João Marcos Couto
Algoritmos e Estruturas de Dados II
Documentação TP1 (Segmentação de Imagens)

1. Introdução

Neste trabalho prático, buscamos criar um algoritmo que facilite a rápida diferenciação e identificação de entidades presentes em uma imagem através da criação de grupos que descrevem quais segmentos (conjuntos de pixels) da imagem as compõem. Para este fim, o algoritmo recebe uma imagem em formato PGM (que descreve os pixels em tons de cinza) e gera uma nova em formato PPM (que descreve os pixels em cores RGB), na qual entidades identificadas estão pintadas em cores de forte contraste em relação umas às outras.

O algoritmo implementado para possibilitar a identificação das entidades presentes na imagem é denominado crescimento de regiões, no qual dado um pixel qualquer, seus quatro vizinhos conexos pertencem ao mesmo grupo que ele apenas se a diferença absoluta entre as intensidades deles é menor ou igual a uma dada constante T. E assim, analisando pixel a pixel, as regiões de entidades se propagam.

2. Implementação

Após ler a proposta do algoritmo a ser implementado, rapidamente optei pelo uso de pilhas na implementação. Essencialmente o que o algoritmo faz é, após receber os tons de cinza dos pixels (vindos do PGM original) e ler as informações do arquivo auxiliar, ele cria uma pilha para cada uma das sementes, e, utilizando um tipoitem que armazena as coordenadas de um pixel, empilha as sementes. Tendo feito isso, o algoritmo então passa a desempilhar cada uma das pilhas marcando em uma matriz de tamanho fixo o grupo do pixel desempilhado e empilhando seus quatro vizinhos caso eles já não pertençam a um grupo e caso a diferença de intensidade entre eles e o pixel desempilhado seja menor ou igual ao linear T. Tendo feito a segmentação dessas regiões, o programa então gera um PPM final, no qual cada pixel é descrito pela cor do grupo da semente que o gerou.

Estruturas de Dados:

Em termos de estrutura de dados, a implementação é primordialmente baseada em pilhas e matrizes. Assim, optei pela criação de dois TADs diferentes, uma que encapsula as estruturas e operações necessárias para a implementação de pilhas (pilha.c/.h) e outra que encapsula as estruturas e operações necessárias para realizar a segmentação da imagem propriamente dita (segmentation.c/.h)

TAD pilha: Foi criado um TipoPilha que contem ponteiros para o topo e fundo de uma determinada pilha, um tipo Célula que contem um apontador para a próxima célula na pilha assim como um TipoItem, que por sua vez armazena as coordenadas X e Y de um determinado pixel empilhado. Por fim um tipo Apontador que é um ponteiro para o tipo Celula descrito anteriormente.

TAD segmentation: A fim de possibilitar a segmentação da imagem, foram criadas duas matrizes, uma (denominada image[][]) que armazena os tons de cinzas dos pixels da imagem PGM original e outra (denominada seg[][]) que armazena a qual grupo dos grupos gerados pelo algoritmo de expansão de grupos cada pixel pertence. Ademais, foi necessária a criação de uma TipoCor, que armazena a intensidade (na maioria dos casos, entre 0 a 255) dos tres canais de cores, vermelho, verde e azul (padrão RGB)

Funções e Procedimentos:

Os TADs utilizados interagem entre si na grande maioria das operações, então a inclusão de um no outro foi imperativa. Assim, temos dois tipos de operação, aqueles referentes à manipulação das pilhas (encapsuladas no TAD pilha) e aquelas que operam sobre as matrizes relacionadas à segmentação da imagem (encapsuladas no TAD Segmentation). Além disso, funções auxiliares (tais como, abertura de arquivos ou

inicialização de uma matriz) foram também definidas dentro da TAD segmentation, visto que numericamente não justificavam a criação de um TAD auxiliar.

