# Documentação do TP1

Gustavo Guedes de Azevedo Barbosa

## 1. Introdução

Operações com imagens vem sendo o foco dos trabalhos práticos da disciplina de AEDS II e eu, como aluno, devo assumir que estou aprendendo muito tanto sobre as matérias na ementa do curso, quanto sobre o estudo de imagens digitais. O conhecimento da linguagem C também tem apresentado grande melhora.

Discutindo com colegas de sala, chegamos à conclusão de uma das utilidades da busca recursiva por um valor de pixel, que seria para aplicar um filtro topográfico à uma imagem, ou até mesmo, fazer o mapeamento do fluxo fluvial e pluvial em um terreno por meio de fotografias aéreas. Portanto, apresento a implementação das funções utilizadas para cumprir o objetivo desse Trabalho Prático.

## 2. Implementação

As estruturas de dados utilizadas na implementação foram estruturas de acesso aleatório (indexadas), que podem ser abstraídas como matrizes de índices linha e coluna, respectivamente. Para representar uma imagem PGM foi implementado um struct com os seguintes elementos: número de colunas e número de linhas da imagem, o valor máximo que um pixel pode assumir e uma matriz que representa a imagem.

As células que armazenam as coordenadas dos píxeis da imagem foram implementadas utilizando-se um struct denominado “struct C”, que contém dois inteiros, “l” e “c”, que armazenam a linha e a coluna do píxel da imagem. Além disso as células possuem um ponteiro para a próxima célula, para que a alocação dinâmica possa ser feita. Já para a pilha foi implementado um struct com um ponteiro de célula que aponta qual é o topo da pilha.

**As funções e os procedimentos implementados foram:**

*Observação: O termo “matriz”, utilizado ao longo do texto, se refere a ponteiro de ponteiros. Utiliza-se tal terminologia como uma forma de abstração dessa informação.*

**PGM* CriaPGM (int l, int c, int maximo):** Recebe valores inteiros referentes ao número de linhas, número de colunas e valor máximo de um pixel do novo tipo PGM que será criado. Essa função utiliza o comando “malloc” repetidamente para alocar dinamicamente (de acordo com as necessidades do usuário em tempo de execução), um ponteiro do tipo PGM e a matriz que receberá a imagem. Além disso, a função verifica se foi possível fazer tal operação. Caso negativo avisa o usuário e termina o programa. Caso positivo, retorna o ponteiro de PGM devidamente alocado.

Ordem de complexidade em relação às atribuições e alocações: Melhor caso é O(1), se o PGM contiver apenas um píxel. O caso médio é igual ao pior caso que é O(n). Sendo “n” a ordem da matriz que representa a imagem.

**PGM* LerPGM (char *entrada):** Recebe uma string (char*) como argumento, sendo o valor dessa variável o nome da imagem que será lida. O algoritmo checa se foi possível abrir o arquivo, imprimindo uma mensagem de erro caso negativo. Caso positivo, faz a leitura das primeiras linhas da imagem, que contém o formato de identificação do PGM (“P2”) e logo após, o numero de colunas, linhas e o valor máximo do pixel. A partir de então é iniciada a leitura dos pixels da imagem, por meio de uma iteração dupla que salva os valores em variáveis do tipo unsigned char, dentro de um ponteiro do tipo PGM. Tais iterações acontecem “c” vezes e “l” vezes, referentes, respectivamente, ao número de colunas e linhas da imagem. Há um teste que verifica se esses valores lidos condizem com as dimensões da imagem, imprimindo uma mensagem de erro e fechando o arquivo caso negativo. *É necessária uma observação: a mascara de leitura dos tipos numéricos unsigned char é “%hhu”, porém ela não é aceita quando utiliza-se a IDE Code Blocks no Windows e culmina em erros de segmentação e consequentemente, execução.*

Ordem de complexidade em relação às atribuições: Melhor caso semelhante à função CriaPGM. Já o caso médio e o pior caso são ambos $O(n^2)$, “n” continua sendo a ordem da matriz que representa a imagem.

**PGM* DesalocaPGM (PGM* alocado):** Recebe um ponteiro do tipo PGM no qual desaloca-se devidamente a matriz “alocado->imagem”, por meio de uma iteração que caminha pelos ponteiros do ponteiro imagem, desalocando-os e logo após, desaloca o próprio ponteiro imagem. Desaloca-se o ponteiro PGM, atribui seu valor a “NULL” para questões de verificação e segurança e então é retornado o ponteiro PGM desalocado.

Ordem de complexidade em relação ao acesso à memória primária: Melhor caso é O(1). Já o caso médio e o pior caso são ambos O(n), “n” continua sendo a ordem da matriz que representa a imagem.

**Pilha* CriaPilha ( )**: Não recebe parâmetros. A função declara uma variável do tipo ponteiro de pilha e aloca um espaço, dinamicamente, na

memória do tamanho de uma variável do tipo pilha. Caso a alocação não possa ser feita, é impressa uma mensagem de erro e o programa se desliga. Aquele espaço é atribuído ao ponteiro, que é retornado pela função.

Ordem de complexidade em relação às atribuições:  O(1), a independe da entrada.

**Celula* Desempilha (Pilha *pilha):** Recebe uma pilha da qual será retirada a última célula empilhada, ou seja, a que se encontra no topo da pilha e retorna essa célula.

Ordem de complexidade em relação às atribuições: O(1), a função independe do tamanho da entrada.

**Celula* Endocitose (int lina, int coluna):** Recebe dois inteiros, a linha e a coluna nos quais o pixel se encontra e insere-os em uma célula.

Ordem de complexidade em relação às atribuições: O(1), a função independe do tamanho da entrada.

i**nt PVazia (Pilha *pilha):** Recebe uma pilha qualquer e verifica se ela está vazia. Caso positivo retorna um (TRUE), caso negativo, zero (FALSE).

Ordem de complexidade em relação às comparações: O(1), a função independe do tamanho da entrada.

i**nt CaminhoValido (int l, int c, Pilha *pilha, PGM *img):** Recebe a linha e a coluna de uma pilha que está se ponderando empilhar, a pilha com os caminhos já percorridos na busca e a imagem. A função começa verificando se a linha e a coluna possuem valores negativos, ou seja, se haveria erro ao tentar acessar aquela posição da matriz da imagem, caso positivo, retorna zero, ou seja, o caminho não é válido. Caso positivo, a função caminha ao longo de toda a pilha para verificar se as coordenadas passadas como parâmetro já estão na pilha, para evitar o empilhamento de células idênticas e, portanto, o *loop infinito*. Se as coordenadas já existirem na pilha, retorna zero, se não existirem retorna um, ou seja, o caminho é válido.

Ordem de complexidade em relação às atribuições e comparações: Melhor caso é O(1), caso médio e pior caso são iguais, ambos O(n), “n” é o número de células que existem na pilha, ou seja, quantos píxels já foram escolhidos.

**void FPVazia (Pilha *pilha):** Recebe uma pilha e a esvazia.

Ordem de complexidade em relação às atribuições: O(1), a função independe do tamanho da entrada.

**void Empilha (Celula *celula, Pilha *pilha):** Recebe uma célula que é empilhada na pilha recebida, também, como parâmetro.

Ordem de complexidade em relação às atribuições: O(1), a função independe do tamanho da entrada.

**void DestroiPilhaCheia (Pilha *cheia):** Recebe uma pilha com mais de uma célula empilhada. Então, a desempilha e desaloca cada célula, no final, deixa a pilha vazia.

Ordem de complexidade em relação à atribuição: Melhor caso, O(1), se a pilha tiver apenas um elemento. Já o médio caso e o pior caso são ambos O(n), “n” é o número de células na pilha.

**void IniciaCaminho (Pilha *pilha, int l, int c):** Recebe uma pilha que será iniciada de acordo com o a linha “l” e a coluna “c” que deseja-se iniciar a busca.

