Trabalho Prático 1 - Algoritmos 1

Guilherme de Abreu Lima Buitrago Miranda - 2018054788

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

guilhermemiranda@dcc.ufmg.br

1. Introdução

O trabalho prático 1 da disciplina de Algoritmos 1 consiste em ajudar os alunos do grupo de Blackjack da UFMG a organizar melhor suas reuniões a respeito do estudo aprofundado do livro Beat the Dealer de Edward Oakley Thorp, na intenção de viajarem para Las Vegas e colocar em prática o conhecimento adquirido.

Assim, é necessário implementar três funções básicas: Swap, Commander e Meeting que, respeitando uma hierarquia previamente definida, troca a relação entre comandante e comandado, encontra o comandante mais jovem e identifica uma possível ordem de fala, respectivamente.

Devido à natureza do problema, a abordagem escolhida para a tarefa foi a modelagem da situação a partir de um grafo, estrutura de dados largamente utilizada dentro de diversos contextos dentro da computação, tais como problemas de trânsito, troca de informações via web, conexões em redes sociais, etc. Além disso, para as três funções principais supracitadas, diferentes variações da busca em profundidade (DFS) foram implementadas.

2. Implementação

2.1. Entrada e saída de dados

A entrada esperada é feita por meio de um arquivo de texto, passado como argumento na execução do programa. Esse arquivo deve ter a seguinte estrutura: na primeira linha, três números inteiros N, M e I, indicando respectivamente o número de membros no time (vértices do grafo), o número de relações diretas (arestas do grafo) e o número de instruções.

Já a segunda linha é dada por N inteiros, cada um representando a idade dos respectivos membros. As M linhas posteriores representam uma relação de comando entre dois membros A e B, e as próximas I linhas se referem às instruções a serem executadas, podendo ser do tipo: S, instrução swap entre X e Y, C, instrução commander a partir de C e M, instrução meeting.

A saída gerada pelo programa é impressa no $stdout$ e se refere às I instruções. Para cada instrução C, é impresso um número Z, que representa a idade da pessoa mais jovem que comanda o membro em questão, direta ou indiretamente. Para cada instrução S, é impresso T caso a troca tenha sido feita ou N caso contrário, seja pela criação de um ciclo ou pela não existência de uma relação direta entre os dois membros. Por fim, para cada instrução M são impressos N números, que representam uma possível ordem de fala na reunião.

2.2. Estruturas de dados

Para a modelagem do problema, foi empregado um grafo $G(V, A)$, em que os vértices V são os membros da equipe e as arestas A são as relações diretas de comandante e comandado.

Para tal, foi implementada uma classe que conta com três atributos: a quantidade de membros no time, um array de tamanho N com as respectivas idades dos membros e N listas de adjacência, cada uma armazenando as relações em que o membro em questão comanda outro membro.

2.3. Algoritmo

Para a implementação das três principais instruções requisitadas, foram desenvolvidas diferentes abordagens do algoritmo de busca em profundidade (DFS), detalhados abaixo. Além disso, métodos para construção, destruição, inserção de aresta e get e set da idade do membro foram criados.

A instrução swap conta com 4 métodos para o correto funcionamento. O primeiro recebe dois parâmetros inteiros relativos ao membro A e B a ter seu relacionamento invertido e, primeiramente, checa se existe uma aresta entre os mesmos, trocando se possível. Essa operação é feita por um método auxiliar, que retorna um atributo booleano informando a respeito da possibilidade da troca.

Posteriormente, se a troca foi feita, é invocado um método auxiliar que detecta ciclos no grafo utilizando uma variação do algoritmo DFS. Após declarar um array de cores de tamanho N membros, começando todos com branco (ou seja, o vértice ainda não foi processado), o método percorre os membros não visitados e chama um quarto método auxiliar. Este é responsável por, primeiramente, colorir o vértice de cinza (ou seja, em processamento) e percorrer todos seus vizinhos, isto é, os vértices adjacentes ao atual, chamando a si próprio de maneira recursiva. Caso, durante a execução algum vértice encontre outro vértice cinza, detectamos um ciclo no grafo e devemos reverter a operação de troca de arestas. Caso contrário, no fim, todos os vértices serão coloridos de preto (fim do processamento) e o grafo não tem ciclo.