TAD Pilha

int Vazia(const TipoPilha *Pilha): recebe o endereço de um TipoPilha e verifica se o ponteiro para o fundo aponta para a mesma célula que o ponteiro para o topo, isto é, se a lista esta vazia, se for o caso, retornara 1, do contrário, 0.

void FPVazia(TipoPilha *Pilha):): recebe o endereço de um TipoPilha e aloca uma célula inicial para ela, isto é, cria uma lista vazia. Para tal, aloca memória do tamanho de uma célula para o topo do TipoPilha recebido, faz o fundo apontar para o mesmo endereço e por fim faz com que o ponteiro prox desta célula aponte para NULL

void Empilha(TipoItem x, TipoPilha *Pilha) : recebe um TipoItem e o endereço de um TipoPilha. Empilha o TipoItem recebido na pilha recebida. Para tal, aloca memoria do tamanho de uma célula para um Apontador temporário, atribui ao TipoItem da Celula apontada por Topo o TipoItem recebido e então faz o ponteiro prox da célula auxiliar apontar para a célula que recebeu o TipoItem. Por fim, faz com que a célula auxiliar seja o novo topo da lista.

int Desempilha(TipoPilha *Pilha, TipoItem *item) : recebe o endereço de uma pilha e o endereço de um TipoItem. Desempilha a célula apontada por topo da lista recebida. Para tal, cria um Apontador temporário e faz com ele aponte para o topo da lista. Então, faz com que o topo aponte para a célula apontada pelo prox do topo original. Por fim, libera o espaço de memoria apontado por q e atribui ao valor apontado pelo endereço de TipoItem recebido como parâmetro, o TipoItem apontado pelo novo Topo da Pilha. Retorna 0 caso a pilha recebida estiver vazia e 1 caso tenha conseguido desempilhar o item com sucesso.

TAD segmentation:

void Init2D(int DimY , int DimX , short int vetor[DimY][DimX]): recebe um vetor de inteiros bidimensional e suas duas dimensões. Inicializa os valores do vetor recebido com 0's. Para tal, navega em todas as posições do vetor utilizando dois laços for, atribuindo 0 a cada posição visitada

FILE * AbreArquivo(char Nome[] , char Ext[] , char Modo []): recebe tres vetores de caracteres, o primeiro é um nome de arquivo, a segunda é a extensão deste arquivo e a ultima é o modo de abertura do arquivo. Realiza a abertura de um arquivo com nome Nome, com a extensão Ext no modo (de fopen) Modo. Para tal, utiliza uma string auxiliar Filename, que recebe Nome e é então concatenada com a extensão. A função então retorna o endereço de um ponteiro para File através da função fopen .

void EmpilhaSemente(TipoCor CorSemente[], int NumSementes, TipoPilha Pilha[], FILE* aux): recebe um array de TipoCor, um inteiro representativo do número de sementes, um vetor de TipoPilha e o endereço do arquivo auxiliar. Para cada semente, lê, do arquivo auxiliar recebido, suas coordenadas X e Y assim como a cor que devem receber os pixels que serão marcados como membros do mesmo grupo que ela após a segmentação. Tendo feito isso, o procedimento então, utiliza um TipoItem auxiliar para empilhar cada uma das sementes em suas respectivas Pilhas.

void Segmenta(int NumSementes,TipoPilha Pilha[],int TamY, int TamX, short int seg[TamY][TamX], short int image[TamY][TamX], int T) : algoritmo principal de todo o programa visto que implementa a expansão de regiões propriamente dita. Recebe um inteiro referente ao numero de sementes ; um vetor TipoPilha referentes ao conjunto de pilhas de todas as sementes ; dois inteiro referentes às dimensões da imagem ; duas matrizes de short int, sendo a primeira referente à matriz seg que armazena os grupos de cada pixel e a segunda referente à matriz image que armazena os valores em escala de cinza de cada pixel da image PGM original ; um inteiro referente ao número T que configura o threshold da diferença entre os tons de cinza para que dois pixels sejam membros de uma mesma região. Para cada uma das NumSementes pilhas, a função desempilha um tipoitem e, na matriz seg, marca o pixel localizado nas coordenadas desempilhadas com o numero do grupo da semente que

inicializou a criação daquela região, isto é, um pixel nas mesma região da semente #1 sera marcado com o valor 1. Tendo feito isso, o algoritmo então empilha os vizinhos deste pixel caso eles ainda não sejam membros de um grupo e caso a diferença entre o pixel e um dado vizinho é menor ou igual a T

