



Aluno: Vinícius de Oliveira Silva

Curso: Ciência da Computação

Disciplina: Algoritmos e Estruturas de Dados II

Documentação de Trabalho Prático

1. Introdução

Grafos são estruturas de dados amplamente utilizadas em Ciência da Computação. Um grafo é um modelo de representação que permite visualizar e compreender as relações de objetos de um determinado conjunto.

O objetivo desse trabalho é implementar um algoritmo capaz de determinar a Árvore Geradora Mínima de um grafo, que é o conjunto de arestas capaz de conectar todos os vértices de um grafo, de forma que o somatório dos pesos de cada aresta seja o menor possível.

Deverão ser implementado tipos abstratos de dados (TAD's) para representar grafos de maneira computacional e realizar operações sobre eles, tais como, inserir arestas, remover arestas, consultar existência de vértices, etc.

Espera-se com isso, praticar técnicas de programação que permitam a inserção e manipulação de dados no "heap".

2. Implementação

Estruturas de Dados:

Para a implementação desse trabalho, foram criados dois tipos abstratos de dados:

- Lista
- Grafo

O TAD Lista é usado como componente ("atributo") do TAD Grafo, e é usado para trabalhar com uma lista encadeada.

O TAD Grafo é a estrutura de dados principal do trabalho e é utilizado para representar um Grafo dentro do contexto do programa, possibilitando assim, a execução das operações necessárias.

Representação do TAD Lista:

Atributos:

- Apontador para Celula Primeiro;
- Apontador para Celula Ultimo;

O atributo Primeiro é um apontador usado para indicar qual é a primeira Célula armazenada na lista. É importante observar que as células são representadas os vértices do Grafo no contexto do programa. Uma Célula é composta por:

- Vértice (inteiro que representa o código de um dos vértices do Grafo);
- Peso (inteiro que representa o peso de uma das arestas do Grafo);
- Apontador para Célula (Apontando para a próxima Célula da lista);

O atributo Ultimo é um apontador usado para indicar qual é a última Célula armazenada na lista.

Representação do TAD Grafo:

Atributos:

- Array de Listas;
- Número de vértices;
- Número de arestas;

O array de Listas é utilizado para armazenar o conjunto de listas de adjacências dos vértices do grafo.

Cada posição do array armazena a lista de adjacência do vértice correspondente aquela posição. A posição 3 do array por exemplo, armazena a lista de adjacência do vértice de código 3.

O atributo Número de vértices é um inteiro que indica quantos vértices o grafo tem.

O atributo Número de arestas é um inteiro que representa quantas arestas o grafo tem.

Funções e Procedimentos:

TAD Lista:

void fazListaVazia (Lista *list): recebe uma lista via ponteiros e a inicializa, alocando memória para o seu apontador “Primeiro”.

int vazia(Lista list): recebe uma lista via ponteiros e testa se a mesma encontra-se vazia, retornando 1 em caso afirmativo e 0 em caso negativo.

void insere(Celula cell, Lista *list): recebe uma Celula (por valor), e a insere na lista especificada via ponteiro. É importante observar que a inserção de células é feita de maneira ordenada, tendo como critério de ordenação, o código dos vértices.

void retira(Pointer p, Lista *list, Vertice *vertex): retira o elemento seguinte ao apontado por “p” da lista (“list”) e o armazena no vértice apontado por “*vertex”.

É importante observar que essa função não é utilizada na implementação desse trabalho, mas mesmo assim está presente em seu código devido ao fato de sua definição fazer parte do conceito de lista encadeada.

void imprime(Lista list): imprime a lista especificada por parâmetro na tela.

