# Instituto Politécnico do Cávado e do Ave

# Escola Superior de Tecnologia

# Curso: Engenharia de Sistemas Informáticos

# Disciplina: Estruturas de Dados Avançadas

# Ano Letivo: 2023-2024

Relatório do Trabalho Prático nº 2

# Aluno: Rodrigo Ferreira Moura

# Número: 27995

# Data: 24/05/2024

# Resumo

Este documento descreve o trabalho realizado para o segundo trabalho prático da unidade curricular de Estruturas de Dados Avançadas. O objetivo foi desenvolver e implementar um programa que manipula grafos, incluindo a importação, manipulação de vértices e arestas, e a preservação e carregamento de grafos em arquivos.

# 1 - Introdução

## Enquadramento

Este documento descreve o trabalho realizado para o segundo trabalho prático da unidade curricular de Estruturas de Dados Avançadas. O trabalho focou-se na manipulação e preservação de grafos, abordando diferentes aspectos como a representação de grafos, operações de travessia e a implementação de métodos para salvar e carregar grafos de arquivos.

## Problema a resolver

O objetivo do trabalho é desenvolver um programa em C que permita a criação, manipulação, travessia e preservação de grafos. O enunciado do trabalho propõe a implementação de diversas funções que permitam a leitura de grafos de um arquivo, a manipulação de vértices e arestas, e a preservação do estado do grafo em um arquivo binário.

# 3 - Trabalho Desenvolvido

## Análise e Modelação

- Representação escolhida para o grafo: Lista de Adjacência  
- Tipo de grafo: Direcionado, Ponderado  
- Dados das Adjacências: Arestas com pesos  
- Preservação do grafo:  
 - Tipo de preservação: Binária  
 - Registros usados: Estruturas de dados para vértices e arestas  
 - Representação dos registros no arquivo: Estruturas binárias com informações de vértices e conexões  
 - Regras para a preservação: Estruturas devem ser mantidas consistentes durante as operações de leitura e escrita

## Implementação

A implementação foi realizada em C, com o uso de arquivos de cabeçalho (.h) e arquivos de código fonte (.c). Abaixo estão alguns detalhes da implementação:

### Excertos de Código

#### structure.h

#ifndef STRUCTURE\_H  
#define STRUCTURE\_H  
  
/\*\*  
 \* @file structure.h  
 \* @brief Defines the data structures for a graph using an adjacency list representation.  
 \*  
 \* This file contains definitions for the Node and Graph structures that represent a graph   
 \* in adjacency list format, which is suitable for various graph algorithms.  
 \*/  
  
/\*\*  
 \* @brief Represents a node in the adjacency list of a graph.  
 \*  
 \* Each node corresponds to an edge in the graph, containing the destination vertex,  
 \* the weight of the edge, and a pointer to the next node in the adjacency list.  
 \*/  
typedef struct Node {  
 int destination; ///< Destination vertex of the edge.  
 int weight; ///< Weight of the edge.  
 struct Node\* next; ///< Pointer to the next node in the adjacency list.  
} Node;  
  
/\*\*  
 \* @brief Represents a graph structure using an adjacency list.  
 \*  
 \* This structure contains the total number of vertices and an array of pointers to Node,  
 \* where each pointer corresponds to the head of the adjacency list for each vertex.  
 \*/  
typedef struct {  
 int vertices; ///< Total number of vertices in the graph.  
 Node\*\* adjList; ///< Pointer to an array of pointers to Node, representing the adjacency list for each vertex.  
} Graph;  
  
#endif // STRUCTURE\_H

#### functions.h

#ifndef FUNCTIONS\_H  
#define FUNCTIONS\_H  
  
#include "structure.h"  
#include <stdbool.h>  
  
/\*\*  
 \* @brief Initializes a graph with a specified number of vertices.  
 \*   
 \* @param graph Pointer to the Graph structure.  
 \* @param vertices Number of vertices in the graph.  
 \* @return bool Returns true if initialization was successful, false if memory allocation failed.  
 \*/  
bool initializeGraph(Graph\* graph, int vertices);  
  
/\*\*  
 \* @brief Adds an edge to the graph between two vertices with a specified weight.  
 \*   
 \* @param graph Pointer to the Graph structure.  
 \* @param source Source vertex of the edge.  
 \* @param destination Destination vertex of the edge.  
 \* @param weight Weight of the edge.  
 \* @return bool Returns true if the edge was successfully added, false if memory allocation failed.  
 \*/  
bool addEdge(Graph\* graph, int source, int destination, int weight);  
  
/\*\*  
 \* @brief Lists all the vertices of the graph along with their respective edges and weights.  
 \*   
 \* @param graph Pointer to the Graph structure.  
 \* @return bool Returns true if the listing was successful, false if the graph is empty.  
 \*/  
bool listGraph(Graph\* graph);  
  