void GerarPPM(int DimY,int DimX ,short int seg[DimY][DimX],short int image[DimY][DimX], TipoCor CorSemente[], FILE* ppm, int NumSementes, int PixelDepth): recebe dois inteiros referentes às dimensões da imagem ; duas matrizes short int, a primeira referente à matriz com os pixels PGM originais e a segunda referente à matriz de grupos gerada pelo algoritmo de crescimento de regiões ; um vetor de TipoCor referente às cores que cada região gerada deve ser pintada no PPM final ; um ponteiro para FILE referente ao ppm final ; um inteiro referente ao numero de sementes no arquivo auxiliar ; um inteiro referente a quantos níveis de intensidade existem na imagem. A função gera o arquivo de saída final com o mesmo nome da imagem original porem com a extensão .ppm. Para tal, analisa o vetor de grupos, seg, e averigua a qual grupo cada pixel pertence e, então, consultando o vetor CorSemente, escreve no arquivo de saída a cor (no padrão RGB) de cada pixel da imagem. No caso de um pixel não ter sido colocado em nenhum dos grupos gerados pela semente, a função atribui como cor desse pixel, seu tom de cinza original no arquivo PGM.

PilhasVazias(int NumSementes, TipoPilha Pilha[]): recebe um inteiro referente ao numero de sementes recebidas no arquivo auxiliar ; um vetor de TipoPilha referente aos conjuntos das pilhas das sementes . A função inicializa as pilhas de cada uma das sementes como pilhas vazias, possibilitando operações futuras sobre elas. Para tal, chama a função FPVazia (já previamente descrita nesta documentação) para cada uma das NumSementes pilhas através de um laço for.

void LerPGM(FILE* pgm,int TamX, int TamY, short int image[TamY][TamX]) : recebe um endereço para arquivo referente ao arquivo pgm de onde virão os tons de cinza dos pixels ; dois inteiro referentes às duas dimensões da imagem; uma matriz de inteiro que armazenara dentro do programa os tons de cinza dos pixels da imagem PGM original. A função realiza a leitura da imagem PGM a ser segmentada, armazenando os pixels no vetor image. Para tal, utiliza dois laços for aninhados e limitados pelas dimensões previamente lidas a fim de percorrer o arquivo inteiro, e, para cada iteração, chamando a função fscanf com parâmetros (pgm, "%d", &image[i][j])

void LerAuxiliar(FILE* aux, int* NumSementes, int* T) : recebe um endereço para arquivo referente ao arquivo auxiliar aux ; dois endereços de inteiros, o primeiro referente ao numero de sementes declarados no arquivo auxiliar e o segundo referente ao numero T que configura o threshold da diferença entre os tons de cinza para que dois pixels quaisquer sejam membros de uma mesma região. Para tal, utiliza a função fscanf para ler do arquivo aux os valores das duas variáveis

3. Estudo de Complexidade

FPVazia: a função tem complexidade na ordem O(1) pois ela faz apenas 3 operações simples modificando ponteiros e alocando memória

PilhasVazias: a função é O(n) pois ela chama (através de um laço for) uma função O(1) (como visto acima) uma vez para cada uma das sementes

Init2D: tem complexidade O(xy) pois é composta por dois laços for aninhados que executam TamY e TamX vezes respectivamente, isto é, você faz x execuções y vezes.

FILE * AbreArquivo: Assumindo que as funções strcpy e strcat são O(1) trata-se de uma função O(1), pois ela apenas efetiva uma chamada dessas funções uma única vez

Empilha: O(1) pois apenas aloca memória uma única vez e faz 3 atribuições a ponteiros

EmpilhaSemente: trata-se de uma função O(n) pois ela é composta por apenas uma laço for que realiza NumSementes iterações, sendo que cada iteração é composta pela chamada de tres funções O(1)

GerarPPM: esta função, uma das principais do programa é composta por pois laços for que são limitados pelas dimensões X e Y da imagem, isto é, o loop é faz X execuções Y vezes. Em cada execução, temos uma sequência de if's e else's, porém em qualquer um dos casos, uma função fscanf(presumidamente, O(1)) é chamada, e portanto a função tem ordem de complexidade O(xy)

