Pontifícia Universidade Católica do Rio de Janeiro
INF1010 - Estruturas de Dados Avançadas

Lucas Demarco Cambraia Lemos - 2110013
Jayme Augusto Avelino de Paiva - 2210289

Tarefa 5
Grafos

Rio de Janeiro
2023.1

1. INTRODUÇÃO

A Tarefa 5 propõe a implementação de algumas operações para grafos, que foram aprendidas em sala de aula, em um grafo dado no enunciado. Para isso, é necessário inicializar o grafo, utilizando lista de adjacências e, em seguida, elaborar funções, junto com suas auxiliares, para percorrer o grafo em uma busca em profundidade. Finalizando a tarefa, implementamos o Algoritmo de Dijkstra, a partir do vértice A, exibindo a distância de cada vértice até a origem.

2. EXPLICAÇÃO DA SOLUÇÃO E IMAGENS DO CÓDIGO-FONTE

As estruturas que utilizamos para trabalhar com o grafo foram as seguintes:

- Graph: estrutura que representa o grafo, com os campos de quantidade total de vértices que possui e um ponteiro para a lista de adjacências:

```
6   typedef struct adjList AdjList;
7   struct adjList{
8       int node;
9       int weight;
10      AdjList* prox;
11  };
```

- AdjaList: representa os vértices do grafo. Cada vértice possui o seu valor, o peso da aresta que o conecta com o vértice da posição do vetor, além de um ponteiro para o próximo vértice, representando a lista.

```
6   typedef struct adjList AdjList;
7   struct adjList{
8       int node;
9       int weight;
10      AdjList* prox;
11  };
```

- Pilha: estrutura auxiliar para a busca em profundidade, que possui um campo para representar o vértice e um ponteiro para o próximo.

```
17  typedef struct pilha Pilha;
18  struct pilha{
19      int elem;
20      Pilha* prox;
21  };
```

Para inicializar o grafo, a função *makeGraph* aloca memória para cada posição do vetor que representa cada vértice. Em seguida, utilizando um vetor de vetores já definido com o peso de cada aresta, criamos a lista de adjacências, com tamanho já definido pelo enunciado da questão

```
23  Graph* makeGraph(){
24      Graph* graph = (Graph*)malloc(sizeof(Graph));
25      if (!graph){
26          printf("ERROR: Falta de memória\n");
27          exit(1);
28      }
29      AdjList** adj = (AdjList**)malloc(7*sizeof(AdjList*));
30      if (!adj){
31          printf("ERROR: Falta de memória\n");
32          exit(1);
33      }
34      int arestaPeso[7][7] = {
35          {0,5,4,2,0,0,0},
36          {5,0,6,0,6,0,9},
37          {4,6,0,3,4,0,0},
38          {2,0,3,0,5,9,0},
39          {0,6,4,5,0,2,6},
40          {0,0,0,9,2,0,3},
41          {0,9,0,0,6,3,0}
42      };
```

```
43      for (int i = 0; i<7; i++){
44          *(adj+i) = (AdjList*)malloc(sizeof(AdjList));
45          if (!(*(adj+i))){
46              printf("ERROR: Falta de memória\n");
47              exit(1);
48          }
49          AdjList* atual = *(adj+i);
50          AdjList* ant = atual;
51          for (int j = 0; j<7; j++){
52              if (arestaPeso[i][j] != 0){
53                  atual->node = j;
54                  atual->weight = arestaPeso[i][j];
55                  ant = atual;
56                  atual->prox = (AdjList*)malloc(sizeof(AdjList));
57                  if (!(atual->prox)){
58                      printf("ERROR: Falta de memória\n");
59                      exit(1);
60                  }
61                  atual = atual->prox;
62              }
63          }
64          free(atual);
65          ant->prox = NULL;
66      }
67      graph->qtdNodes = 7;
68      graph->adj = adj;
69      return graph;
70  }
```

Para realizar a busca em profundidade do grafo fornecido, implementamos a função *dfs*, que empilha os vértices que serão visitados. Como a busca em profundidade percorre as arestas ligadas ao vértice por último, a pilha serve para organizar a ordem que cada vértice deve ser exibido. Além disso, foi usado um vetor auxiliar de inteiros, que coloca valor 1 para quando já visitamos e 0 para os vértices que faltam. Esse *array* tem a função de auxiliar e impedir de visitarmos o mesmo vértice duas vezes.

