TAREFA 5 – EAD

SINGULAR:

Nome: Enzo Tres Mediano

Matricula: 2211731

//ENUNCIADO:

Estruturas de Dados Avançadas (EDA) – INF1010 Departamento de Informática – PUC-Rio 2023.1 Tarefa 5 – Algoritmos sobre Grafos Dado o grafo: 1) Represente o grafo acima usando lista de adjacências. 2) Implemente na Linguagem C os seguintes algoritmos, criando o TDA de grafos:

a) Calcule o caminho mais curto para os nós, a partir do vértice A, utilizando o algoritmo de Dijkstra.

b) Indicar um caminho de uma busca em profundidade, a partir do vértice A. Orientações de entrega: Faça o upload no site do EAD dos seguintes:

- TDA de grafos contendo os arquivos .c e .h correspondentes
- Relatório em .pdf contendo:

• Identificação. Nome e matricula do(s) aluno(s), sendo no máximo 2 alunos por trabalho;

• Introdução. Breve descrição sobre o conteúdo do trabalho;

• Estrutura do programa. Descrição da interface e as funções implementadas;

• Solução. Descrição da solução de cada algoritmo, incluindo capturas do código fonte e a saída do programa;

• Observações e conclusões. Descrição de aspectos que precisem ser destacados, tais como dificuldades e facilidades encontradas. Data de Entrega: Até 23:59 do dia 28/junho/2023

//PseudoCodigo dos Funcoes e Solucao

```
Graph* createGraph() {
    Malloc do Grafo

    Malloc dos Nodes dentro do Grafo

    Coloca todos os Nodes como NULL

    Insere Nodes individualmente usando função de criar Node

    graph->n = numero de Nodes;

    retorno grafo
}
```

Cria a totalidade do gráfico, mas não é o que está criando os nós individuais. Espaço Mallocs para o endereço dos Nodos. Usa uma quantidade menor de codificação para permitir que o sistema saiba quais nós precisam de quais informações.

```
void createGraphNode(GraphNode** n, char c, int* weights, char* nodes, int
numNodes) {  //Creates graph nodes that later point to chainedlist of adj

    malloc do Node especifico

    Inserir valores dentro do Node

    Node* currentNode = (*n)->listAdj;

    Loop que vai inserindo todos os Nodes do listAdj
}
```

Cria os nós específicos do gráfico e insere todos os nós adj como uma lista encadeada com esse nó. A explicação da estrutura será mostrada mais tarde com uma imagem desenhada de como ela deve aparecer.

```
Node* createNode(char c, int weight) {  //creates Node (self explanatory)
    Malloc para Node especifico

    Inserir valores dentro do Node criado

    return Node criado
}
```

A função cria os nós adj para a lista encadeada que segue o nó gráfico. Função simples, então não há muitos detalhes para incluir sobre esta função.

```
void freeGraph(Graph* graph) { //Frees graph (self explanatory)

    Inicializar valores para usar depois

    Loop para fazer free de cada valor dentro do Graphnodes

        Free de cada Node nas listas de adj
        Free do Node GraphNode

    }
    Free do endereço que representa o grafo inteiro
}
```

Função que libera a tabela do gráfico em sua totalidade quanto a limpeza para os mallocs feitos. Função simples, então não há muitos detalhes para incluir sobre esta função.

```
void printGraph(Graph* graph) {

    Inicializar valores para usar depois

    Loop para fazer print de cada valor dentro do Graphnodes

        Free de cada Node nas listas de adj
        Free do Node GraphNode
}
```

Função para imprimir o gráfico de forma a garantir que tudo foi criado corretamente (Também utilizado para responder à questão 1 do enunciado). Função simples, então não há muitos detalhes para incluir sobre esta função

```
int minDisIndex(int dist[], bool visited[]) {

    Inicializar valores para usar depois

    Loop que vai achando caminho que tem menor peso e também não foi
visitado antes

    Retorna indexo que tinha o menor caminho
}
```

Obtém duas listas com uma tendo pesos e a outra com a informação se um nó específico foi visitado ou não. O bool visited[] funciona porque usa o mesmo método de hash do array GraphNodes. Em seguida, ele examina cada caminho e descobre qual não foi visitado antes e tem o menor peso. É muito utilizado na função Dijkstra.

```
void printPath(int parent[], int curr, Graph* graph) {
    Ver se o Node tem pai ou nao
    Chama o proprio printPath para fazer um recursao
    Print do caractere
}
```

Usando o mesmo método de hash de antes, ele também tem o pai do nó sendo usado para que a função possa trabalhar de trás para frente no gráfico para imprimir cada caractere do caminho da origem até o último nó no caminho.

```
void dijkstra(Graph* graph, char source) {

    Inicializar valores para usar depois


    Loop que termina quando o count chega no mesmo valor que o numero de
vertices que tem no grafo.

        Pega indice do caminho menor.
        Atualiza o Node para que da para saber depois que foi encontrado

        Inicializar valores para usar depois

        Loop para caminhar o lista de adj
            Ver se Node ta visitado ja e se qual caminho ta menor que o
outro no situacao.
            Inserir valores no Node índice e no Node pai


    }

    Usa um funcao para impremir os caminhos criados nessa funcao de
Dijkstra.
}
```

Usando o mesmo método de hash de antes, ele também tem o pai do nó sendo usado para que a função possa trabalhar de trás para frente no gráfico para imprimir cada caractere do caminho da origem até o último nó no caminho.

```
void dFS(Graph* graph, int checked[], int vertex) {
    Coloca que o Node ta visitado
    Node* currNode = graph->GraphNodes[vertex];
    Imprimir Node visitado

    Loop ate NULL
        Situacao aonde vai ter o recursao do funcao
        Ir pro próximo Node usando o ->next
    }

}
```

Usa uma função de recursão, pois precisa ver todos os nós. Embora haja a atualização para os nós que estão sendo recursados, ele não usará a função recursiva várias vezes em um único nó. O recursivo foi o melhor, pois funcionou de maneira semelhante a uma pilha, que é a estrutura de dados usada para busca em profundidade.

Em resumo, o programa primeiro coleta informações sobre quais nós se conectam e alocam espaço para representar o grafo usando uma lista de adjacências. Depois o programa imprimiria o gráfico como para responder a primeira pergunta do enunciado. Dentro da função Dijkstra, podemos manobrar entre nó e nó usando um sistema de hash para o gráfico, sendo capaz de identificar o hash de um caractere com um simples uso de menos 'A' para que possamos transformar um caractere em seu int para o hash .Então entramos no loop for com a contagem que se repete até que a contagem atinja o número de vértices no grafo. Cada vez que ele faz o loop, ele identifica o caminho menos custoso e atualiza as variáveis do peso atual do nó de onde veio e garante que ele vá para o próximo nó para que o loop possa se repetir com um novo nó. Após esta função, o programa faz uma impressão de todas as distâncias e os caminhos percorridos para chegar da origem ao último nó. Depois disso, a questão 2a está completa. Para a busca em profundidade foi utilizada uma função de recursão simples onde atualiza os nós a serem considerados como visitados cada vez que um novo nó entra na função recursiva. Isso é feito para que, posteriormente, dentro do loop, o programa saiba que não deve usar nós que já passaram pela função recursiva. Depois de passar por uma função que imprime o caminho que a busca de profundidade percorreu, a questão 2b está finalizada.

Output da Programa:

```
Here is how the graph turned out:
|GraphNode: A |-> |Node: D 2|-> |Node: C 4|-> |Node: B 5|
|GraphNode: B |-> |Node: H 9|-> |Node: E 6|-> |Node: C 6|-> |Node: A 5|
|GraphNode: C |-> |Node: E 4|-> |Node: D 3|-> |Node: B 6|-> |Node: A 4|
|GraphNode: D |-> |Node: F 9|-> |Node: E 5|-> |Node: C 3|-> |Node: A 2|
|GraphNode: E |-> |Node: H 6|-> |Node: F 2|-> |Node: D 5|-> |Node: C 4|-> |Node: B 6|
|GraphNode: F |-> |Node: H 3|-> |Node: E 2|-> |Node: D 9|
|GraphNode: H |-> |Node: F 3|-> |Node: E 6|-> |Node: B 9|
Path with Dijkstra:
Vertex          Distance        Path
A -> A          0               A
A -> B          5               A B
A -> C          4               A C
A -> D          2               A D
A -> E          7               A D E
A -> F          9               A D E F
A -> H          12              A D E F H
Path with DFS:
A D F H E C B
```

//Conclusao

Havia muito a aprender à medida que avançava cada vez mais na designação. A tarefa passou a ser de dificuldade moderada, onde não era simples de implementar, mas também não era extremamente difícil encontrar uma solução. A primeira vez que tentei completar a atribuição para a função Dijkstra foi com um minHeap, mas acabou sendo mais complicado do que eu esperava e, como era um gráfico tão pequeno para a atribuição, optei por usar uma estrutura contendo hashes e vinculada listas. No geral, uma boa atribuição que demonstra as expectativas do que está por vir no futuro ao trabalhar novamente nas estruturas de dados.