TAD Grafo:

void leGrafo(Grafo *graph): recebe um grafo via ponteiros e faz a leitura das arestas desse grafo. Essa função chama a função *scanf* repetidas vezes para primeiramente ler o numero de vértices e arestas do grafo e em seguida, ler os seus valores.

void fazGrafoVazio(Grafo *graph, int NumVertices, int NumArestas): inicializa um novo grafo, alocando memória para as listas de adjacências de todos os seus vértices. Cada lista de adjacência é inicializada vazia.

void insereAresta(Vertice *v1, Vertice *v2, int Peso, Grafo *graph): insere a aresta formada pelos vértices “v1” e “v2” com peso “Peso” no Grafo “graph”. Para fazer isso, a função inclui “v2” na lista de adjacência de “v1”, e inclui também “v1” na lista de adjacência de “v2”. Essas operações são feitas através de chamadas à função “insere” do TAD Lista.

int existeAresta(Vertex v1, Vertex v2, Grafo graph): recebe por parâmetro dois vértices e verifica se existe uma aresta entre esses dois vértices no Grafo graph, que também é recebido por parâmetro.

int listaAdjVazia(Vertex v1, Grafo graph): função que recebe um vértice e um grafo por parâmetro e verifica se a lista de adjacência desse vértice é vazia naquele grafo. Em termos práticos, essa função é utilizada para saber se um vértice existe num determinado grafo. O valor retornado é 1 caso a lista de adjacência seja vazia ou 0 caso não seja.

Pointer primeiroListaAdj(Vertex v1, Grafo graph): função que retorna um ponteiro que aponta para o primeiro vértice válido da lista de adjacência de “v1” (recebido por parâmetro) no Grafo “graph” (também recebido por parâmetro). Essa função não é utilizada nesse trabalho, mas está disponível no TAD grafo pois pode ser útil em caso de reaproveitamento de código.

void retiraAresta(Vertex v1, Vertex v2, int Peso, Grafo *graph): função que pesquisa no Grafo “graph” (recebido por ponteiro) uma aresta entre os vértices “v1” e “v2” (recebidos por parâmetro). Caso a aresta seja encontrada, a mesma é removida do grafo através do uso da função retira do TAD lista.

void arvoreMinima(Grafo source, Grafo *destination): função responsável por determinar a árvore geradora mínima do grafo “source” (recebido por parâmetro) e armazená-la no grafo “destination” (recebido por ponteiro). Essa função chama a função *escolheAresta* (interna ao TAD) repetidas vezes para montar a árvore desejada.

void escolheAresta(Grafo graphG, Grafo *graphA, Vertex *v, Vertex *u, int *peso): essa é uma função auxiliar e interna ao TAD Grafo, não sendo disponibilizada em sua interface. A função tem como objetivo determinar a menor aresta possível do Grafo “graphG” tal que um dos vértices que a compõem esteja contido no “graphA” (recebido por ponteiro) e o outro não. Os parâmetros “v”, “u” e “peso” vem vazios para a função e só são úteis para permitir que a função retorne mais de um valor. “v” é “retornado” contendo um vértice pertencente a “graphA”, “u” é “retornado” contendo um vértice pertencente apenas a “graphG”, e “peso” é retornado contendo o menor peso possível de alguma aresta que satisfaça as condições anteriores.

void liberaGrafo(Grafo *graph): função utilizada para liberar a memória ocupada pelo Grafo “graph” (recebido por ponteiro).

void imprimeOrdenado(Grafo graph): função responsável por imprimir um grafo conforme as especificações fornecidas no enunciado do trabalho. A impressão é feita de maneira ordenada, usando como critérios de ordenação os códigos dos vértices que compõem as arestas do grafo.

void imprimeGrafo(Grafo graph): função que imprime um grafo conforme os padrões fornecidos por [1]. Essa função não é utilizada nesse trabalho, mas está disponível no TAD Grafo pois pode ser útil em caso de reaproveitamento do TAD em outros contextos.

Programa Principal:

O programa principal cria uma variável do tipo Grafo e em seguida, chama a função *leGrafo* para preencher a mesma. Uma outra variável do tipo Grafo é então criada para representar a árvore geradora mínima do grafo anteriormente criado. A função *arvoreMinima* é chamada e gera a árvore mínima para o primeiro grafo e a armazena no segundo.

O segundo grafo é então impresso através da função *imprimeOrdenado*.

Toda a entrada de dados é lida de stdin e impressa em stdout.

É importante também observarmos que em momento algum, os atributos do TAD Grafo são acessados no programa principal. Apenas as funções enunciadas na interface do TAD são acessadas.

Organização do Código, Decisões de Implementação e Detalhes Técnicos:

O código está dividido em 5 arquivos principais. Os arquivos *Lista.c* e *Lista.h* implementam o TAD lista, já os arquivos *Grafo.c* e *Grafo.h* implementam o TAD Grafo, enquanto o arquivo *TP1.c* implementa o programa principal.

Uma importante decisão de implementação que foi tomada durante a implementação desse trabalho, foi a criação da função *escolheAresta*, dedicada à tarefa de selecionar qual será a próxima aresta a ser inclusa na árvore geradora mínima. Essa função foi criada com o intuito de simplificar o desenvolvimento da função *arvoreMinima*, fornecendo assim, mais clareza ao código.

O compilador utilizado foi o MacOSX Gcc através da IDE Eclipse Kepler, no sistema operacional Mac OSX 10.9.2 (Mavericks).

3. Análise de Complexidade

A análise de complexidade será feita em função da variável n que representa o número de vértices do Grafo de entrada.

TAD Lista:

Função fazListaVazia: como a função apenas executa comandos $O(1)$, e não há loops, podemos dizer que sua Ordem de complexidade é $O(1)$.

Função vazia: como a função apenas executa comandos $O(1)$, e não há loops, podemos dizer que sua Ordem de complexidade é $O(1)$.

Função insere: a complexidade dessa função é $O(n)$ no pior caso (quando se quer inserir um vértice cujo código é maior do que o de todos os vértices já inseridos) e $O(1)$ no melhor caso (quando se quer inserir um vértice cujo código é menor do que o de todos os vértices já inseridos).

Função retira: como a função recebe um ponteiro apontando para o elemento anterior ao elemento que deseja-se remover da lista, podemos dizer que sua complexidade é $O(1)$. É importante notar que não é necessário que se faça uma pesquisa de elementos na lista.

Função imprime: como cada elemento da lista deve ser impresso, concluímos que a Ordem de complexidade dessa função é $O(n)$.

TAD Grafo:

Função leGrafo: Dentro dessa função, é executado um loop para ler cada uma das arestas do Grafo. Como o grafo não pode ter mais de uma aresta conectando os mesmos vértices, concluímos que a complexidade da função é $O(n)$ (caso de um grafo completo).

Função fazGrafoVazio: Essa função aloca memória para a lista de adjacência de cada um dos vértices do grafo, portanto, podemos dizer que sua complexidade é $O(n)$.

Função insereAresta: Podemos dizer que a complexidade dessa função é $O(2n)$ pois ela faz duas chamadas à função *insere* do TAD Lista, que por sua vez, é $O(n)$.

Função existeAresta: A complexidade dessa função é $O(n)$, já que ela precisa percorrer o grafo a fim de encontrar a aresta que conecta os dois vértices recebidos por parâmetro.

Função listaAdjVazia: Dentro dessa função, é feito um teste com o objetivo de saber se a lista de adjacência de um único vértice está vazia. O teste é feito através de uma única chamada à função *vazia* do TAD Lista (que é $O(1)$), portanto, podemos dizer que a complexidade dessa função também é $O(1)$.

Função primeiroListaAdj: Como essa função apenas executa comandos $O(1)$ e não existem loops ou chamadas recursivas, concluímos que sua complexidade é $O(1)$.

Função retiraAresta: Como a função precisa fazer uma pesquisa vértice a vértice no grafo até encontrar a aresta que deve ser removida, podemos concluir que a complexidade dessa função é $O(n)$.

Função arvoreMinima: Dentro dessa função são executadas n chamadas da função *escolheAresta* e por isso, a sua complexidade é dada por $n \cdot (n^2 - n)$, que é igual a $n^3 - n^2$.

Função escolheAresta: No pior caso, quando cada vértice está conectado a todos os outros, a complexidade dessa função é dada por $n \cdot (n-1)$, que é igual a $n^2 - n$. Chegamos a essa conclusão baseando no fato de que, nesse caso, a função deve percorrer toda a lista de adjacência de todos os vértices. É importante ressaltar que um vértice nunca pode estar conectado a ele mesmo, e é esse o motivo de a complexidade da função não ser $O(n^2)$.