/\*\*  
 \* @brief Performs a Depth-First Search (DFS) starting from a specified vertex, marking all visited vertices.  
 \*   
 \* @param graph Pointer to the Graph structure.  
 \* @param vertex Initial vertex for the DFS.  
 \* @param visited Array of integers representing the visited vertices.  
 \* @param total Pointer to an integer that accumulates the total weight of visited edges.  
 \* @return bool Returns true if the search was performed successfully, false if there was an error due to invalid parameters.  
 \*/  
bool DFS(Graph\* graph, int vertex, int visited[], int\* total);  
  
/\*\*  
 \* @brief Conducts a complete DFS traversal starting from a specified initial vertex, printing the result and the total weight.  
 \*   
 \* @param graph Pointer to the Graph structure.  
 \* @param startVertex Initial vertex for the DFS traversal.  
 \*/  
void DFSTraversal(Graph\* graph, int startVertex);  
  
#endif // FUNCTIONS\_H

#### functions.c

/\*\*  
 \* @file funcoes.c  
 \* @brief Implementations of graph operations such as initialization, edge addition, listing, and depth-first search traversal.  
 \*  
 \* This file contains functions for managing a graph structure, including initializing a graph, adding edges, listing the graph's adjacency list,  
 \* performing a depth-first search, and traversing the graph starting from a specified vertex.  
 \*/  
  
#include "structure.h"  
#include <stdlib.h>  
#include <stdbool.h>  
  
/\*\*  
 \* @brief Initializes a graph with a specified number of vertices.  
 \*   
 \* @param graph Pointer to the graph to be initialized.  
 \* @param vertices Number of vertices in the graph.  
 \* @return bool True if the graph was successfully initialized, False if memory allocation failed.  
 \*/  
bool initializeGraph(Graph\* graph, int vertices) {  
 graph->vertices = vertices;  
 graph->adjList = malloc(vertices \* sizeof(Node\*));  
 if (graph->adjList == NULL) {  
 return false;  
 }  
 for (int i = 0; i < vertices; ++i) {  
 graph->adjList[i] = NULL;  
 }  
 return true;  
}  
  
/\*\*  
 \* @brief Adds an edge to the graph.  
 \*   
 \* @param graph Pointer to the graph.  
 \* @param source Source vertex of the edge.  
 \* @param destination Destination vertex of the edge.  
 \* @param weight Weight of the edge.  
 \* @return bool True if the edge was successfully added, False if memory allocation failed.  
 \*/  
bool addEdge(Graph\* graph, int source, int destination, int weight) {  
 Node\* newNode = malloc(sizeof(Node));  
 if (newNode == NULL) {  
 return false;  
 }  
 newNode->destination = destination;  
 newNode->weight = weight;  
 newNode->next = NULL;  
  
 if (graph->adjList[source] == NULL) {  
 graph->adjList[source] = newNode;  
 } else {  
 Node\* current = graph->adjList[source];  
 while (current->next != NULL) {  
 current = current->next;  
 }  
 current->next = newNode;  
 }  
 return true;  
}  
  
/\*\*  
 \* @brief Lists the graph by printing its adjacency list.  
 \*   
 \* @param graph Pointer to the graph.  
 \* @return bool True if the graph is not empty, False otherwise.  
 \*/  
bool listGraph(Graph\* graph) {  
 if (graph == NULL || graph->adjList == NULL) {  
 return false;  
 }  
 printf("Graph:\n");  
 for (int i = 0; i < graph->vertices; ++i) {  
 printf("Vertex %d:", i);  
 Node\* current = graph->adjList[i];  
 while (current != NULL) {  
 printf(" -> %d(%d)", current->destination, current->weight);  
 current = current->next;  
 }  
 printf("\n");  
 }  
 return true;  
}  
  
/\*\*  
 \* @brief Performs a Depth-First Search (DFS) starting from a given vertex.  
 \*   
 \* @param graph Pointer to the graph.  
 \* @param vertex The starting vertex for the DFS.  
 \* @param visited Array to track visited vertices.  
 \* @param total Pointer to sum the weights of visited edges.  
 \* @return bool True if the operation was successful, False if any parameter is invalid.  
 \*/  
bool DFS(Graph\* graph, int vertex, int visited[], int\* total) {  
 if (graph == NULL || graph->adjList == NULL || visited == NULL || total == NULL) {  
 return false;  
 }  
 visited[vertex] = 1;  
 printf("%d ", vertex);  
 Node\* current = graph->adjList[vertex];  
 while (current != NULL) {  
 if (!visited[current->destination]) {  
 \*total += current->weight;  
 DFS(graph, current->destination, visited, total);  
 }  
 current = current->next;  
 }  
 return true;  
}  
  