Já para a instrução Commander, dois métodos foram implementados: o primeiro deles retorna o grafo invertido a partir do original, para facilitar a execução do DFS a fim de encontrar a idade da pessoa mais jovem que comanda A, direta ou indiretamente. Posteriormente, com o grafo já invertido, o método commander é invocado, executando uma variação iterativa do DFS (utilizando uma stack para simular a pilha de recursão), armazenando na variável $minAge$ a menor idade encontrada até o momento. Caso o membro em questão não seja comandado por ninguém, o método retorna -1, o que é um indicativo para a saída $C *$.

Por fim, para a instrução Meeting, outros dois métodos foram desenvolvidos: o primeiro cria uma pilha, a ser impressa no fim, armazenando a ordem topológica (possível ordem de fala dos membros) e um array booleano, dizendo se o membro já foi ou não visitado. Posteriormente, percorrendo todos os membros, é chamado um método auxiliar caso aquele membro ainda não tenha sido visitado. Nesse método auxiliar, o membro atual é marcado como visitado e é feita a checagem se os vizinhos já foram visitados, chamando recursivamente o mesmo método caso negativo, adicionando o membro à pilha

após a chamada. Dessa forma, após o processamento de todos os vértices do grafo, é retornada a pilha com uma possível ordenação topológica.

Não obstante, no arquivo main.cpp há a implementação de um procedimento que lê o conteúdo do arquivo de texto passado como parâmetro e invoca os métodos em questão da classe TeamGraph, fazendo a impressão da saída no $stdout$.

2.4. Instruções de compilação e execução

O compilador utilizado para o desenvolvimento do programa foi o g++/gcc $8.3.0$, usando a flag $-std = c$++11 numa máquina com a versão 4.19.49 do Kernel Linux. Para executar o programa, ao entrar em seu diretório, no terminal, deve ser digitado o comando $make$. Com isso, será gerado um binário $tp1$ e o algoritmo pode ser executado passando um arquivo de texto como argumento, por exemplo: $./tp1\ entrada.txt$

3. Análise de Complexidade

A complexidade de tempo do algoritmo será $O(V + A)$, conforme mostrado abaixo, analisando todas as três principais instruções. Já no que se refere à complexidade de espaço, temos que, como armazenamos um array de tamanho V com as idades de todos os membros do time e V listas de adjacências, que somadas tem a quantidade A de arestas do grafo (relações de comando entre os membros), a complexidade de espaço da criação do grafo principal é $O(V) + O(V + A) = O(V + A)$.

Além disso, na instrução Meeting, é instanciado um novo grafo, com todas as suas arestas invertidas, o que tem complexidade de espaço $O(V + A)$, além da pilha utilizada na instrução Commander e Meeting, que tem complexidade $O(V)$, pois guardará, no máximo, todos os vértices do grafo. Dessa forma, temos que a complexidade final de espaço será: $O(V) + O(V + A) + O(V + A) + O(V) = O(V + A)$, assim como a complexidade de tempo.

3.1. Instrução Swap

Na instrução Swap, para checar a existência de uma ligação entre A e B e, se possível, fazer sua inversão, temos uma complexidade de $O(V)$, pois no pior caso o vértice A é ligado a todos os outros vértices do grafo e B é o último elemento na lista de adjacências.

Posteriormente, é executada uma DFS procurando por ciclos no grafo. Para tal, o algoritmo deve iniciar em um vértice em específico, checar todas as suas arestas e visitar os outros vértices recursivamente. Dessa forma, o mesmo visita todos os vértices e passa por todas as arestas, resultando em uma complexidade de tempo $O(V + A)$.

Por fim, no pior dos casos em que um ciclo é encontrado, a operação de inversão da aresta deve ser desfeita, o que tem complexidade $O(V)$, pois a lista de adjacências de B terá que ser percorrida até o fim para a remoção de A, e a inclusão de B na lista de adjacências de A terá custo constante $O(1)$. Dessa forma, temos que a complexidade da instrução será $O(V) + O(V + A) + O(V) = O(V + A)$.