LerAuxiliar: é O(1) pois trata-se apenas de duas atribuições feitas a duas variáveis distintas por uma função fscanf

Desempilha: apenas realiza tres atribuições, é, portanto, O(1)

Segmenta: esta função, que implementa o algoritmo de expansão de grupos, é composta por um laço for dentro de um laço while que por sua vez esta aninhando debaixo de outro laço for. O laço for mais externo é limitado pelo numero de sementes e assim executa n vezes. Já o laço while, apenas para de ser executado quando a lista em questão fica vazia, e isso apenas ocorre quando não houverem mais itens a serem desempilhados da lista. Isso quer dizer que, na pior das hipóteses, todos os pixels da imagem pertencem a alguma das regiões e seriam então empilhados (uma única vez, visto um pixel não é empilhado se ele já for membro de uma das regiões. Portanto, o laço while seria executado xy vezes, em vista de que haveriam xy pixels para desempilhar antes que a lista estivesse vazia. Para cada execução do while, antes da chamada do for interno, apenas operações O(1) são efetivadas. Já o laço for mais interno, executa seu loop exatamente 4 vezes (pois, para todo pixel, existem 4 vizinhos) e, em cada execução, ou um pixel é empilhado ou não, portanto O(1). Assim, temos que a complexidade da função será dada por n*xy*4*O(1) = O(nxy) ;

LerPGM: assim como muitas das funções anteriores, sua complexidade é dada por O(xy) em vista de que é composta por 2 laços for, cada um limitado por uma das dimensões da imagem e de que cada uma das iterações do for interno é composta por apenas uma chamada de fscanf presumidamente, O(1) ;

Vazia: tem complexidade O(1) pois realiza apenas uma comparação para averiguar se o topo e o fundo da lista apontam para a mesma posição de memória, isto é, se a lista recebida esta vazia

Assim, somando todas as funções e procedimentos utilizados na implementação deste trabalho pratico, temos que sua complexidade será da ordem O(xyn), onde n é o numero de sementes recebidas no arquivo auxiliar e x,y são as dimensões da imagem sendo segmentada.

4. Conclusão

No geral a implementação deste programa transcorreu sem maiores problemas, pois embora tenham parecidos alguns desafios técnicos naturais a serem superados, a grande maioria das abstrações presentes neste problema são bastante intuitivas, como por exemplo o fato de que uma imagem pode ser descrita como uma sequência de números que especificam a cor de cada um dos seus pixels, ou ainda que a identificação de uma entidade em uma imagem pode ser feita a partir da análise do delta das intensidades entre pixels vizinhos. Isto faz com que o trabalho seja do tipo no qual a partir do momento em que o programador consegue "ter a sacada" e formular no papel um tipo abstrato de dados suficiente para o problema, a implementação ocorre de maneira relativamente suave até o termino do programa.

Nota ao monitor: Fernando, havia terminado o programa a muitos dias e deixe para implementar o ponto extra nesta sexta-feira. Todavia, me deparei com um problema sério com o CodeBlocks que simplesmente não consegui resolver a tempo. O que aconteceu é que assim que eu declarava mais um vetor [TamX][TamY], necessário para a minha implementação com sementes aleatórias do ponto extra,o programa passava a dar crash, mesmo que todas as outras modificações e adições estivessem comentadas.. Ou seja, o CodeBlocks estava com o limite extremamente pequeno de alocação de memória para o programa e com isso o programa não passava da compilação. Deixei comentada em minha main tudo que seria pertinente para a implementação do ponto extra

para caso você pode cogitar a possibilidade de avaliar minha tentativa.. Caso não, peço que simplesmente ignore as operações e declarações que deixei comentadas na main.

5. Bibliografia

[CLRS3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms*, 3rd edition, MIT Press, 2009.*

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage, 2017

https://stackoverflow.com/questions/34844003/changing-array-inside-function-in-c

https://www.tutorialspoint.com/c_standard_library/c_function_fscanf.htm

comentários

, em vista de que seria um processo relativamente intuitivo de implementar a ideia de que o ato de desempilhar um pixel