Universidade de Trás-os-Montes e Alto Douro
Escola de Ciências e Tecnologia. Departamento de Engenharias

Lic. Engenharia Informática
Algoritmia e Estruturas de Dados (1º Ano - 2º Semestre). **2023 - 2024**

CÓDIGO BASE DE **GRAFOS**

CÓDIGO BASE DE GRAFO – NÃO-GENÉRICO (DE INTEIROS NESTE CASO)

```
// Header file for GRAPH

#ifndef _GRAPH_DEFINITIONS
#define _GRAPH_DEFINITIONS
…
//---------------------------------------------------------

#ifndef _LOGICAL
#define _LOGICAL

typedef enum { ERROR = 0, OK = 1 } STATUS;
typedef enum { FALSE = 0, TRUE = 1 } BOOLEAN;

#endif // !_LOGICAL

//---------------------------------------------------------

// graph edge definition
typedef struct _EDGE {
   int v, w;
   float weight;
}EDGE;

//---------------------------------------------------------

// graph definitions
typedef struct _GRAPH_M {
   int nVertices;
   int nEdges;
   int* pD;
   float** adjMatrix;
}GRAPH_M;

typedef struct _GRAPH_L {
   int nVertices;
   int nEdges;
   int* pD;
   LINKED_LIST** adjList;
}GRAPH_L;

//---------------------------------------------------------

#ifndef _VERTICE_DATA_MACRO
#define _VERTICE_DATA_MACRO

// macro definition for easy access to a vertice data field
#define DATA_V(X) ((X)->pD)

#endif // !_DATA_MACRO

//---------------------------------------------------------
// Functions Declarations -------------------------------
// (only for adjacency matrix representation) -----------
//---------------------------------------------------------

// Constructors and Destructors -------------------------

STATUS initGraph_M(GRAPH_M*, int);
STATUS destroyGraph_M(GRAPH_M*);

// Input and Output -------------------------------------

EDGE setEdge(int, int, float);

STATUS addVertexGraph_M(GRAPH_M*, int);
STATUS addEdgeGraph_M(GRAPH_M*, EDGE, BOOLEAN);  // the BOOLEAN argunment is to indicate whether
                                                 // the graph is directed or not
```

```
int removeVertexGraph_M(GRAPH_M*);
EDGE removeEdgeGraph_M(GRAPH_M*);

// Query ------------------------------------------------

BOOLEAN emptyGraph_M(GRAPH_M G);

STATUS printAdjMatrix(GRAPH_M);
STATUS printVertices(GRAPH_M);

STATUS DFSTraverseGraph_M(GRAPH_M);
STATUS DFSGraph_M(INT_GRAPH_M, EDGE, int*, int*, int*);

#endif // !_GRAPH_DEFINITIONS
```

```
// Source file for GRAPH_M

#include "Graph.h"

//-------------------------------------------------
// Constructor and Destructor Functions -----------

STATUS initGraph_M(GRAPH_M* pG, int size)
{
   // adjacency matrix allocation
   if ((pG->adjMatrix = (float**)malloc(size * sizeof(float*))) == NULL)
   {
      printf("\nGraph: adjacency matrix memory allocation error.");
      return ERROR;
   }

   for (int i = 0; i < size; i++)
   {
      if ((pG->adjMatrix[i] = (float*)malloc(size * sizeof(float))) == NULL)
      {
         printf("\nGraph: adjacency matrix memory allocation error.");
         for (int j = i; j >= 0; j--)
            free(pG->adjMatrix[j]);
         free(pG->adjMatrix);
         return ERROR;
      }
   }

   // vertices' data array allocation
   if ((pG->pD = (int*)malloc(size * sizeof(int))) == NULL)
   {
      printf("\nGraph: vertices data memory allocation error.");
      return ERROR;
   }

   // adjacency matrix update
   for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
         pG->adjMatrix[i][j] = 0.0;

   pG->nEdges = 0;
   pG->nVertices = 0;

   return OK;
}

//-------------------------------------------------

