****Projeto estruturas de dados avançados

Parte II - Grafos

João Pedro Júnior Barbosa 27964

*Orientador*

Luís Gonzaga Martins Ferreira

Projeto apresentado  
ao Instituto Politécnico do Cávado e do Ave

para obtenção do Grau de Mestre em LESI

maio, 2024

Declaração

Nome: João Pedro Júnior Barbosa

Endereço eletrónico:a27964@alunos.ipca.pt

Título do Projeto: Projeto Estruturas De Dados Avançados (Grafos)

Orientador: Luís Gonzaga Martins Ferreira

Ano de conclusão: 05,2024

Designação do Curso de Mestrado: Mestrado em LESI

Nos exemplares das Dissertações /Projetos/ Relatórios de Estágio de mestrado ou de outros trabalhos entregues para prestação de Provas Públicas, e dos quais é obrigatoriamente enviado exemplares para depósito legal, deve constar uma das seguintes declarações:

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA DISSERTAÇÃO/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA DISSERTAÇÃO/TRABALHO (indicar, caso tal seja necessário, nº máximo de páginas, ilustrações, gráficos, etc.), APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

DE ACORDO COM A LEGISLAÇÃO EM VIGOR, NÃO É PERMITIDA A REPRODUÇÃO DE QUALQUER PARTE DESTA DISSERTAÇÃO/TRABALHO

Instituto Politécnico do Cávado e do Ave, \_\_\_/\_\_\_/\_\_\_\_\_\_

Assinatura: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ÍNDICE

ÍNDICE III

1.Introdução 1

Enquadramento 1

Funcionalidades a Implementar 1

Documentação e Testes 2

2.Trabalho desenvolvido 3

Análise e Modelação: 3

Representação e tipo de grafo 3

Preservação do Grafo 4

Implementação 6

Alínea 1.1: Definir a Estrutura de Dados GR 6

Alínea 1.2: Implementação de funções básicas 7

Alínea 2: Carregamento de dados 8

Alínea 3: Procura em Profundidade/Largura, caminhos possíveis e soma de Vertices 12

Alínea 4: Soma máxima e caminhos entre 2 vertices 14

3.Análise Refletiva 16

Abordagens exploradas, mas deixadas de lado 16

Observações 16

4.Conclusões 17

6.Referências Bibliográficas 18

7.Anexos 19

# 1.Introdução

### 

### Enquadramento

Nesta segunda fase do projeto, o objetivo é aplicar conceitos avançados de teoria dos grafos e programação em C para resolver um problema computacional de alta complexidade. O desafio envolve estruturas de dados, algoritmos de procura e técnicas de otimização para calcular o somatório máximo de inteiros numa matriz arbitrária, respeitando regras específicas de conexão entre os elementos.

### Funcionalidades a Implementar

1. Definição da Estrutura de Dados do Grafo (GR)

• Uma estrutura de dados para representar um grafo orientado;

• Suportar um número variável de vértices;

• Incluir funções para criação, adição e remoção de vértices e arestas.

2. Modelagem do Problema com Grafos

• Representar cada elemento da matriz como um vértice do grafo;

• As arestas entre vértices devem indicar a possibilidade de somar dois elementos adjacentes na matriz;

• Permitir configuração pelo utilizador das regras de conexão (por exemplo, conexões apenas na mesma linha ou coluna).

3. Carregamento de Dados a partir de um Ficheiro de Texto

• Carregar uma matriz de inteiros a partir de um ficheiro de texto, respeitando qualquer Dimensão;

• Os valores na matriz serão separados por ponto e vírgula.

Exemplo de formato:

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. Operações de Manipulação de Grafos

• Implementar algoritmos de busca em profundidade (DFS) ou largura (BFS) para identificar todos os caminhos possíveis que atendam às regras de conexão;

• Desenvolver uma função para calcular a soma dos valores dos vértices num dado caminho.

5. Encontrar o Caminho com a Maior Soma

• Utilizar as estruturas e algoritmos desenvolvidos para encontrar o caminho que proporciona a maior soma possível dos inteiros no grafo, seguindo a regra de conexão estabelecida;

• O programa deve fornecer tanto a soma máxima quanto o caminho correspondente (ou caminhos, se houver múltiplos com a mesma soma máxima).