3.2. Instrução Commander

Na instrução commander, primeiramente devemos inverter todas as arestas do grafo, o que tem complexidade $O(A)$, já que cada uma deve ser percorrida exatamente uma vez

e alterada na respectiva lista de adjacências. Posteriormente, é feita a execução de uma busca em profundidade, algoritmo que percorre todos os vértices do grafo e, para cada vértice, percorre todos os vizinhos que ainda não foram visitados, isto é, percorre exatamente uma vez todos os vértices e todas as arestas do grafo, resultando na complexidade $O(V + A)$.

Além disso, como a variação da DFS implementada é iterativa, foi necessária a utilização de uma pilha, mas a inserção e remoção dos membros na mesma tem custo constante $O(1)$, o que faz com que ainda assim tenhamos que a complexidade da instrução sendo $O(V + A)$.

3.3. Instrução Meeting

Por fim temos que, na instrução meeting, é executada uma DFS, adicionando os membros a uma pilha, que armazenará a ordem topológica a ser impressa no fim do processamento. Conforme citado na instrução commander, as operações na pilha tem custo constante $O(1)$ e, portanto, a ordem de complexidade da instrução será dada unicamente pela execução da DFS.

Dessa forma, pela necessidade de se percorrer todos os vértices e, a partir desses vértices, percorrer as arestas do grafo, adicionando na pilha quando um vértice terminar de ser processado, temos que a complexidade da instrução será $O(V + A)$.

4. Avaliação Experimental

Para a análise experimental, foram gerados diversos casos de teste, com vértices variando de 5 até 100, adicionando de 5 em 5 e tendo o número máximo de arestas possível, desde que não formemos ciclos. Além disso, cada execução rodou nove instruções, simulando uma execução normal: três commanders, três swaps e três meetings.

A fim de evitar a interferência devido à troca de contexto do processador junto ao sistema operacional, cada teste foi executado 1000 vezes e, a partir dos tempos, calculou-se a média e o desvio padrão.

Apoiando-se nesses dados, gerou-se o seguinte gráfico:

Como o esperado, vemos um crescimento linear em relação à soma de vértices e arestas do grafo, o que corrobora a complexidade de tempo encontrada na seção anterior $O(V + A)$.

4.1. Perguntas e Respostas

4.1.1. Por que o grafo tem que ser dirigido?

O grafo tem que ser dirigido devido à natureza do problema. Por se tratar de uma hierarquia, A comandar B não é o mesmo que dizer que B comanda A e, portanto, o grafo deve ser dirigido.

4.1.2. O grafo pode ter ciclos?

Não. Caso houvesse um ciclo no grafo, teríamos que uma pessoa A comandaria B e, de alguma forma, B também comandaria A, direta ou indiretamente, por exemplo: A comanda B, B comanda C e C comanda A, o que gera um ciclo e uma quebra na hierarquia. Não obstante, devido à necessidade da existência de uma ordem topológica (ordem de fala segundo a instrução meeting), esse grafo não pode admitir ciclos.

4.1.3. O grafo pode ser uma árvore? O grafo necessariamente é uma árvore?

O grafo pode ser uma árvore, mas não necessariamente é uma árvore, dada a possibilidade de que uma pessoa C seja diretamente comandada por outras duas pessoas, A e B, o que não faz jus à estrutura de uma árvore.

5. Conclusão

O desenvolvimento do algoritmo proposto foi bastante interessante para colocar em prática os conceitos e algoritmos vistos em sala de aula a respeito de grafos.

A maior dificuldade encontrada foi na implementação das diferentes variações do algoritmo de busca em profundidade, sobretudo as variações recursivas, devido à necessidade de manipulação de alguns ponteiros de arrays que armazenavam informações a respeito do processamento dos vértices.

Isso posto, o trabalho mostrou-se muito interessante para praticar a lógica de programação, o desenvolvimento na linguagem C++ e aumentar a experiência com manipulação de grafos e algoritmos de busca no mesmo.

6. Referências Bibliográficas

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida:Algoritmos – Teoria e Prática 3a. Edição, Elsevier, 2012.

MS LATHA SHILVANTH. Java: For Programming (2018). Createspace independent Publishing Platform. ISBN-10: 1985254603