Trabalho Prático Matemática Discreta – DCC216 TW

Samuel Assis Vieira – 2018109736

Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte , MG – Brazil
samuelassis@dcc.ufmg.br

1. Introdução

O trabalho prático tinha como objetivo nos fazer trabalhar e se familiarizar com relações binárias, sendo assim foi pedido a implementação de um programa que lesse um conjunto de pares representando a relação e então verificasse uma série de propriedades da relação como simetria, anti-simetria, transitividade, reflexividade, irreflexividade e assimetria.

2. Implementação

Para a solução deste problema a estrutura de dados que utilizei foi uma matriz quadrada representando as relações em suas coordenadas. O programa lê todas as entradas de relações e coloca na coordenada correspondente o valor 1, por exemplo,(1,3) tinha o valor 1 na coordenada matriz[1][3].

Foram usados na implementação vetores da *struct Pair,* struct criada para representar um par de relação, a fim de facilitar o retorno das funções de verificação das propriedades. Para as propriedades as foram implementadas 6 funções, uma para cada, sendo elas:

- **reflexive(), unreflective() e symmetric() :**

  Essas três funções tem funcionamento e implementações muito parecidas sendo assim irei as descrever junto. As funções recebem como parâmetro: a matriz de relações, um vetor de pares por referencia e dois inteiros com o maior e menor numero das relações. A funções declaram um inteiro *boo* e ele é o retorno das funções (1 para propriedade válida e 0 para inválida). A função seta *boo* como 1 e então através de dois laços aninhados percorre a matriz verificando a validade da propriedade, caso inválida o programa seta *boo* como 0 e salva o par que invalida a propriedade no vetor de pares. Após percorrer a matriz as funções retornam *boo.*

- **antisymmetric() e asymmetric() :**

  Assim como as acima, essas duas funções tem funcionamento igual. Ambas recebem como parâmetro: um vetor de pares por referência, a matriz de relações e dois inteiros com o maior e o menor numero das relações. A função inicialmente seta um inteiro *boo* como 1 e então percorre a matriz através de dois laços aninhados e procura pelos pares que invalidam a propriedade, caso estes existam então *boo* é setado 0 e os dois pares que invalidam a propriedade são salvos no vetor de pares e também suas coordenadas na matriz são setados com o valor 2 para evitar que pares já salvos sejam salvos novamente no vetor. Após percorrer a matriz as funções chamam uma função para recuperar a matriz a seu estado original de entrada e então retornam *boo.*

- **transitive() :**

  A função que verifica se a relação é transitiva. Recebe como parâmetro: dois vetores de pares por referência, a matriz de relações e dois inteiros com o maior e o menor valor das relações. Como as outras, a função seta um inteiro *boo* como 1 e percorre a matriz com dois laços aninhados, entretanto a cada coordenada que seja 1 é iniciado outro laço e dentro deste é testada a validade da propriedade. Caso seja invalida *boo* é setada 0, o par que invalida é salvo no 1º vetor de pares e as coordenadas deste par na matriz são setadas com valor 3 para evitar repetição. No caso contrário, ou seja, quando a propriedade é válida um dos pares é salvo no 2º vetor e as coordenadas na matriz são setadas com valor 2 para evitar repetição. Ao final dos laços a função retorna *boo.*

Em todas essas funções a fim de diminuir o numero de comparações inicio os laços com o menor valor da relação.

- **MatrixRecovery() :**

  É uma função auxiliar para recuperar os valores setados como 2 para 1 novamente na matriz de relações. Recebe como parâmetro: a matriz e um vetor de pares com as coordenadas a serem recuperadas. A função percorre o vetor com um laço e vai dois a dois setando as coordenadas na matriz como 1.

- **Vclear():**

  Procedimento auxiliar para limpar um vetor da *struct Pair.* Recebe como parâmetro: um vetor de pares e um inteiro indicando o tamanho do vetor. A função percorre o vetor zerando todas as posições.

- **Vprint() e Tprint() :**

  Procedimento auxiliares para printar na tela o conteúdo de um vetor de pares. Ambas recebem como parâmetro um vetor e um inteiro indicando o tamanho do vetor. Percorrem o vetor printando o conteúdo do vetor. Diferem apenas na forma do print, Vprint printa os valores um a um e Tprint printa dois a dois.

- **Main():**

  A função main faz toda a gerência do programa. Começa lendo os dados e salvando o maior e menor numero das relações,respectivamente m e M, em seguida aloca dinamicamente a matriz de relações do tamanho MxM e zera todas as suas posições. Lê então os pares de relação e os marca na matriz e conta a quantidade de entradas. Aloca dinamicamente os dois vetores de pares neles serão armazenados os pares que invalidam as propriedades. Em seguida a função chama cada uma das propriedades e baseado no seu valor de retorno printa na tela se são verdadeiras ou falsas e nesse caso printa os pares q estão no vetor de pares. A cada vez que o vetor é utilizado ele é limpo em seguida. Por fim a função também testa se a relação é equivalente, se é de ordem parcial e printa o fecho transitivo da relação.

3. Instruções de Compilação e Execução

Foi criado um Makefile para facilitar a compilação do programa. Todo o código foi feito em C. O código foi desenvolvido e testado numa maquina com processador Intel Core 2 Duo, 4GB RAM e SO Linux Mint 20 Cinnamon.

Para realizar a execução do makefile e compilação do programa basta baixar o arquivo SamuelAssisVieira.zip descompactar e acessar o mesmo pelo terminal e então digitar o comando ‘make’, o programa será compilado e o executável terá o nome ‘main’, para executá-lo basta digitar ‘./main’.

4. Analise de Complexidade

4.1. Complexidade de Espaço

- **Main():**

  Sendo a entrada com o maior e menor numero como M e m respectivamente e com n entradas de pares. A função declara 14 inteiros(14 * 4 bytes), aloca uma matriz quadrada de MxM inteiros e aloca também 2 vetores da *struct Pair (8 bytes, 2 int)* de tamanhos n e 2n. Sendo assim a complexidade de espaço pode ser dada pela expressão: $56 + 4\ M^2 + 24n$ bytes

  As outras funções tem custos de memoria constante.

4.2 Complexidade de Tempo

- **main() :** O laço aninhado para alocar e zerar a matriz é o mais custoso da função. Com complexidade $O(M^2)$ no pior caso e $O(1)$ no melhor. Sendo M o maior número dos pares da relação.
- **reflexive(), unreflective(), symmetric(), antisymmetric() e assymetric():** As funções trabalham com laços aninhados para percorrer a matriz, começando com m (menor valor da relação) até M (maior valor da relação) sendo assim são feitas n = M – m comparações em cada laço. As funções tem no melhor caso $O(1)$ e $O(n^2)$ no pior caso.
- **Transitive():** É a função mais custosa de todo o algoritmo. Trabalha com 3 laços aninhados, sendo o 3º ativado em situações especificas. Os laços começam com m (menor valor da relação) até M (maior valor da relação) sendo assim são feitas n = M – m comparações em cada laço. No melhor caso ela não entra no 3ª laço ficando com complexidade $O(n^2)$, já o pior caso é o 3º laço ser sempre acionado resultando na complexidade $O(n^3)$.

Sendo assim podemos afirmar que o programa tem complexidade de $O(n^3)$.

5. Conclusão

Durante a implementação do trabalho prático não tive grandes dificuldades, a função que foi mais custosa foi a transitive(). Demorei um pouco mais pra entender os três laços aninhados, mas o fato de não ter que se preocupar tanto com a complexidade do programa facilitou. Achei o trabalho interessante e coerente.

Referencias

Slides da disciplina de Matemática Discreta