Função liberaGrafo: A função deve percorrer toda a lista de adjacência de todos os vértices, liberando a memória ocupada por cada elemento da lista. Como um vértice nunca está conectado a ele mesmo, podemos dizer que a complexidade dessa função é $n \cdot (n-1)$, que é igual a $n^2 - n$.

Função imprimeOrdenado: Mais uma vez, a função deve percorrer toda a lista de adjacência de todos os vértices do grafo, portanto sua complexidade é da ordem de $n^2 - n$.

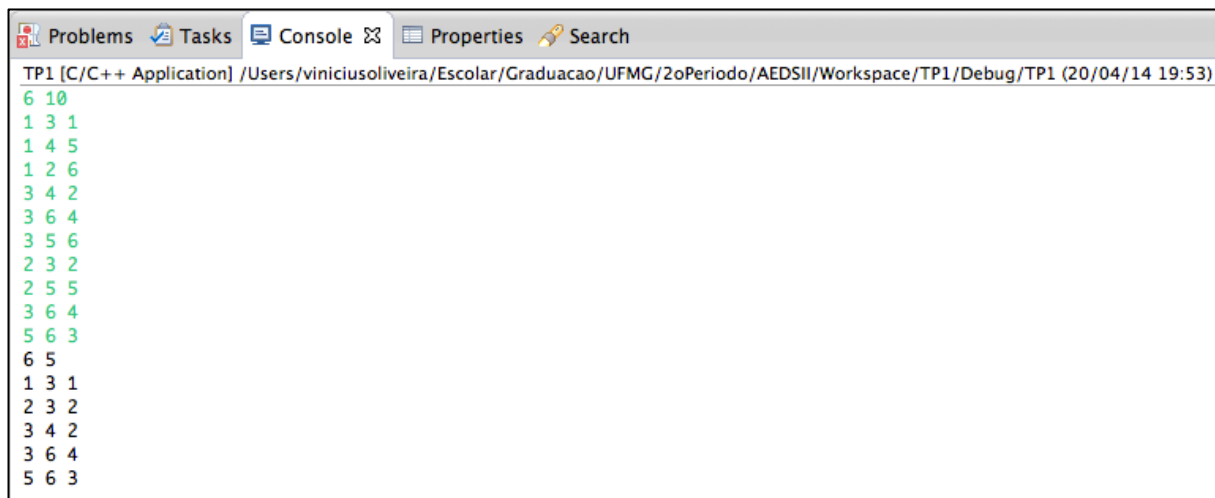
Função imprimeGrafo: Devido ao fato de essa função ser muito parecida com a função *imprimeOrdenado*, podemos concluir que sua complexidade é a mesma, $n^2 - n$.

4. Testes

Inúmeros testes foram realizados com o programa no intuito de avaliar seu comportamento em diversas situações específicas, tais como inserção de arestas repetidas, inserção de arestas com pesos iguais, entrada não ordenada, etc. Todos os testes foram bem sucedidos e a saída gerada corresponde à saída esperada.

Os testes foram executados em um computador equipado com processador Core 2 Duo @2.4GHz e 4GB de memória. A saída abaixo mostra uma execução típica do programa:

Console do Eclipse executando o trabalho prático:



```
TP1 [C/C++ Application] /Users/viniciusoliveira/Escolar/Graduacao/UFGM/2oPeriodo/AEDSII/Workspace/TP1/Debug/TP1 (20/04/14 19:53)
6 10
1 3 1
1 4 5
1 2 6
3 4 2
3 6 4
3 5 6
2 3 2
2 5 5
3 6 4
5 6 3
6 5
1 3 1
2 3 2
3 4 2
3 6 4
5 6 3
```

Obs.: Os caracteres em verde representam a entrada do Trabalho e os em preto representam a saída.

5. Conclusão

A implementação desse trabalho foi mais longa e custosa do que o esperado devido a pequenos erros de programação que demoraram tempo demasiado longo até serem detectados e corrigidos, visto que poucos exemplos válidos de entrada e saída estavam disponíveis. Após a identificação desses pequenos erros, no entanto, a implementação do algoritmo transcorreu de maneira satisfatória.

6. Referências

[1] Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning.