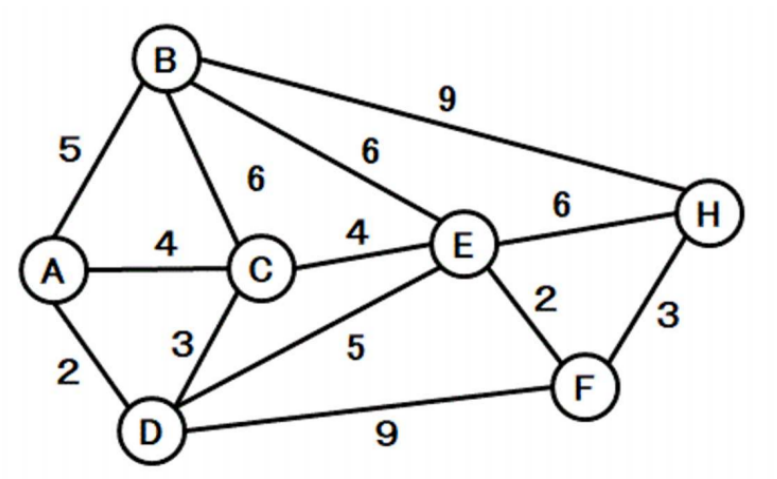
Estruturas de Dados Avançadas (EDA) – INF1010
Departamento de Informática – PUC-Rio - 2023.1
Tarefa 5 – Algoritmos sobre Grafos

Lívia Lutz dos Santos - 2211055

Luiza Marcondes Paes Leme - 2210275

Introdução:

O objetivo da tarefa é implementar o grafo abaixo por meio de listas de adjacências.



Além disso, fazer uma busca por profundidade a partir do vértice A, assim como aplicar o algoritmo de Dijkstra para descobrir o caminho mais curto a partir do mesmo vértice.

Estrutura do programa:

Usamos apenas 1 header (grafo.h) para definir as funções e structs usadas pela main.

```c
/*Lívia Lutz dos Santos - 2211055
Luiza Marcondes Paes Leme - 2210275
*/

typedef struct grafo Grafo;
typedef struct viz Viz;

Viz* criaViz(Viz* head, char noj, int peso);

Grafo* grafoCria(int nv, int na);

void imprimeGrafo(Grafo* g);

void dfs(Grafo* g, char v);

void dijkstra(Grafo* g, char v);
```

No módulo grafo.c importamos o header (grafo.h) e as bibliotecas <stdio.h> e <stdlib.h> para lidar com printf e alocações dinâmicas.

Definimos um array de char global contendo os caracteres de cada vértice do grafo para auxiliar nos algoritmos implementados e um array de inteiros para marcar quais vértices foram visitados (1) ou não (0).

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "grafo.h"

/*Array com cada letra no grafo*/
char caracteres[7] = { 'A', 'B', 'C', 'D', 'E', 'F', 'H' };

/*Array de vertices ja visitados nas buscas*/

int visitados[7] = { 0,0,0,0,0,0,0 };
```

Na struct viz, temos o campo noj, que representa cada letra dentro de cada vértice(char), o campo peso(int), que contém o peso de cada aresta que sai de cada vértice e um ponteiro para o próximo vértice (Viz*).

Na struct grafo, temos o campo nv, que representa o número de vértices do grafo, o campo na que representa a quantidade de arestas no grafo e um vetor de ponteiros para Viz, representando a lista encadeada (lista de adjacências) que parte de cada vértice do grafo, que irá conter os vértices adjacentes à ele.

```c
/*Struct que reprensenta a lista de adjacências*/

struct viz {
    char noj; /* representa cada letra dentro do nó */
    int peso; /*Representa o peso de cada aresta*/
    Viz* prox; /*Ponteiro para o proximo no*/
};

struct grafo {
    int nv; /* numero de nos ou vertices */
    int na; /* numero de arestas */
    Viz** viz; /* viz[i] aponta para a lista de arestas incidindo em i */
};
```

Na função auxiliar retornaIndice, percorremos o array de char caracteres até acharmos o elemento que corresponde ao parâmetro, assim retornando o índice correspondente.

```c
/*Função auxiliar que retorna o índice de um caracter no array de caracteres*/
int retornaIndice(char v) {
    int i;
    for (i = 0; i < 7; i++) {
        if (v == caracteres[i]) {
            return i;
        }
    }
}
```

Na função criaViz, passamos como parâmetro um ponteiro para a cabeça da lista encadeada contendo os vértices do grafo, o caracter correspondente ao nó que será inderido, assim como o peso de sua aresta. Começamos alocando dinamicamente um ponteiro do tipo Viz e verificamos se ocorreu de forma esperada, caso contrário, exibimos uma mensagem de erro e a função retorna NULL. Assim, colocamos os parâmetros nos devidos campos da estrutura. OBS: Como o próximo será o cabeça da lista, inserimos esse novo vizinho no início da lista.

```c
/*Cria lista encadeada de vizinhos*/
Viz* criaViz(Viz* head, char noj, int peso) {
    /* insere vizinho no inicio da lista */
    Viz* no = (Viz*)malloc(sizeof(Viz));

    if (no == NULL) {
        printf("Erro na alocacao de memoria para o no\n");
        return NULL;
    }

    no->noj = noj;
    no->peso = peso;
    no->prox = head;

    return no;
}
```

Na função grafoCria, passamos como parâmetros o número de arestas e de vértices do grafo (2 inteiros). Começamos alocando dinamicamente um ponteiro do tipo Grafo e testamos se a alocação ocorreu de forma esperada, caso contrário mostramos uma mensagem de erro é a função retorna NULL. Guardamos os parâmetros nos seus devidos campos e alocamos dinamicamente a lista de adjacências do grafo e no caso de erro fazemos o mesmo procedimento descrito acima só que para a cabeça da lista encadeada. Então, percorremos cada índice da lista definindo seu valor como NULL para que possamos adicionar os nós posteriormente, assim a função retina o ponteiro de Grafo montado.

```c
/*Cria um grafo */
Grafo* grafoCria(int nv, int na) {
    int i;
    Grafo* g = (Grafo*)malloc(sizeof(Grafo));

    if (g == NULL) {
        printf("Erro na alocacao de memoria para o grafo\n");
        return NULL;
    }

    g->nv = nv;
    g->na = na;
    g->viz = (Viz**)malloc(sizeof(Viz*) * nv);

    if (g->viz == NULL) {
        printf("Erro na alocacao de memoria para o vizinho\n");
        return NULL;
    }

    /*Inicializa a lista com NULL*/
    for (i = 0; i < nv; i++) {
        g->viz[i] = NULL;
    }

    return g;
}
```

Na função imprimeGrafo, passamos o grafo a ser impresso na tela como parâmetro e começamos verificando se seu ponteiro é NULL,caso seja, executamos o mesmo procedimento de erro descrito para a função acima. Então, para cada vértice, guardamos a cabeça da sua lista num ponteiro do tipo Viz auxiliar e até chegarmos ao fim da lista (ponteiro NULL) exibimos o caractere de cada vértice e o peso de sua aresta, então passamos para o próximo elemento da lista, repetindo o processo.

```c
/*Imprime cada vértice do grafo e sua lista de adjacências com os pesos e conteúdos de cada vizinho */
void imprimeGrafo(Grafo* g) {

    if (g == NULL) {
        printf("Grafo nao existe\n");
        return;
    }

    for (int i = 0; i < g->nv; i++) {
        printf("Vizinhos do no %c ", caracteres[i]);
        Viz* viz = g->viz[i];
        while (viz != NULL) {
            printf("-> %c(Peso %d) ", viz->noj, viz->peso);
            viz = viz->prox;
        }
        printf("\n");
    }

}
```

Na função dfs definimos 2 variáveis inteiras para guardar o índice e para servir de contador. Chamamos a função retornaIndice para obter o índice do char do parâmetro e guardamos o cabeça da lista numa variável ponteiro do tipo Viz, marcando como visitado (1) o índice do char no array visitados e imprimindo o caractere. Então fazemos um loop com esse ponteiro enquanto ele não é nulo e guardamos o índice do seu char e vemos se ele já foi visitado, se ainda não foi, continuamos a dfs a partir dele, senão, passamos para o próximo nó.

```c
void dfs(Grafo* g, char v) {

    /*Variavel de contagem e array de caracteres visitados*/

    int i = 0,j = 0;

    i = retornaIndice(v);

    /*Ponteiros para a lista de adjacencias*/

    Viz* no = g->viz[i];

    /*Marcando v como visitado no array e imprimindo sua letra*/

    visitados[i] = 1;
    printf("%c", caracteres[i]);

    /*Percorrendo a lista encadeada*/

    while (no != NULL) {

        /*Pega o indice do caracter contido no vertice no*/
        j = retornaIndice(no->noj);

        /*Se ele ainda nao foi visitado, continuamos a busca a partir dele*/
        if (visitados[j] == 0) {
            dfs(g, no->noj);
        }

        /*Passamos para o proximo no da lista*/
        no = no->prox;

    }

}
```

Na função auxiliar indMenorDistancia, definimos a distância infinita de um vértice até o outro com a macro INT_MAX e o índice com a menor distância como -1. Então, percorremos os arrays de distância e visitados passados como parâmetros e verificamos se aquele vértice já foi visitado e sua distância é menor que infinito, se sim, agora sua distância e seu índice são os menores e retornamos o índice.

```c
/*Função auxiliar que retorna o índice da lista de adjacências que tem a menor distância do nó corrente*/
int indMenorDistancia(int distancias[7], int visitado[7]){
    /*OBS: Consideramos a macro INT_MAX como infinito, pois é o maior inteiro representável*/
    int min = INT_MAX, minInd = -1;

    /*Percorremos os arrays dos parâmetros*/
    for (int i = 0; i < 7; i++) {
        /*Se o elemento no índice ainda não foi visitado e a distância é menor que infinito*/
        if (visitado[i] == 0 && distancias[i] <= min) {
            /*Sua distância é agora a mínima, assim como seu índice*/
            min = distancias[i];
            minInd = i;
        }
    }
    /*Retornamos o índice com a menor distância encontrada*/
    return minInd;
}
```

Na função dijkstra, definimos arrays de inteiros para distância e visitado e uma variável de contagem. Então, para cada elemento no array de distância, definimos como infinito, guardamos o índice do char passado como parâmetro e definimos a distância da origem (índice calculado) como 0. Continuamos com um loop para cada vértice, definimos a origem como visitado e para cada índice da lista de adjacências, pegamos o índice de cada caractere e se sua distância for maior q a da origem + seu peso e ele ainda não foi visitado, agora sua distância é igual a da origem. Após isso, imprimimos os caracteres e as distâncias em um passo a passo de cada vértice do caminho e guardamos o índice de menor distância para que possamos imprimir o caminho mais curto a partir dele.

```c
void dijkstra(Grafo* g, char v) {

    /*Array de distancias da origem aos outros vertices e de visitados*/
    int distancias[7], j = 0, visitado[7] = { 0,0,0,0,0,0,0 };

    /*Inicializando o array com distancias infinitas*/
    for (int i = 0; i < 7; i++) {
        distancias[i] = INT_MAX;
    }
    /*Pega o indice de v (origem)*/
    int i = retornaIndice(v);

    /*Distancia da origem para a origem e 0*/
    distancias[i] = 0;

    for (int cont = 0; cont < 7; cont++) {
        /*Marcando vertice como visitado*/
        visitado[i] = 1;

        /*Adiciona a distância se for menor do que a que já tiver*/
        for (Viz* a = g->viz[i]; a != NULL; a = a->prox) {
            j = retornaIndice(a->noj);
            if (distancias[j] > distancias[i] + a->peso && visitado[j] == 0) {
                distancias[j] = distancias[i] + a->peso;
            }
        }

        /*Imprime as distancias*/
        for (int i = 0; i < 7; i++) {
            printf("%c: %d\n", caracteres[i], distancias[i]);
        }
        printf("\n");

        /*Pega o indice do vertice com menor distancia*/
        i = indMenorDistancia(distancias, visitado);
    }

    printf("Caminho mais curto a partir da origem %c:\n", v);
    /*Imprime as distancias*/
    for (i = 0; i < 7; i++) {
        printf("%c: %d\n", caracteres[i], distancias[i]);
    }

}
```

Solução:

Para a main, começamos criando o grafo utilizando a função grafoCria e depois definimos um array com as letras do grafo e uma variável para a origem a ser considerada nos algoritmos. Para a representação do grafo em lista de adjacências, fizemos as inserções individualmente em ordem decrescente para ficar em ordem crescente de letras na estrutura e na impressão.

```c
int main(void) {

    /*Cria um grafo de 7 vértices e 13 arestas */
    Grafo* g = grafoCria(7, 13);

   /*Array com as letras que estarão contidas em cada vértice e char do caracter do vertice de origem */
    char letras[7] = { 'A', 'B', 'C', 'D', 'E', 'F', 'H' },origem = 'A';

    /*OBS : Decidimos colocar A como nó 0 pois ele será a origem
    para fazer o algoritmo de Dijkstra e o dfs*/

    /* Criando vizinhos de A*/
    g->viz[0] = criaViz(g->viz[0], letras[3], 2.0);
    g->viz[0] = criaViz(g->viz[0], letras[2], 4.0);
    g->viz[0] = criaViz(g->viz[0], letras[1], 5.0);

    /* Criando vizinhos de B*/
    g->viz[1] = criaViz(g->viz[1], letras[6], 9.0);
    g->viz[1] = criaViz(g->viz[1], letras[4], 6.0);
    g->viz[1] = criaViz(g->viz[1], letras[2], 6.0);
    g->viz[1] = criaViz(g->viz[1], letras[0], 5.0);

    /* Criando vizinhos de C */
    g->viz[2] = criaViz(g->viz[2], letras[4], 4.0);
    g->viz[2] = criaViz(g->viz[2], letras[3], 3.0);
    g->viz[2] = criaViz(g->viz[2], letras[1], 6.0);
    g->viz[2] = criaViz(g->viz[2], letras[0], 4.0);
```

```c
/* Criando vizinhos de C */
g->viz[2] = criaViz(g->viz[2], letras[4], 4.0);
g->viz[2] = criaViz(g->viz[2], letras[3], 3.0);
g->viz[2] = criaViz(g->viz[2], letras[1], 6.0);
g->viz[2] = criaViz(g->viz[2], letras[0], 4.0);

/* Criando vizinhos de D */
g->viz[3] = criaViz(g->viz[3], letras[5], 9.0);
g->viz[3] = criaViz(g->viz[3], letras[4], 5.0);
g->viz[3] = criaViz(g->viz[3], letras[2], 3.0);
g->viz[3] = criaViz(g->viz[3], letras[0], 2.0);

/* Criando vizinhos de E */
g->viz[4] = criaViz(g->viz[4], letras[6], 6.0);
g->viz[4] = criaViz(g->viz[4], letras[5], 2.0);
g->viz[4] = criaViz(g->viz[4], letras[3], 5.0);
g->viz[4] = criaViz(g->viz[4], letras[2], 4.0);
g->viz[4] = criaViz(g->viz[4], letras[1], 6.0);

/* Criando vizinhos de F */
g->viz[5] = criaViz(g->viz[5], letras[6], 3.0);
g->viz[5] = criaViz(g->viz[5], letras[4], 2.0);
g->viz[5] = criaViz(g->viz[5], letras[3], 9.0);

/* Criando vizinhos de H */
g->viz[6] = criaViz(g->viz[6], letras[5], 3.0);
g->viz[6] = criaViz(g->viz[6], letras[4], 6.0);
g->viz[6] = criaViz(g->viz[6], letras[1], 9.0);
```

Por fim, a main chama a função imprimeGrafo, a de dijkstra e a de dfs, colocando prints informativos sobre o que irá ser impresso, uma vez que as funções chamadas possuem prints dentro delas.

```c
printf("Grafo como lista de adjacencias:\n");
/* Imprime o grafo em ordem alfabética */
imprimeGrafo(g);

printf("\nImplementacao do algoritmo de Dijkstra a partir do vertice %c:\n", origem);

/*Imprime passo a passo do algoritmo e resultado*/
dijkstra(g, origem);

/*Faz a busca por profundidade no grafo usando percurso por profundidade*/
printf("\nBusca em profundidade a partir do vertice %c:\n",origem);

dfs(g, origem);

return 0;
```

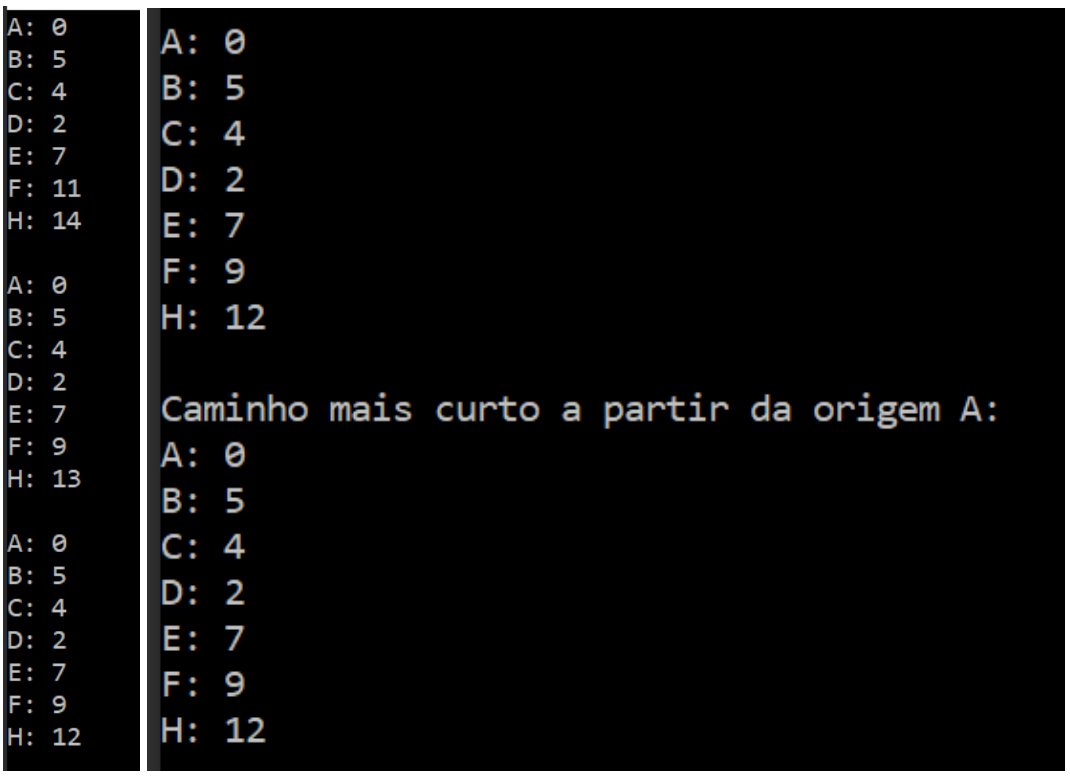
Essa parte final da main é responsável pelas impressões mostradas a seguir.

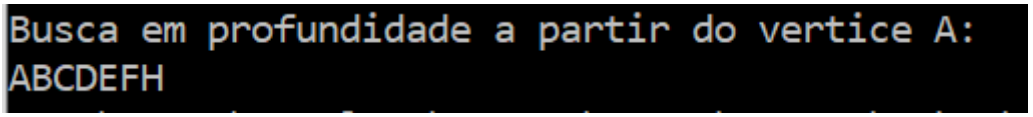
```
Grafo como lista de adjacencias:
Vizinhos do no A -> B(Peso 5) -> C(Peso 4) -> D(Peso 2)
Vizinhos do no B -> A(Peso 5) -> C(Peso 6) -> E(Peso 6) -> H(Peso 9)
Vizinhos do no C -> A(Peso 4) -> B(Peso 6) -> D(Peso 3) -> E(Peso 4)
Vizinhos do no D -> A(Peso 2) -> C(Peso 3) -> E(Peso 5) -> F(Peso 9)
Vizinhos do no E -> B(Peso 6) -> C(Peso 4) -> D(Peso 5) -> F(Peso 2) -> H(Peso 6)
Vizinhos do no F -> D(Peso 9) -> E(Peso 2) -> H(Peso 3)
Vizinhos do no H -> B(Peso 9) -> E(Peso 6) -> F(Peso 3)
```

```
Implementacao do algoritmo de Dijkstra a partir do vertice A:
A: 0
B: 5
C: 4
D: 2
E: 2147483647
F: 2147483647
H: 2147483647

A: 0
B: 5
C: 4
D: 2
E: 7
F: 11
H: 2147483647

A: 0
B: 5
C: 4
D: 2
E: 7
F: 11
H: 2147483647
```





Conclusão:

O programa funcionou como esperado e os resultados dos algoritmos foram verificados com cálculos à mão. Implementar a função de inserir na lista de adjacências foi trivial, uma vez que implementamos com uma lista encadeada, estrutura que já conhecíamos.

Tivemos alguns problemas de erros externos que resolvemos definindo as structs na main e enquanto desenvolvíamos a função dfs, que é recursiva, tivemos que declarar o array de visitados como global pois a cada chamada da função recursiva, o array voltava a seus valores originais, o que resultava em um loop infinito, logo isso levou um pouco mais de tempo por conta desse raciocínio, mas não tivemos nenhuma dificuldade além dessas.