Ordem de complexidade em relação às atribuições: O(1), a função independe do tamanho da entrada.

**void ImprimeCaminho (Pilha *pilha):** Recebe uma pilha da qual as células serão desempilhadas em outra pilha, para inverter a ordem de impressão. Então, nessa nova pilha, imprime-se as células desempilhadas. Essas células também são desalocadas após a impressão.

Ordem de complexidade em relação às atribuições: Uma vez que um loop passa por toda a pilha e realiza a função “Desempilha”, a função é O(n), “n” é o tamanho da pilha entrada.

**void ImprimeCelula (Celula *celula):** Recebe uma célula que será impressa de acordo com a notação: (linha, coluna).

Ordem de complexidade em relação ao acesso de memória: O(1).

**void Busca (PGM *img, Pilha *pilha):** Função “core” do código, recebe a imagem na qual será buscado o pixel 0 e uma pilha para armazenar as coordeadas pelas quais a busca passa no seu caminho até o zero. Utiliza-se de outras funções, mas tem como essência a recursão, que chama a própria função enquanto o valor do pixel referente à última célula empilhada for diferente de zero. Então o algorimtimo verifica quais píxeis podem ser comparados e verifica qual deles é o menor. Uma variável é iniciada como 256 a cada recursão, para impedir que caso um pixel que não pode ser acessado tiver o menor valor, a busca não pare por não encontrar o menor pixel acessável. Toda vez que o algoritimo encontra o menor pixel, ele salva as coordenadas desse pixel e as insere em uma célula. Essa célula é empilhada

na pilha da busca e usada como o “parâmetro” para a nova busca. E assim será, enquanto a condição de parada não for satisfeita.

Ordem de complexidade em relação às atribuições e acessos de memória: Melhor caso O(1), se o primeiro pixel da imagem for um 0. Já o médio e o pior caso são ambos $O(n^2)$, pois duas operações de atribuição do retorno de uma função são O(n), dessa forma, sabendo-se que “n” será a distância, passando pelos menores píxeis adjacentes, até o zero, teremos: (n + n-1 + n-2 + … + n-(n-1)), que tem como resultado $(n^2+n)/2$, que é $O(n^2)$.

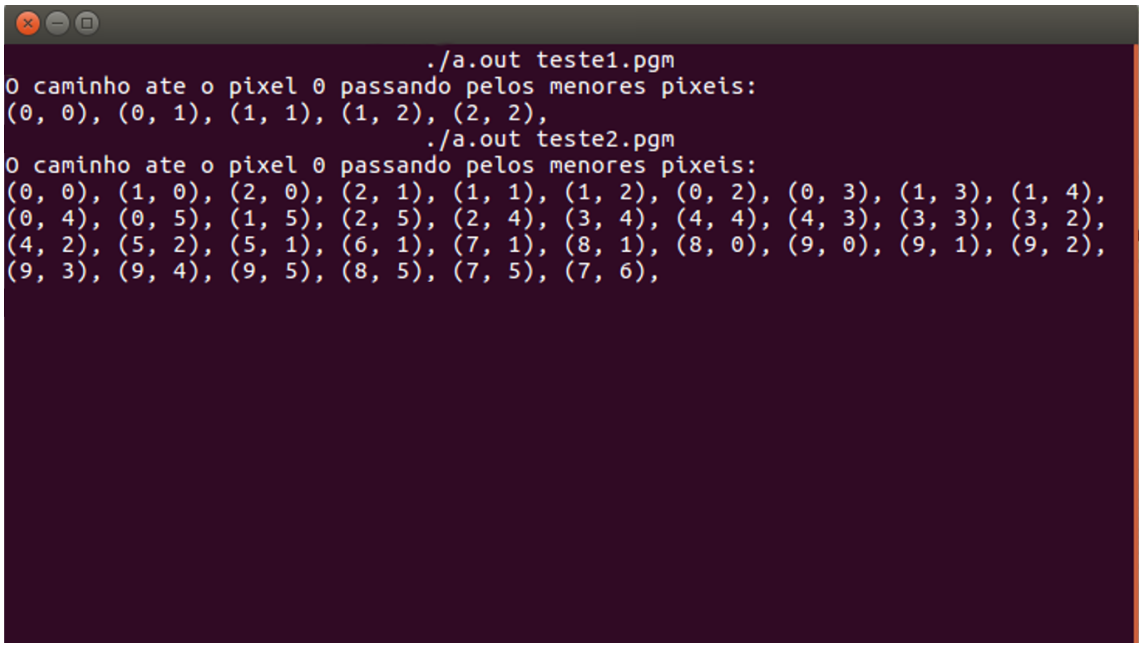
**Programa principal (main.c):**

A função main recebe argumentos passados ao iniciar o executável, o valor desses argumentos deverá vir na chamada do executável do programa, dentro do seu respectivo diretório, na forma: caso Windows, “foo.exe arg1”, caso Linux “./foo.out arg1”. Os argumentos devem conter o formato do arquivo, que nesse caso em específico é “*.pgm”. O primeiro passo é declarar as variáveis necessárias para testar a implementação dos TADs e dos algoritmos de alocação dinâmica. O PGM* minha_imagem é a imagem na qual o usuário deseja realizar a busca recursiva. O Pilha *caminho é a pilha na qual será armazenado o caminho do ponto inicial até o pixel de valor zero. Inicia-se, então, a pilha do caminho a partir das coordenadas (0, 0) e executa a busca. Ao fim da busca, imprime-se a pilha.

**Organização do código, decisão de implementação e detalhes técnicos:**

O código está dividido em três arquivos principais: tp1.c, tp1lib.h, que implementam o Tipo Abstrato de Dados, e main.c implementa o programa principal. Foi utilizado o compilador GNU gcc v4.5.1 no Sistema Operacional Ubuntu 16.04 64-bits. O programa “Valgrind”, versão 3.11.0, com a ferramente de verificação de memória foi usado para assegurar que não haja “*memory leaks*”. A forma de executar o programa já foi especificada no tópico “Programa principal (main.c)”.

**Testes:**




O programa foi executado utilizando as imagens “teste1” e “teste2” passadas na declaração do Trabalho Prático e os resultados foram idênticos aos esperados, enviados também no arquivo do TP:

“(2,2), (1,2), (1,1), (0,1), (0,0)”, para “teste1.pgm”.

E (7,6), (7,5), (8,5), (9,5), (9,4), (9,3), (9,2), (9,1), (9,0), (8,0), (8,1), (7,1), (6,1), (5,1), (5,2), (4,2), (3,2), (3,3), (4,3), (4,4), (3,4), (2,4), (2,5), (1,5), (0,5), (0,4), (1,4), (1,3), (0,3), (0,2), (1,2), (1,1), (2,1), (2,0), (1,0), (0,0), para “teste2.pgm”.

**Conclusão:**

A implementação do trabalho transcorreu sem maiores problemas e os resultados obtidos foram extremamente satisfatórios. Tive uma grande dificuldade inicial de saber como não passar por coordenadas já visitadas, mas depois de certo tempo, conclui que a utilização de uma função que percoresse a pilha com os caminhos já selecionados resolveria o problema. Surgiram alguns problemas também quanto aos “*memory leaks*”, no entanto esses foram resolvidos devidos à melhor compreensão de ponteiros e endereços de memória em C.

Ao tentar executar o programa nas imagens do TP0, percebi que resultava em erros. Após uma investigação, percebi que a busca entrava em um espiral, ou seja, chegava um momento que não existia mais caminhos válidos para a busca. Isso ocorre devido à falta de um mecanismo para fazer um “*rewind”* e tomar uma decisão diferente para garantir o sucesso do algoritimo.

**Referências**

Ziviani, N., Projetos de Algoritimos com implementações em PASCAL e C, 3ª edição, CENGAGE Learning, 2010.

**Anexos:**

Listagem dos programas

-tp1lib.h

-tp1.c

-main.c