### Documentação e Testes

• Documentar extensivamente o código, explicando a lógica das principais funções e decisões de implementação;

• Executar casos de teste com matrizes de diferentes dimensões e complexidades para demonstrar a eficiência da solução.

# 2.Trabalho desenvolvido

### Análise e Modelação:

#### Representação e tipo de grafo

Para resolver o problema de encontrar o somatório máximo de inteiros a partir de uma matriz, utilizei a estrutura de dados de grafos. A modelação do problema e a representação dos dados são fundamentais para a implementação eficiente do algoritmo. Aqui está a análise detalhada das escolhas de representação:

**Estrutura de Dados**

• Grafo: O grafo é representado por uma estrutura que contém uma lista de vértices. Esta lista é

gerenciada por um apontador que aponta para o início da lista de vértices.

• Vértice: A estrutura do vértice contém um identificador único, **id**, um apontador para o próximo vértice na lista de vértices, **proxvertice** e um apontador para a lista de adjacências, **proxAdj**.

• Adjacência: A estrutura de adjacência representa uma aresta entre dois vértices no grafo. Contém o identificador do vértice de destino, **id**, o peso da aresta, **peso**, que corresponde ao valor do elemento da matriz) e um apontador para a próxima adjacência, **next**.

**Tipo de Grafo**

Para este problema, o tipo de grafo escolhido é um grafo orientado.

• Grafo Orientado: As arestas têm uma direção e significa que a relação de conexão entre os vértices é unidirecional. Isto é apropriado para representar a soma de elementos adjacentes da matriz onde a direção da soma importa (por exemplo, de um elemento para o próximo numa linha ou coluna).

• Peso das Arestas: O peso de cada aresta corresponde ao valor do elemento da matriz que está a ser somado.

**Dados das Adjacências**

Uma imagem com texto, captura de ecrã, número, Tipo de letra

Descrição gerada automaticamente

Figura 1 Dados das Adjacências

#### Preservação do Grafo

Para garantir que o grafo possa ser preservado e carregado posteriormente, implementei a função **GuardaGrafoBinario** para armazenar o grafo num ficheiro binário. Esta abordagem garante que a estrutura do grafo, incluindo os vértices e as arestas, seja preservada de maneira eficiente.

**Tipo de Preservação**

• **Binária**: A escolha pelo formato binário deve-se à eficiência em termos de espaço e velocidade de leitura/escrita. Em comparação com formatos de texto, os ficheiros binários ocupam menos espaço e são mais rápidos para processar, além de serem muito mais seguros.

**Registos Usados**

• **Vértices** **e** **Arestas:** A estrutura do grafo é representada pelos vértices e as suas adjacências (arestas). Cada vértice é registado juntamente com as suas adjacências.

• **Número de Adjacências:** Para cada vértice, armazenei também o número de adjacências para facilitar a leitura posteriormente.

**Representação dos Registos no Ficheiro**

Cada vértice e as suas adjacências são escritos no ficheiro binário da seguinte maneira:

**1.** **Identificador do Vértice (int):** Representa o identificador único do vértice.

**2. Número de Adjacências (int):** Indica quantas arestas (adjacências) o vértice possui.

**3. Adjacências:** Cada adjacência é representada por dois inteiros:

• **Identificador do Vértice Adjacente (int):** O identificador do vértice de destino da aresta.

• **Peso da Aresta (int):** O peso associado à aresta.

**Regras para a Preservação**

**1. Validação de Entrada:** Verificar se o grafo e o nome do ficheiro não são nulos, antes de proceder.

**2. Abertura do Ficheiro:** Abrir o ficheiro no modo binário para escrita “**wb**”. Caso não consiga abrir, retornar falso.

**3. Escrita de Dados:**

• Percorrer todos os vértices no grafo;

• Para cada vértice, escrever o seu identificador e o número de adjacências;

• Percorrer todas as adjacências do vértice, escrevendo o identificador do vértice adjacente e o peso da aresta;

**4. Fecho do Ficheiro:** Fechar o ficheiro após a escrita completa dos dados;

**5. Retorno:** Retornar verdadeiro se a operação for bem-sucedida, caso contrário, retornar falso.

Uma imagem com file, Simetria, círculo, origami

Descrição gerada automaticamente

Figura 2 Exemplo do grafo carregado

### Implementação

#### Alínea 1.1: Definir a Estrutura de Dados GR

Definir uma estrutura de dados GR para representar um grafo. Esta estrutura deve ser capaz de representar grafos dirigidos e deve suportar um número variável de vértices.

typedef struct Adjacencias {

int id;

int peso;

struct Adjacencias\* next;

}Adjacencias;

typedef struct Vertices {

int id;

Adjacencias\* proxAdj;

struct Vertices\* proxVertice;

}Vertices;

typedef struct Grafo {

Vertices\* inicioGrafo;

}Grafo;

Uma imagem com texto, Tipo de letra, diagrama, file

Descrição gerada automaticamente

Figura 3 Representação Da Estrutura de Dados

#### 

#### Alínea 1.2: Implementação de funções básicas

A implementação deve incluir funções básicas para a criação do grafo, adição e remoção de vértices e arestas.

#pragma region Vertices

Vertices\* CriaVertice(int id);

bool ExisteVertice(Vertices\* inicio, int id);

Vertices\* InsereVertice(Vertices\* vertices, Vertices\* novo, bool\* res);

Vertices\* OndeEstaVertice(Vertices\* inicio, int id);

void destroiAdjacencias(Adjacencias\* destroir);

Adjacencias\* EliminaAllAdj(Adjacencias\* listaAdjacencias, bool\* res);

Adjacencias\* EliminaAdj(Adjacencias\* listaAdjacencias, int codAdj, bool\* res);

Vertices\* EliminaVertice(Vertices\* vertices, int codVertice, bool\* res);

Vertices\* EliminaAdjacenciaVertices(Vertices\* vertices, int codAdj, bool\* res);

#pragma endregion

#pragma region Arestas

Adjacencias\* NovaAdjacencia(int id, int peso);

Adjacencias\* InsereAdj(Adjacencias\* listaAdja, int idDestino, int peso);

void DestroiAdjacencia(Adjacencias\* ptAdjacent);

Adjacencias\* EliminaTodasAdjacencias(Adjacencias\* listAdj, bool\* res);

#pragma endregion

#pragma region Grafo

Grafo\* CriaGrafo();

bool ExisteVerticeGrafo(Grafo\* g, int idVertice);

Grafo\* InsereVerticeGrafo(Grafo\* g, Vertices\* novo, int\* res);

Vertices\* OndeEstaVerticeGrafo(Grafo\* g, int idVertice);

Grafo\* EliminaVerticeGrafo(Grafo\* g, int codVertice, bool\* res);

Grafo\* EliminaAdjGrafo(Grafo\* g, int origem, int destino, bool\* res);

Grafo\* InsereAdjacenciasGrafo(Grafo\* g, int idOrigem, int idDestino, int peso, bool\* res);

void MostrarGrafo(Vertices\* grafo);

#pragma endregion

#### Alínea 2: Carregamento de dados

Após definir a estrutura de dados GR, pretende-se modelar o problema utilizando grafos. Cada elemento da matriz de inteiros será representado por um vértice no grafo. As arestas entre vértices devem representar a possibilidade de somar dois elementos adjacentes na matriz, sob uma regra de conexão específica que poderá ser configurada pelo utilizador (por exemplo, apenas elementos na mesma linha ou coluna, não permitindo diagonais, ou qualquer outra regra); Carregamento para a estrutura de dados da alínea anterior GR dos dados de uma matriz de inteiros constante num ficheiro de texto. A operação deverá considerar matrizes de inteiros com qualquer dimensão, sendo os valores separados por vírgulas.

Implementei uma função para carregar os dados da matriz de inteiros a partir de um ficheiro de texto e criar o grafo correspondente:

Grafo\* carregarMatrizParaGrafo(char fileName[], int\* numLinhas, int\* numColunas) {

FILE\* fp;

char linha[MAXCHAR];

char\* token;

fp = fopen(fileName, "r");

if (fp == NULL) {

return NULL;

}

\*numLinhas = 0;

\*numColunas = 0;

Grafo\* grafo = CriaGrafo();

if (grafo == NULL) {

fclose(fp);

return NULL;

}

// ciclo para ler cada linha do arquivo

while (fgets(linha, MAXCHAR, fp) != NULL) {

// Incrementa o número de linhas

(\*numLinhas)++;

token = strtok(linha, ";");

int colunaAtual = 0;

while (token != NULL) {

int valor = atoi(token);

if (valor != 0) {

// Verifica se o vértice de origem existe, se não, cria

if (!ExisteVerticeGrafo(grafo, \*numLinhas - 1)) {

Vertices\* novoVertice = CriaVertice(\*numLinhas - 1);

if (novoVertice == NULL) {

fclose(fp);

return NULL;

