UNIVERSIDADE FEDERAL DE MINAS GERAIS – UFMG
GRADUAÇÃO EM ENGENHARIA ELÉTRICA, 2° PERÍODO
Algoritmos e Estruturas de Dados II – TA2
Profª Gisele L. Pappa

Relatório - Trabalho Prático 02

Árvore Geradora Mínima

Nander Santos do Carmo - 2018019931

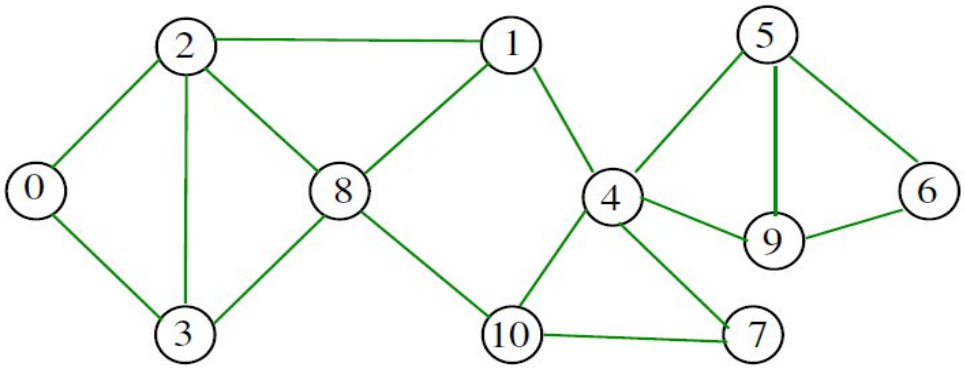
06 de novembro de 2018

Belo Horizonte

Sumário

Introdução

A teoria dos grafos é uma área voltada para o estudo de objetos dentro um determinado conjunto. Esse conjunto é formado por *vértices,* geralmente simbolizados como pontos ou círculos, e arestas, que representam a relação entre dois vértices e que são representadas através de linhas que ligam esses pontos. Dessa forma, de acordo com a plicação pode-se definir dois tipos básicos de arestas uni e bidirecionais, se uma aresta pode conectar um vértice a ele mesmo, e se haverão Dependendo da aplicação, arestas podem ou não ter direção, pode ser permitido ou não arestas ligarem um vértice a ele próprio e se poderão existir ciclos no grafo. Assim, este trabalho visa aproveitar o conceito de grafos para otimizar uma situação hipotética em que se pretende ligar computadores no Departamento de Ciências da Computação da UFMG (DCC) com o mínimo de cabo possível. Tudo isso através do conceito de árvore geradora mínima de um grafo e do Algoritmo de Krushkal.

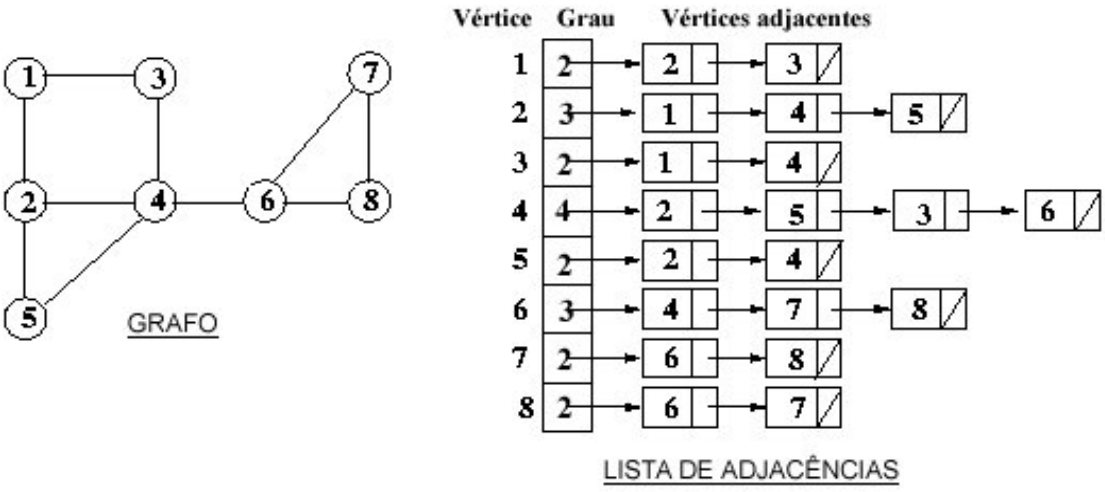


Implementação

Estrutura de Dados

Para a implementação deste trabalho prático foi criado um Tipo Abstrato de Dados composto, chamado "*grafo*" . Esse TAD implementado se trada da junção de um TAD Lista com um TAD lista de Adjacência. Assim, dentro do mesmo, temos as declarações de funções tradicionais para listas adaptadas para o uso necessário e funções relacionadas à manipulação de grafos. No caso deste projeto, para fins de simplificação, o tipo Lista de Adjacência foi representado apenas como Grafo, por entender que a Lista de Adjacência é apenas uma maneira de representar um grafo.

O tipo *Grafo* nada mais é do que um vetor de listas, em que cada posição do vetor armazena, dentre outras coisas, um ponteiro para uma lista encadeada, como pode ser representado na imagem abaixo:



Funções relacionadas ao tipo Lista

- ✓ void FazListaVazia(Lista*);
- ✓ int TestaListaVazia(Lista*);
- ✓ void InsereLista(Celula*, Grafo*, int);
- ✓ void MontaLista (Lista*, TipoItem);
- ✓ void RetiraLista(Apontador, Apontador, Lista*);
- ✓ void ImprimeLista(Lista*);
- ✓ Lista OrdenaLista(Lista*);
- ✓ void ImprimeListaArquivo(Lista*, FILE*);

- void FazListaVazia(Lista*):

  Essa função recebe um ponteiro para um tipo Lista e aloca a primeira célula (célula cabeça) da lista encadeada. Essa função não retorna nada.

- int TestaListaVazia(Lista*):

  Essa função recebe como referência, para fins de economia de memória, uma lista encadeada e verifica se está vazia ou se já foi preenchida com algum dado. Essa função retorna 0 quando não está vazia e 1 quando está.

- void InsereLista(Celula*, Grafo*, int):

  Essa função recebe o ponteiro para uma célula qualquer que se deseja inserir na lista (última posição), um grafo como referência e um inteiro que indica em qual posição do vetor Lista de adjacência está a lista na qual se deseja inserir a célula. Essa função não retorna nada.

- void MontaLista (Lista*, TipoItem):

  Essa função monta uma lista encadeada que foi criada. A função cria e aloca uma célula, insere na lista que foi passada como referência e diferentemente da função InsereLista, a função não mexe diretamente com grafos. Por escolha do programador foram criadas funções distintas de forma a simplificar a implementação do programa. Essa função não retorna nada.

- void RetiraLista(Apontador, Apontador, Lista*):

  Essa função recebe o ponteiro para uma lista e dois apontadores: um para a célula que se deseja retirar e outro para a célula anterior. Essa função não retorna nada.

- void ImprimeLista(Lista*):

  Essa função recebe, apenas para economia de memória, uma lista como referência e imprime na tela, célula por célula essa lista. Essa função não retorna nada.

- Lista OrdenaLista(Lista*):

  Essa função recebe um ponteiro para uma lista e ordena, em ordem crescente, da célula que possui o menor vértice vinculado até a maior. Essa função não retorna nada.

- void ImprimeListaArquivo(Lista*, FILE*):

  Essa função, assim como a ImprimeLista, recebe como referência uma lista, também para economia de memória e imprime, porém, em um arquivo cujo

endereço foi passado através do ponteiro para o tipo FILE que a função recebe como parâmetro. Essa função não retorna nada.

Funções relacionadas ao tipo Lista de Adjacência

- ✓ void DefineTamanhoGrafo(Grafo*, int);
- ✓ int TestaGrafoVazio(Grafo *grf);
- ✓ void InsereVerticeGrafo(Grafo*, int, int, int);
- ✓ void RetiraVerticeGrafo(Grafo*, int, int, int);
- ✓ void ImprimeGrafo(Grafo*);
- ✓ int VerificaGrafo(Grafo*, int, int);
- ✓ void GeraGrafoMinimo(Grafo*);
- ✓ void OrdenaGrafo(Grafo*);
- ✓ void ImprimeGrafoArquivo(Grafo*, FILE*);

- void DefineTamanhoGrafo(Grafo*, int):

  Essa função recebe um grafo como referência, um inteiro e determina o tamanho do grafo. Assim, ele aloca, dinamicamente, o vetor de listas que compõe a lista de adjacência. Essa função não retorna nada.

- int TestaGrafoVazio(Grafo *grf):

  Essa função recebe um ponteiro para o tipo grafo e testa se o grafo está vazio, ou seja, se todas as listas contidas no vetor estão vazias. Dessa forma, esta função chama a TestaListaVazia. Essa função retorna um inteiro: 1 caso o grafo esteja de fato vazio e 0 caso contrário.

- void InsereVerticeGrafo(Grafo*, int, int, int):

  Essa função recebe um grafo como referência e três outros inteiros como parâmetro, que representam os vértices de entrada e saída do grafo e o peso da aresta da célula que se deseja inserir em uma das listas do vetor. Assim, essa função chama a InsereLista. Essa função não retorna nada.

- void RetiraVerticeGrafo(Grafo*, int, int, int):

  Essa função recebe um ponteiro para o tipo grafo e três inteiros que representam as coordenadas do vértice que se deseja retirar do grafo. Assim, essa função chama a RetiraLista. Essa função não retorna nada.

- void ImprimeGrafo(Grafo*):

Essa função recebe, para fins de economia de memória, um grafo como referência e imprime na tela. Essa função chama internamente a ImprimeLista para realizar a impressão das listas internas do vetor de listas. Essa função não retorna nada.

- int VerificaGrafo(Grafo*, int, int):

  Essa função recebe como referência um grafo e dois inteiros, que representam os grafos de entrada e saída de uma aresta. Essa função foi criada para verificar se um conjunto de vértices cumpre as exigências do Algoritmo de Krushkal para saber se um vértice pode ou não ser inserido na árvores mínima. Essa função retorna um inteiro: 1 caso o vértice possa ser inserido, ou seja atenda as especificações do Algoritmo de Krushkal e 0 caso contrário.

- void GeraGrafoMinimo(Grafo*):

  Essa função é a equivalente ao Algoritmo de Krushkal. Recebe como referência um grafo original, cria um novo grafo internamente no qual insere apenas vértices que cumpram os seguintes critérios:

  Escolhe a menor aresta possível que conecte dois vértices sendo que um esteja já inserido na árvore mínimo e outro que não esteja.

  Assim, essa função chama internamente a VerificaGrafo. Em seguida o grafo original é substituído pelo novo gerado pelo algoritmo. Essa função não retorna nada.

- void OrdenaGrafo(Grafo*):

  Essa função recebe um grafo por referência e ordena, primeiramente, através da chamada da função OrdenaLista, as listas contidas no vetor, e por fim o próprio vetor de forma que o grafo fique primeiro ordenado de forma crescente em relação ao primeiro vértice e por fim ordenado de forma crescente em relação ao segundo vértice, respeitando a primeira ordenação. Essa função não retorna nada.

- void ImprimeGrafoArquivo(Grafo*, FILE*):

  Essa função realiza a impressão do gravo, dessa vez em um arquivo. Para a impressão das listas internas do vetor do grafo é realizada a chamada da função ImprimeGrafoArquivo. Essa função não retorna nada.

Programa Principal

O programa principal (main.c) recebe como parâmetros dois arquivos: um para leitura das entradas que construirão o grafo original, que será reduzido e outro de saída, no qual será impresso o grafo que representa a Árvore Geradora Mínima, produzida através do Algoritmo de Krushkal.

O programa abre, de acordo com o argumento *argv[1]* o arquivo passado para leitura, testa se foi possível abrir o arquivo e, em caso positivo, cria um grafo com o número de vértices lido da primeira linha do arquivo através da função DefineTamanhoGrafo do TAD *grafo* que foi linkado no programa.

Em seguida, o programa entra em um loop que será executado um determinado número de vezes, equivalente ao número de arestas lido também na primeira linha do arquivo. Nesse loop, será lida toda linha, sendo que os valores lidos serão enviados através da função InsereVerticeGrafo para que os vértices sejam inseridos no grafo criado. Na sequência, o arquivo de leitura é fechado. Após isso são chamadas as funções GeraGrafoMinimo, OrdenaGrafo e ImprimeGrafo, que realizarão a geração da árvores mínima do grafo, ordenarão o novo grafo gerado e imprimirão na tela para consulta rápida.

A seguir o arquivo de saída é aberto, o programa testa se houve sucesso para abrir esse arquivo e, em caso tenha havido, o programa chama a função ImprimeGrafoArquivo para realizar a impressão do grafo já reduzido. O programa também realiza um teste para ver se foram passados o número correto de parâmetros para a função.

Detalhes Técnicos e Tomadas de decisão

Foram criados 3 arquivos principais: grafo .c e grafo .h que representam o TAD criado, lembrando que por opção do aluno os TAD's Lista e Grafo (Lista de ajacência foram unidos em um) arquivo principal do programa, main.c. O compilador utilizado foi o *GNU GCC Compiler* vinculado ao editor de texto *Atom.* O projeto foi desenvolvido e testado em Linux (Distribuição Ubuntu LTS 18.04).

Os teste de consistência internos realizados pelo programa abrangem somente erros de passagem de arquivo como parâmetro e número incorreto de parâmetros para a função. Assim, a precisão da saída está relacionada diretamente à disponibilização de dados no formato certo. Do tipo:

Exemplo de entrada aceita:

```
9 13
0 1 4
0 7 8
1 2 8
2 3 7
2 5 4
2 8 2
3 4 9
5 6 2
3 5 14
6 7 1
4 5 10
6 8 6
7 8 7
```

Saída:

```
9 8
0 1 4
0 7 8
2 3 7
2 5 4
2 8 2
3 4 9
5 6 2
6 7 1
```

Análise de Complexidade

A ordem de complexidade para cada função implementada no projeto e no TAD serão analisadas em função de um número *n* que simboliza o tamanho do grafo (número de vértices, ou posições do vetor) e *m* que representa o tamanho da lista de cada posição do vetor desse grafo:

TAD - Parte referente à Lista :

- void FazListaVazia(Lista*):

  Essa função realiza apenas uma operação de comapração. Assim, a função é O(1).

- int TestaListaVazia(Lista*):

  Essa função realiza apenas operações O(1) de atribuição. Assim, a função é O(1).

- void InsereLista(Celula*, Grafo*, int):

  Essa função exerce apenas operações de atribuição, que são de O(1). Assim, a função é O(1).

- void MontaLista (Lista*, TipoItem):

  Essa função executa apenas operações O(1) de atribuição. Logo a função é O(1).

- void RetiraLista(Apontador, Apontador, Lista*):

  Essa função reliza apenas operações O(1), como atribuições e comparações. Logo a função é O(1).

- void ImprimeLista(Lista*):

  Essa função executa algumas operações de atribuição e comparação que são O(1) e possui um loop executado *m* vezes. Assim, a função é O(*m*).

- Lista OrdenaLista(Lista*):

  Essa função possui algumas operações O(1), chama as funções FazListaVazia e TestaListaVazia que também são O(1) e possui um loop executado *m* vezes. Logo a função é O(*m*).

- void ImprimeListaArquivo(Lista*, FILE*):

Essa função também possui alguns comandos O(1) e um loop executado *m* vezes. Assim, a função é O(*m*).

TAD - Parte referente ao Grafo (Lista de Adjacência):

- void DefineTamanhoGrafo(Grafo*, int):

  Essa função possui algumas operações O(1), chama a função FazListaVazia que também é O(1) e possui dois loops executados *n* vezes. Assim, a função é O(n).

- int TestaGrafoVazio(Grafo *grf):

  Essa função possui um loop executado *n* vezes contendo comandos O(1). Assim, a função é O(*n).*

- void InsereVerticeGrafo(Grafo*, int, int, int):

  Essa função possui alguns comandos O(1), chama a função InsereLista que também é O(1) e, por fim, apresenta um loop executado *n* vezes, sendo assim, O(*n*).

- void RetiraVerticeGrafo(Grafo*, int, int, int):

  Essa função possui um loop executado *m* vezes, algumas funções O(1), além de chamar a função RetiraLista que também é O(1). Logo a função é O(*m*).

- void ImprimeGrafo(Grafo*):

  Além das funções O(1), essa função apresenta um loop executado *n* vezes, dentro do qual é chamada a função ImprimeLista que é O(*m*). Assim, a função é O(*mn*).

- int VerificaGrafo(Grafo*, int, int):

  Essa função apresenta, além de operações O(1) de atribuição e comparação, um loop executado *n* vezes, dentro do qual existe outro loop executado *m* vezes. Logo a função é O(*mn*).

- void GeraGrafoMinimo(Grafo*):

  Essa função apresenta várias operações O(1) de atribuição e comparação, além de chamar as funções InsereVerticeGrafo e RetiraVertieGrafo, que são O(*n*) e O(*m*), respectivamente. Além disso, dentro de um loop executado *m* vezes é chamada a função VerificaGrafo que é O(*mn*). Assim, essa função é O(m^2n).

- void OrdenaGrafo(Grafo*):

  Além dos comandos O(1), essa função apresenta um loop executado *n* vezes, no qual é chamado a função OrdenaLista que é O(*m*) e mais uma sequência de comandos equivalentes ao método de ordenação por *Inserção* que é O(n^2). Assim, dependendo do valor de *m* e *n* a função pode ser O(n^2) ou O(*mn*).

- void ImprimeGrafoArquivo(Grafo*, FILE*):

  Essa função apresenta algumas operações O(1) e um loop executado *n* vezes dentro do qual a função ImprimeListaArquivo, que é O(*m*), é chamada. Assim, a função é O(*mn*).

Projeto Principal:

O programa principal possui, além de funções O(1), chamadas de funções do TAD, no caso: DefineTamanhoGrafo, InsereVerticeGrafo, GeraGrafoMinimo, OrdenaGrafo, ImprimeGrafo, ImprimeGrafoArquivo. Como visto na análise de complexidade do TAD grafo realizada acima a função de maior ordem e que por consequência é responsável por definir a ordem de complexidade total do programa é a função GeraGrafoMinimo que é O(m^2n) . Assim, a função principal é O(m^2n)

Conclusão e Anexos

Após a implementação desse trabalho o entendimento sobre a construção, manipulação e utilidade dos Grafos foi consolidada. Conceitos como listas encadeadas, alocação dinâmica, criação de TAD's, leitura e impressão de arquivos puderam ser massivamente praticados, sendo que as principais dificuldades encontradas foram durante a criação das funções do TAD.

Serão enviados os arquivos: grafo.h, grafo .c, main.c (função principal) .