UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
CAMPUS CASCAVEL
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TAD GRAFO EM LINGUAGEM C VIA LISTA DE ADJACÊNCIA
ESTRUTURA DE DADOS

GUSTAVO ANTONIO MARTINI
GUSTAVO MACEDO
VINICIUS GILNEK DRAGE

Cascavel-PR
2022

UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
CAMPUS CASCAVEL
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TAD GRAFO EM LINGUAGEM C VIA LISTA DE ADJACÊNCIA
ESTRUTURA DE DADOS

GUSTAVO ANTONIO MARTINI
GUSTAVO MACEDO
VINICIUS GILNEK DRAGE

Suas funções, assinaturas e metodologia de teste

Cascavel-PR
2022

Sumário

1 Introdução

Este Documento tem como objetivo mapear e explicar as funções implementadas, em C, para TAD Grafo na disciplina de estrutura de dados. Para facilitar, além de ter sido solicitado pelo docente, existe o arquivo Makefile para compilar e testar um caso e ainda guardando o resultado em um ".txt", ainda permanece disponível um arquivo que pode ser acessado com menu de iterações disponível para uso.

2 Metodologia

Código desenvolvido em sincronia com GITHUB, utilizando a IDE do Visual Studio (Microsoft), o programa acompanhará um teste e um executável iterativo para o uso das funções criados, como mencionado na introdução, via Makefile.

3 Desenvolvimento e resultados

O tipo de Grafo que será implementado no programa é não-direcionado e ponderado, o que gera algumas características, em determinadas funções, que serão discutidas ao longo do texto.

A princípio para implementação do TAD foram utilizadas listas simplesmente encadeadas para sua criação, além de existir um ponteiro para o último elemento da lista- útil para implantação da lógica utilizada no problema-, seus nodos contém um tipo "long long int"que servem como armazenadores de ponteiros para estrutura vértice, pois a representação da lista é feita após a definição do nodo o que causa "Warnings"em, praticamente, todas as funções. Os nodos são constituídos também por: tamanho, que é usado somente na lista de arestas, e ponteiro para o próximo.

```
1 typedef long long int Tdado;     // define o tipo de dado dentro do node
2
3 typedef struct Tnode
4 {
5     Tdado info;                  //dado armazenado no node
6     int distancia;
7     struct Tnode *next;          //ponteiro para o proximo node
8 }Tnode;
```

Código do nodo da lista e o tipo "Tdado".

```
typedef struct TlistSE          //define o node head da list
{
    Tnode *first;               // ponteiro para o primeiro elemento da
    lista
    unsigned lenght;            // numero de elementos armazenado
    Tnode *last;                // ponteiro para o ultimo elemento da lista
}TlistSE;
```

Código da definição da Strutura "TlistSE".

Existe uma estrutura Grafo, esta que contém a lista de vértices responsável pela localização das vértices, assim como seu comprimento está ligado a quantidade de vértices Presentes no Grafo, cada "Tdado"da lista é sempre um ponteiro para Estrutura Vértice.

```
typedef struct Vertice
{
    TlistSE arestas;                // lista de arestas para cada vertice
}Vertice;


typedef struct Grafo
{
    TlistSE vertices;               // lista de vertices do grafo
}Grafo;
```

Código das estruturas do Grafo e da Vértice.

Na estrutura Vértice permanece a lista de arestas, está que é responsável por dizer para qual ponto a aresta vai, assim como o seu tamanho.

Funções presentes na Lista encadeada:

Existem, ao todo, 14 funções presentes na lista encadeada, estas que por sua são implementadas na biblioteca "Lista_SE.c".

```
1  void intitlist(TlistSE *L);
2  void deletelist(TlistSE *L);
3  int insertLeft(Tdado x, TlistSE *L);
4  int insertRight(Tdado x, TlistSE *L);
5  bool emptylist(TlistSE L);
6  int leghtList(TlistSE L);
7  Tdado removeleft(TlistSE *L);
8  Tdado removeRight(TlistSE *L);
9  Tnode* searchlist( Tdado x, TlistSE *L);
10 int searchposiinlist( Tnode *x, TlistSE *L);
11 Tnode* searchlistbyposi( int posi , TlistSE *L);
12 int insertlist(Tdado x,int p,TlistSE *L);
13 Tdado removelist(int p,TlistSE *L);
14 int insertRightifDistance(Tdado x,int distancia , TlistSE *L);
```

Todas as funções presentes na "ListaSE.h"e suas respectivas assinaturas.

Initlist: é responsável por iniciar a lista, deixando os seus ponteiros nulos, assim como seu comprimento igual a zero.

Deletelist: remove nodo a nodo, desalocando-os, da lista até acabarem, após esse passo desaloca a lista.

InsertLeft: insere um nodo no início da lista. Retorna 0 se for bem sucedida a inserção e 1 se ocorrer uma falha.

InsertRight: insere um nodo no final da lista. Retorna 0 se for bem sucedida a inserção e 1 se ocorrer uma falha.

Emptylist: retorna verdadeiro se o comprimento da lista for igual a zero.

LeghtList: retorna o tamanho da lista.

Removeleft: remove o primeiro nodo da lista, retorna o retirado. Retorna o nodo removido, se for nulo a lista está vazia.

Removeright: remove o último nodo da lista, retorna o retirado. Retorna o nodo removido, se for nulo a lista está vazia.

Searchlist: procura um Tdado solicitado em uma lista especificada pelo programador na função, retorna o ponteiro para o primeiro nodo em que Tdado for encontrado na lista, se não existir é retornado um ponteiro NULL.

Searchposiinlist: procura um ponteiro nodo igual ao que é enviado como parâmetro, se não existir nenhum será retornado o tamanho da lista + 1, o resultado é a posição do nodo enviado, a contagem começa em zero.

Searchlistbyposi: procura um nodo na lisa relativo a sua posição, retorna o ponteiro do nodo quando chega na posição solicitada, se for maior que a lista ou negativo retorna NULL.

Insertlist: insere na lista na posição desejada, se for igual a zero utiliza a função **insertLeft** e se for igual ao tamanho da lista utiliza **insertRight**, se a posição for maior que o tamanho da lista ou menor que 0 retorna 1 e não modifica a lista, além disso se não for possível alocar retorna 2 e uma mensagem.

Removelist: remove da lista na posição solicitada, invoca **removeleft** se for o primeiro nodo e se for o último chama **removeright**, se estiver no meio avança até o nodo n-1 guarda seu endereço e guarda o endereço do próximo, relativo ao nodo que será removido, e retira na posição solicitada, após a remoção reorganiza os ponteiros da lista. Retorna o ponteiro do nodo removido ou, em caso de falha retorna NULL.

InsertRightifDistance: função criada e utilizada apenas para

lista aresta, insere um nodo no final da lista e ainda solicita o tamanho da aresta esse que será guardado no nodo. Retorna 0 para o sucesso da inserção e 1 caso ocorra uma falha de alocação.

Todas esse funções mencionadas estão utilizadas para o funcionamento da biblioteca feita para tratar a TAD Grafo.

Funções Feitas para tratar a TAD Grafo:

Permanecem, ao todo, 9 funções presentes na lista encadeada, estas que por sua vez são usadas durante implementação do biblioteca Grafo.

```
1  void initGrafo(Grafo *G);
2  bool GrafoVazio(Grafo G);
3  int addVertice(Grafo *G);
4  int addAresta(Grafo *G,int vertice_1,int vertice_2,int tamanho_aresta);
5  void printGrafo(Grafo grafo);
6  Tnode* existearesta(Grafo *G, int Vertice_1, int Vertice_2);
7  int removeAresta(Grafo *G,int vertice_1, int vertice_2);
8  void printexistearesta(Grafo *G, int vertice_1, int vertice_2);
9  int removeVertice(Grafo *G, int vertice);
10 void printVerticeN(Grafo grafo, int vertice);
11 void DestroiGrafo(Grafo *G);
12 int menorAresta(Grafo G);
```

Todas as funções presentes no "Grafo.h"e suas respectivas assinaturas.

InitGrafo: Função que inicia o Grafo, inicia o tipo lista, com a função **initlist**, que guarda o endereço para as estruturas vértices.

Grafovazio: Verifica se a lista de vértices do grafo está vazio, se sim retorna TRUE.

AddVertice: Função responsável por adicionar vértices ao grafo, cria uma nova estrutura vértice e sua lista é inicializada com a função **initlist**. O único parâmetro da função é o ponteiro da estrutura grafo, pois o número do vértice é relativo a sua posição na lista. Caso falhe retorna 1, em sucesso retorna 0.

AddAresta: adiciona uma aresta entre vértices 'a' e 'b', no começo o endereço dos dois vértices é buscado, usando a função **searchlistbyposi**, lembrando que seu índice é relacionado com sua posição na lista, uma vez encontrados ambos é solicitada a inserção na lista da estrutura aresta de cada uma das vértices, o

conteúdo do nodo adicionado na vértice ’a’ é o ponteiro para o vértice ’b’, e vice-versa, e o tamanho da aresta utilizando a função **insertRightifDistance**. Em falhas nenhuma aresta é adicionada e é retornado 1, em sucesso o retorno é 0. Aresta não pode ser adicionada para o mesmo vértice, assim como não podem existir duas arestas entre as mesmas vértices, verificada com o método **Existearesta**.

PrintGrafo: função com um nome auto sugestivo, imprime o Grafo no terminal, vértice, numerado de 0 até o tamanho da lista de vértices - 1, impresso em conjunto com suas arestas, que, além de seu índice, demonstram para qual vértice ela vai.

Existearesta: verifica se existe aresta entre 2 vértices, encontra a primeira vértice relativa a posição na lista de vértices, com a função **searchlistbyposi**, então guarda seu endereço, após isso procura a segunda vértice e entra na lista de arestas da primeira, avança enquanto o "Tdado", que guarda o endereço da estrutura de vértice, for diferente do ponteiro da segunda, se não encontrar aresta retorna nulo e, se achar, retorna o ponteiro para o nodo da lista aresta da primeira vértice.

RemoveAresta: executa a remoção de aresta, parâmetros são 2 vértices, se existe aresta entre eles ela é removida, e o grafo deles, a função utiliza **existearesta** para encontrar o endereço dos nodos das arestas que serão removidas, encontra a estrutura vértice no grafo e entre na lista de aresta procurando pelo nodo igual ao retornado anteriormente, com a função **searposiinlist**, faça isso para as duas vértices, se tudo ocorrer bem retorna zero, senão retorna 1.

Printexistearesta: basicamente utiliza **existearesta** para poder imprimir o texto se existe ou não aresta.

RemoveVertice: remove vértice do grafo, procura a vértice a ser removida, via **searchlistbyposi**, vai em sua lista de aresta e vai buscando uma a uma, até seu destino, para chamar o método **removeAresta** enquanto existem arestas, após remover todas as arestas apaga a lista das mesmas- **deletelist**- , a vértice é desalocada e ,após isso, é removida da lista de vértices **removelist**, ao final remove o vértice da lista.

PrintVerticeN: procura a vértice, com **searchlistbyposi**, e imprime sua lista de arestas.

DestroiGrafo: utiliza **removeVertice** enquanto houverem vértices no gráfico.

MenorAresta: é uma função idêntica à **printGrafo**, porém ao invés de imprimir todos guarda a posição da menor aresta encontrada, se encontrar com tamanho igual ela é sobrescrita. Retorna o tamanho da aresta, se for zero quer dizer que o grafo está vazio.

Após explicado o funcionamento podemos partir para a compilação, uma vez que tenha baixado o repositório, basta executar "make", obrigatoriamente terminal Linux, que o teste será realizado e um programa de testes via terminal, chamado "MenuIteracao.x", será compilado. Além disso pode ser executado "make clean"para limpar os arquivos compilados.

4 Testes

O teste é, basicamente, um compilado de uso das funções do grafo, é feito junto ao makefile e é gravado em um arquivo chamado "teste.txt". Os teste são feitos com a inicialização do grafo, adicionamento de vértices, o grafo é impresso, é verificada a existência de vértice, adicionada a aresta e verificada a existência novamente. Procura da menor aresta e após todas as funções aplicadas o grafo é deletado e verificado como vazio.