/\*\*  
 \* @brief Traverses the graph using DFS starting from a specified initial vertex and prints the total weight of the traversed edges.  
 \*   
 \* @param graph Pointer to the graph.  
 \* @param startVertex The starting vertex for the DFS traversal.  
 \*/  
void DFSTraversal(Graph\* graph, int startVertex) {  
 int\* visited = malloc(graph->vertices \* sizeof(int));  
 for (int i = 0; i < graph->vertices; ++i) {  
 visited[i] = 0;  
 }  
 int total = 0;  
  
 printf("DFS starting from vertex %d: ", startVertex);  
 DFS(graph, startVertex, visited, &total);  
 printf("\nTotal weight: %d", total);  
 printf("\n");  
  
 free(visited);  
}

#### main.c

/\*\*  
 \* @file main.c  
 \* @brief Main program to demonstrate graph operations.  
 \*  
 \* This program reads graph data from a file, initializes a graph structure, adds edges based on the file data,  
 \* lists the graph, performs a DFS traversal from a specified vertex, and finally cleans up the allocated resources.  
 \*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <stdbool.h>  
#include "structure.h"  
  
/\*\*  
 \* @brief Main function that orchestrates reading graph data, initializing graph, adding edges, and performing traversals.  
 \*   
 \* @return int Returns 0 on success, 1 on file open error, and 0 on other failures with cleanup.  
 \*/  
int main() {  
 Graph graph; ///< Graph instance to hold the adjacency list.  
  
 FILE\* file; ///< File pointer for reading the graph data.  
 char line[100]; ///< Buffer to hold each line of input from the file.  
  
 // Open the file containing the graph data.  
 file = fopen("dados.txt", "r");  
 if (file == NULL) {  
 perror("Error opening file");  
 return 1;  
 }  
  
 // Count the number of vertices in the graph by reading line counts.  
 int nVertices = 0;  
 while (fgets(line, sizeof(line), file)) {  
 nVertices++;  
 }  
  
 // Reset the file pointer to the beginning of the file for actual data processing.  
 fseek(file, 0, SEEK\_SET);  
  
 // Initialize the graph with the counted vertices.  
 bool success = initializeGraph(&graph, nVertices);  
 if(!success) {  
 return 0;  
 }  
   
 // Read the file again to populate the adjacency list.  
 int i = 0;  
 char\* value;  
 while (fgets(line, sizeof(line), file)) {  
 value = strtok(line, ";");  
 int j = 0;  
 while (value != NULL && j < nVertices) {  
 int weight = atoi(value);  
 success = addEdge(&graph, i, j, weight);  
 if(!success) {  
 return 0;  
 }  
 value = strtok(NULL, ";");  
 j++;  
 }  
 i++;  
 }  
  
 // Close the file as it's no longer needed.  
 fclose(file);  
  
 // List all vertices with their edges and weights.  
 success = listGraph(&graph);  
 if(!success) {  
 return 0;  
 }  
   
 // Perform a DFS traversal starting from vertex 0.  
 int startVertex = 0;  
 success = DFSTraversal(&graph, startVertex);  
 if(!success) {  
 return 0;  
 }  
   
 // Clean up dynamically allocated memory for each node.  
 for (i = 0; i < graph.vertices; ++i) {  
 Node\* current = graph.adjList[i];  
 while (current != NULL) {  
 Node\* temp = current;  
 current = current->next;  
 free(temp);  
 }  
 }  
 // Free the adjacency list array itself.  
 free(graph.adjList);  
  
 return 0;  
}

#### matriz.txt

7;53;183;439;863  
497;383;563;79;973  
287;63;343;169;583  
627;343;773;959;943  
767;473;103;699;303

# 4 - Análise de Resultados/Reflexão

- Abordagens exploradas, mas deixadas de lado: Implementação de grafos usando matriz de adjacência devido à ineficiência de espaço.  
- Considerações sobre decisões tomadas: A escolha por listas de adjacência foi devido à eficiência de espaço e tempo para grafos esparsos.

# 5 - Conclusões e Trabalho Futuro

- O trabalho permitiu a implementação eficiente de grafos como tambem as suas operações principais.  
- Trabalho a desenvolver no futuro: Implementação de novas operações de grafos, otimização de algoritmos(algoritmo de dijkstra,dfs,bfs) e expansão do suporte a grafos ponderados e não direcionados.

# Referências Bibliográficas

https://github.com/luferIPCA/EDA-2023-2024.git

Chagpt4o

<https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/>

<https://graphonline.ru/>

# Anexos

## Anexo I - Enunciado do Trabalho Completo