STATUS destroyGraph_M(GRAPH_M* pG)
{
   // underflow
   if (emptyGraph_M(*pG))
      return ERROR;

   // freeing allocated memory
   free(pG->pD);
   for (int i = 0; i < pG->nVertices; i++)
      free(pG->adjMatrix[i]);
   free(pG->adjMatrix);

   pG->nEdges = 0;
   pG->nVertices = 0;

   return OK;
}
```

```
//-----------------------------------------------
// Input Functions -------------------------------

EDGE setEdge(int v, int w, float weight)
{
   EDGE e;
   e.v = v; e.w = w; e.weight = weight;

   return e;
}

//-----------------------------------------------
// Query Functions -------------------------------

BOOLEAN emptyGraph_M(GRAPH_M G)
{
   return ((G.nVertices == 0) ? TRUE : FALSE);
}

//-----------------------------------------------
// Query Functions – Traversing & Printing --------

// recursive DepthFirstSearch version
STATUS DFSGraph_M(GRAPH_M G, EDGE e, int order[], int from[], int* pCnt)
// non-generic - suitable only for int data
// order[] is to keep the order vertices are visited
// from[] is to keep where from vertices are visited
{
   // treats starting vertex differently
   if(*pCnt == 0)
      printf("\nvertex %-3d: %d", 0, G.pD[0]);

   order[e.w] = (*pCnt)++;
   from[e.w] = e.v;
   // searches for the next non-empty connection
   for (int j = 0; j < G.nVertices; j++)
      if (G.adjMatrix[e.w][j] != 0)
         if (order[j] == -1)
         {
            printf("\nvertex %-3d: %d", j, G.pD[j]);
            // searches for the next non-empty connection of the previous destiny vertex
            DFSGraph_M(G, setEdge(e.w, j, 0), order, from, pCnt);
         }

   return OK;
}

//-----------------------------------------------
// Query Functions - Printing --------------------

STATUS printAdjMatrix(GRAPH_M G)
{
   // underflow
   if (emptyGraph_M(G))
      return ERROR;

   printf("\n\nAdjacency Matrix:\n\n");
   printf("      ");
   for (int i = 0; i < G.nVertices; i++)
      printf("%3d  ", i);
   printf("\n\n");
   for (int i = 0; i < G.nVertices; i++)
   {
      printf("%3d   ", i);
      for (int j = 0; j < G.nVertices; j++)
         printf(" %2.1f ", G.adjMatrix[i][j]);
      printf("\n");
   }

   return OK;
}

//-----------------------------------------------
```

```
STATUS printVertices(GRAPH_M G)
// non-generic - suitable only for int data
{
   // underflow
   if (emptyGraph_M(G))
      return ERROR;

   printf("\nVertices:\n");
   for (int i = 0; i < G.nVertices; i++)
      printf("\nvertex %-3d: %d”, i, G.pD[i]);

   return OK;
}

//-------------------------------------------------

STATUS DFSTraverseGraph_M(GRAPH_M G)
// non-generic - suitable only for int data
{
   // underflow
   if (emptyGraph_M(G))
      return ERROR;

   // starting vertex
   EDGE e;
   e.v = 0;
   e.w = 0;
   e.weight = 0;

   // order[] is to keep the order vertices are visited
   // from[] is to keep where from vertices are visited
   int order[NMAX_VERTICES];
   int from[NMAX_VERTICES];
   for (int i = 0; i < NMAX_VERTICES; i++)
   {
      order[i] = -1;
      from[i] = -1;
   }
   int cnt = 0;

   printf("\nvertices");
   DFSGraph_M(G, e, order, from, &cnt);  //recursive DFS function – prints each vertex in turn

   // prints order[] and from[]
   printf("\n\nvisiting order");
   for (int i = 0; i < G.nVertices; i++)
      printf("\nvertex %-3d: #%d", i, order[i]);
   printf("\n\nantecessors");
   for (int i = 0; i < G.nVertices; i++)
      printf("\nvertex %-3d: from vertex %d ", i, from[i]);

   return OK;
}
```