A função *dfs* percorre a lista de adjacências para cada posição do vetor, até que encontre *NULL*, que significa que percorreu todos os vértices ligados ao vértice representado por aquela posição do vetor. Segue abaixo a imagem do código da função..

```
105 void dfs(Graph* graph, Pilha* pilha, int* visitados){
106     if (!pilha){
107         printf("\n");
108         return;
109     }
110     printf("%c ", pilha->elem == 6 ? pilha->elem+1+65 : pilha-
    >elem+65);
111     Pilha* newPile = (Pilha*)malloc(sizeof(Pilha));
112     if (!newPile){
113         printf("ERROR: Falta de memória\n");
114         exit(1);
115     }
116     newPile->prox = pilha->prox;
117     AdjList* adj = *((graph->adj)+pilha->elem);
118     free(pilha);
119     while(adj){
120         if (!(*(visitados+adj->node))){
121             *(visitados+adj->node) = 1;
122             newPile->elem = adj->node;
123             Pilha* newNewPile = (Pilha*)malloc(sizeof(Pilha));
124             if (!newNewPile){
125                 printf("ERROR: Falta de memória\n");
126                 exit(1);
127             }
128             newNewPile->prox = newPile;
129             newPile = newNewPile;
130         }
131         adj = adj->prox;
132     }
133     pilha = newPile->prox;
134     free(newPile);
135     return dfs(graph,pilha,visitados);
136 }
```

O algoritmo de Djikstra foi feito inicializando uma lista de nós visitados toda zerada de tamanho igual ao números de nós do grafo. Outra lista de distância foi inicializada com o maior valor possível de um inteiro em C (para ser tratado como infinito) para cada nó do grafo. Após isso, a distância para o nó inicial é igualada a 0 e uma variável para guardar a quantidade de visitas é feita. Isso é utilizado para saber quando o algoritmo deve parar, que é no caso de a quantidade de visitas ser igual ao número de nós do grafo, o que significará que todos os nós foram visitados e, portanto, estão com a distância calculada otimizada.

As distâncias da lista são atualizadas num loop sempre verificando se a nova distância achada até os nós em adjacentes ao nó atual são menores que a distância na lista de distâncias, se alguma delas for, a distância na lista é atualizada. Após isso o nó que o algoritmo está visitando atualmente é trocado para o nó de menor distância não visitado até então e marcado como visitado. Com isso, o algoritmo continua até que todos sejam visitados, como dito anteriormente. Por fim, o algoritmo retorna uma lista com as distâncias otimizadas.

```c
int* djikstra(Graph* graph, int initialNode){
    int* dist = (int*)malloc(sizeof(int)*(graph->qtdNodes));
    if (!dist){
        printf("ERROR: Falta de memória\n");
        exit(1);
    }
    int visitados[graph->qtdNodes];
    for(int i=0; i<graph->qtdNodes;i++){
        *(dist+i) = INFINITY;
        visitados[i] = 0;
    }
    *(dist+initialNode) = 0;
    int qtdVisitas = 0;
    int actualNode = initialNode;
    while(qtdVisitas < graph->qtdNodes){
        qtdVisitas++;
        visitados[actualNode] = 1;
        AdjList* actual = *((graph->adj)+actualNode);
        while (actual){
            *(dist+actual->node) = (actual->weight + *(dist+actualNode) < *(dist+actual->node) ? actual->weight + *(dist+actualNode) : *(dist+actual->node));
            actual = actual->prox;
        }
        int menorDist = INFINITY;
        for (int i=0; i<graph->qtdNodes; i++){
            if (!visitados[i] && *(dist+i) <= menorDist){
                menorDist = *(dist+i);
                actualNode = i;
            }
        }
    }
    return dist;
}
```

3. CONCLUSÃO

A busca *dfs* não teve muito problema em ser implementada, visto que exigiu mais do nosso conhecimento acerca de listas, junto com a regra da busca para os grafos. Como explicado pelo professor Seibel, a ordem que percorre pode variar, de acordo com a maneira que o grafo foi inicializado, ou seja, como a lista está disposta.

Para o Algoritmo de Djikstra, mostrou-se um pouco mais trabalhoso, mas conseguimos implementar. O resultado foi totalmente de acordo com o grafo mostrado no enunciado, já que as distâncias estão condizentes com o grafo usado como base. A saída do programa pode ser visualizada abaixo: