TP 2 - Árvore Geradora Mínima

DCC004 - ALGORITMOS E ESTRUTURAS DE DADOS II - TN

Yuri Diego Santos Niitsuma | UFMG | April 27, 2014

Sumário

Introdução

A teoria dos grafos é um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto.

O objetivo desse trabalho é, de um grafo não direcionado, encontrar a árvore geradora mínima utilizando o algoritmo de Prim a partir de entradas.

Espera-se com isso praticar os conceitos básicos de programação utilizando lista de adjacência.

Implementação

Estrutura de Dados

Para a implementação do trabalho foi criado um Tipo Abstrato de Dados Grafo utilizando lista de adjacência retirada do livro Projeto de Algoritmos (Nivio Ziviani), com as diferença que o vetor da lista de adjacência é dinamicamente alocado.

Para utilizar o algoritmo de Prim foi criado um TAD AGM (Árvore Geradora Mínima) do qual utiliza a estrutura de dados do TAD Grafo.

Funções e Procedimentos

O TAD Grafo criado possui as seguintes funções:

void FGVazio (TipoGrafo &Grafo): recebe um grafo por referência e gera apenas os vértices, nenhuma aresta.

void InsereAresta (TipoGrafo &Grafo, int &V1, int &V2, int &iPeso): recebe o grafo, os dois vértices e o peso da aresta por referência e adiciona a aresta no grafo de forma já ordenada pelos vértices na lista.

void Imprime(TipoGrafo *Grafo): imprime todas as arestas com o peso relacionado entre os vértices dois a dois da forma crescente "$v_i\ v_j\ peso$" tal que $v_i \leq v_j$ para todo grafo com vértices enumeráveis.

O TAD Grafo criado possui as seguintes funções:

void ArvoreMin(TipoGrafo &Grafo, TipoGrafo &Grafo_min): função que recebe como referência o Grafo e o Grafo_min vazio e gera a árvore geradora mínima no Grafo_min utilizando o algoritmo de Prim.

void ImprimeMIN(TipoGrafo &Grafo): função para utilizar a impressão do TAD Grafo para o formato dos testes do Prático.

Programa Principal

O programa principal cria duas variáveis do tipo graf, Grafo e Grafo_min, e depois chama as várias funções do tipo abstrato de dados. O programa recebe como entrada inicial o número de vértices e o número de arestas do Grafo inicial e utilizando o TAD para gerar os grafos vazios.

Depois o programa entra no loop de número de vértices com entrada de dois vértices e o peso da aresta entre os vértices, utilizando o TAD Grafo são inseridos as arestas no grafo.

Após o processo o programa gera a árvore geradora mínima na variável Grafo_min e imprime todas as arestas.

Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código está dividido em cinco arquivos principais: Grafo.c e Grafo.h implementam o TAD Grafo, AGM.c e AGM,h implementam o TAD Árvore Geradora Mínima e main.c implementa o programa principal. O valor especificado para a constante MAX_VERT foi 0xFFFFFF, de forma a permitir diferentes testes com o programa. Para manipular melhor o TAD Grafo, eu resolvi criar e implementar a função respeitando o objetivo de utilizar a lista de adjacência na representação do grafo... O compilador utilizado foi o **Xcode 5.1(5B130a)** no sistema operacional **Mac OS X 10.9.2**. O compilador utilizado é o próprio **GNU/GCC** disponibilizado pelo sistema operacional.

Análise de Complexidade

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

TAD Grafo

Função FGVazio: a função FGVazio executa alguns comandos $O(1)$ e depois entra em um loop que é executado n vezes. Dentro desse loop, é feito uma atribuição $O(1)$. Logo: $O(1) + n \cdot O(1) = O(n)$.

Função InsereAresta: a função InsereAresta depende o valor adicionado no vértice com atribuições constantes $O(1)$ onde a complexidade depende do loop para inserir o valor ordenado

varrendo a lista das arestas na lista de adjacência.
Melhor caso: $O(1)$
Pior Caso: $O(n)$

Função Imprime: a função Imprime varre toda a lista de adjacência em todos os vértices relacionados e imprime em ordem crescente as arestas. Considerando atribuições e comparações como $O(1)$ temos um loop (while) dentro do outro onde o primeiro é $O(n)$ e o segundo é variável entre o melhor e o pior caso dependente do número de arestas no grafo.
Considerando no pio caso onde cada vértice é conectado com todos os vértices em um grafo de n vértices temos $\binom{n}{2}$ arestas. Assim a função imprime tem complexidade:
Melhor caso: $O(n)$ com o grafo completo de n-1 vértices.
Pior caso: $n \cdot O(n-1) = O(n^2)$.

TAD AGM

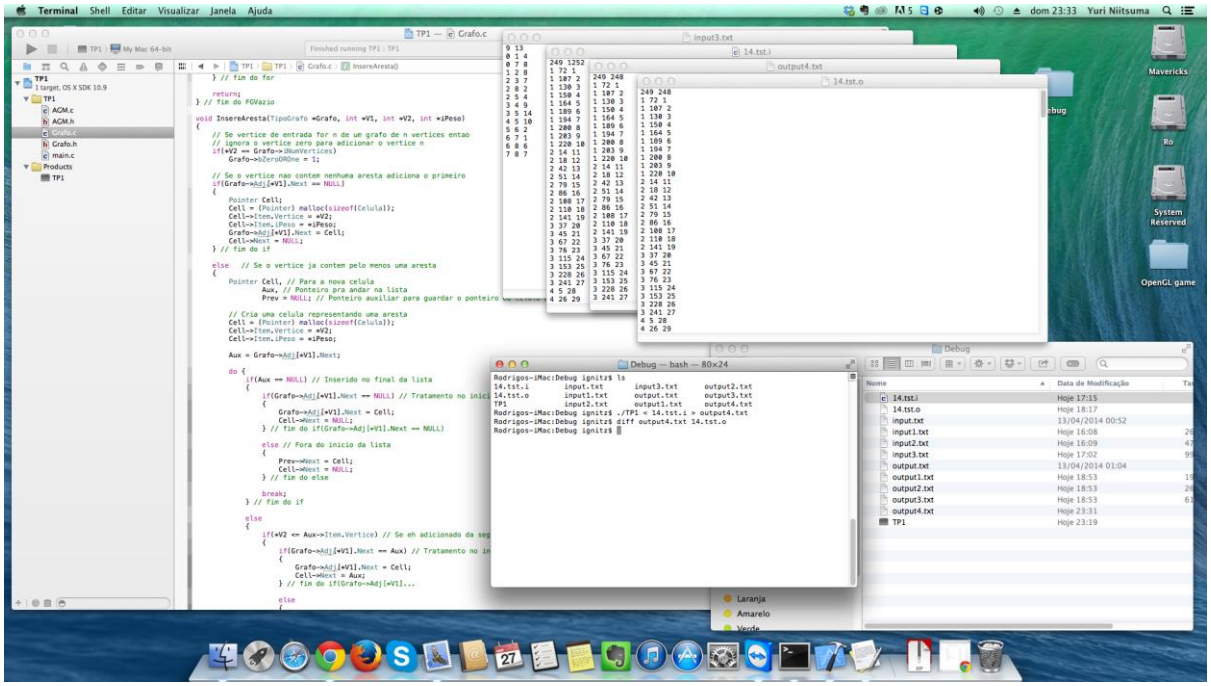
Função ArvoreMin: a função possui atribuições constantes até o primeiro for com complexidade $n+1$ atribuições, logo após possui 3 loops acoplados. O primeiro while para contar as arestas mínimas com complexidade $O(n-1) = O(n)$, que é a quantidade mínima de arestas para a AGM. O segundo while serve para percorrer toda a lista de adjacência pelo vetor dos vértices $O(n)$. O terceiro em cada lista de vértice é varrido todas as arestas em busca do menor peso disponível e depende da quantidade de arestas disponíveis, melhor caso: $O(1)$, pior caso: $O(n-1) = O(n)$ pois cada vértice pode possuir $n-1$ arestas.

Melhor caso: $O(n) \cdot O(n) \cdot O(1) = O(n^2)$
Pior caso: $O(n) \cdot O(n) \cdot O(n) = O(n^3)$

Função ImprimeMIN: a função apenas utiliza a função Imprime de complexidade $O(n^2)$. Logo segue a mesma complexidade da função chamada.

Testes

Vários testes foram realizados para procurar possíveis bugs, alguns grafos simples de 2, 3, 4 e 5 vértices foram usados no processo de codificação da Árvore Geradora Mínima para averiguar a funcionalidade correta e corrigir erros de programação. Segue abaixo de como foi o processo de testes no sistema.



Com o 14.tst.i disponibilizado no Moodle foi possível verificar que alguns testes iriam utilizar o vértice 0 até $n-1$ e outros o vértice de 1 até n. Com alguns ajustes foi possível detectar, adaptar e imprimir conforme necessário.

No site do Prático até o momento o código falha nos testes 1 e 3 e tem sucesso nos restantes.

Conclusão

O processo para gerar o código teve 3 versões, a primeira utilizava um vetor de tamanho fixo copiado do livro do Nivio, como não sabia o tamanho máximo dos vértices resolvi escrever do zero para montar um modelo mais complexo pra intuito de aprendizado, o modelo escolhido foi a matriz utilizando listas na Figura 3.10 na página 93. Mas após verificar na documentação do TP que era necessário lista de adjacência para montar o grafo resolvi reescrever o código mas com o vetor de lista com malloc pro tamanho conforme na entrada do usuário.
A implementação obteve dificuldades apenas em tratar casos especiais nas inserções das extremidades das arestas nas listas dos vértices e no algoritmo de Prim.

O Trabalho foi produtivo na habilidade de manipulação de ponteiros e listas.

Anexos

Listagem dos programas

- *main.c*
- *Grafo.h*
- *Grafo.c*
- *AGM.h*
- *AGM.c*