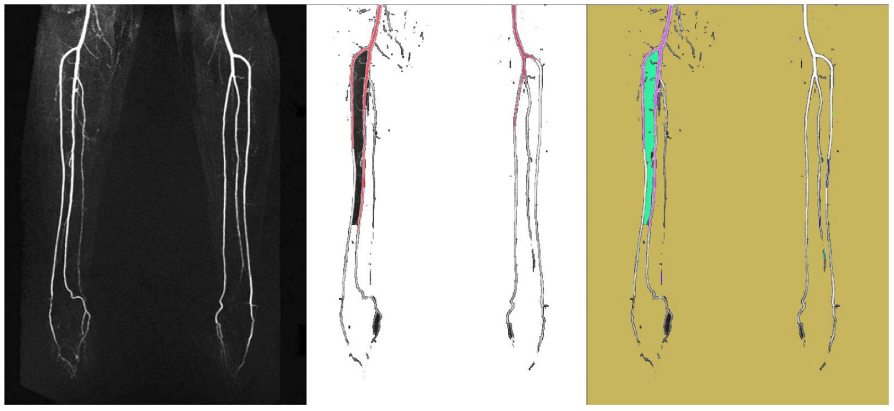

FIGURA 4: Utilizando a imagem vessels. Da esquerda para direita: imagem original, imagem gerada com sementes pré-definidas, imagem gerada com sementes aleatórias

5 Conclusão

A implementação do trabalho foi bem sucedida. A maior dificuldade foi em montar a estrutura do código (TADs e funções), tendo em vista as operações básicas de filas como enfileirar, desenfileirar e verificar se vazia. A opção de utilizar fila foi por maior similaridade à lógica elaborada no planejamento do trabalho. Durante os teste foi percebido que a lógica de testar os seis vizinhos de um pixel trouxe melhores resultados quando comparado ao teste de somente quatro. No código foi mantido quatro linhas comentadas que podem ser usadas para usar o algoritmo para testar os vizinhos das diagonais. O algoritmo recursivo não é a melhor opção de solução para esse problema. Tal solução é instável e pode ocasionar stack overflow em máquinas menos robustas devido à limitação do tamanho da memória stack.

Após análise dos resultados, pode-se dizer que a segmentação de imagens foi um sucesso. O algoritmo iterativo que utiliza sementes pré-definidas gerou imagens dentro dos parâmetros. Além disso, o algoritmo recursivo que utiliza sementes aleatórias, mesmo utilizando um número alto de sementes quando comparado ao arquivo auxiliar, às vezes deixou de segmentar regiões de interesse na imagem. Isso devido à probabilidade de encontrar todas as regiões relevantes ser baixa.

6 Referências

[1] COMPUTER Vision. Disponível em: <https://en.wikipedia.org/wiki/Computer_vision>. Acesso em: 01 maio 2018.

[2] IMAGE Segmentation. Disponível em: <https://en.wikipedia.org/wiki/Image_segmentation>. Acesso em: 01 maio 2018.

[3] O QUE são e onde estão o “stack” e “heap”?. Disponível em: <https://pt.stackoverflow.com/questions/3797/o-que- s%C3%A3o-e-onde-est%C3%A3o-o-stack-e-heap>. Acesso em: 01 maio 2018.

[4] YUHENG, Song; HAO, Yan. Image Segmentation Algorithms Overview. Disponível em: <https://arxiv.org/ftp/arxiv/pap ers/1707/1707.02051.pdf>. Acesso em: 01 maio 2018.

[5] ZIVIANI, Nivio. Projeto de Algoritmos com Implementações em Pascal e C. 3. ed. [S.l.]: Cengage Learning, 2010. 660 p.

8 Anexos

Lista dos arquivos:

1. *main.c*
2. *segmentacao-imagens.c*
3. *segmentacao-imagens.h*