}

int res;

grafo = InsereVerticeGrafo(grafo, novoVertice, &res);

if (res != 1) {

fclose(fp);

return NULL;

}

}

// Verifica se o vértice de destino existe, se não, cria

if (!ExisteVerticeGrafo(grafo, colunaAtual)) {

Vertices\* novoVertice = CriaVertice(colunaAtual);

if (novoVertice == NULL) {

fclose(fp);

return NULL;

}

int res;

grafo = InsereVerticeGrafo(grafo, novoVertice, &res);

if (res != 1) {

fclose(fp);

return NULL;

}

}

bool res;

grafo = InsereAdjacenciasGrafo(grafo, \*numLinhas - 1, colunaAtual, valor, &res);

if (!res) {

fclose(fp);

return NULL;

}

}

// Avança para a próxima coluna

token = strtok(NULL, ";");

colunaAtual++;

}

if (colunaAtual > \*numColunas) {

\*numColunas = colunaAtual;

}

}

fclose(fp);

return grafo;

}

Implementei uma função para guardar o conteúdo do grafo em binário e uma função para carregar esse conteúdo.

bool GuardaGrafoBinario(Grafo\* grafo, char fileName[]) {

if (grafo == NULL || fileName == NULL) {

return false;

}

FILE\* fp = fopen(fileName, "wb");

if (fp == NULL) {

return false;

}

Vertices\* verticeAtual = grafo->inicioGrafo;

while (verticeAtual != NULL) {

// Escreve o vértice atual no ficheiro binário

if (fwrite(&verticeAtual->id, sizeof(int), 1, fp) != 1) {

fclose(fp);

return false;

}

// Escreve o número de adjacências do vértice atual

int numAdjacencias = 0;

Adjacencias\* adjacenciaAtual = verticeAtual->proxAdj;

while (adjacenciaAtual != NULL) {

numAdjacencias++;

adjacenciaAtual = adjacenciaAtual->next;

}

if (fwrite(&numAdjacencias, sizeof(int), 1, fp) != 1) {

fclose(fp);

return false;

}

// Escreve as adjacências do vértice atual no ficheiro binário

adjacenciaAtual = verticeAtual->proxAdj;

while (adjacenciaAtual != NULL) {

int adjData[2] = { adjacenciaAtual->id, adjacenciaAtual->peso };

if (fwrite(adjData, sizeof(int), 2, fp) != 2) {

fclose(fp);

return false;

}

adjacenciaAtual = adjacenciaAtual->next;

}

verticeAtual = verticeAtual->proxVertice;

}

fclose(fp);

return true;

}

int CarregaGrafoBinario(char fileName[]) {

FILE\* fp = fopen(fileName, "rb");

if (fp == NULL) {

return -1;

}

while (1) {

int idVertice;

if (fread(&idVertice, sizeof(int), 1, fp) != 1) {

if (feof(fp)) break;

fclose(fp);

return -1;

}

int numAdjacencias;

if (fread(&numAdjacencias, sizeof(int), 1, fp) != 1) {

fclose(fp);

return -1;

}

for (int i = 0; i < numAdjacencias; i++) {

int adjData[2];

if (fread(adjData, sizeof(int), 2, fp) != 2) {

fclose(fp);

return -1;

}

}

}

fclose(fp);

return 0;

}

#### Alínea 3: Procura em Profundidade/Largura, caminhos possíveis e soma de Vertices

Implementar operações de manipulação de grafos, incluindo procura em profundidade ou em largura, para identificar todos os caminhos possíveis que atendem às regras de conexão definidas. Desenvolver também uma função para calcular a soma dos valores dos vértices num dado caminho.

Nesta situação, optei pela procura em profundidade com recursividade, uma vez que foi aquele que intendi melhor. Para além disso, a função apresenta todos os caminhos possíveis e a soma dos ID de todos os vértices de todos os caminhos.

Grafo\* ProcuraProfundidadeRec(Grafo\* g, int origem, int destino, bool\* visitado, int\* caminho, int indice, int\* somaCaminhos) {

visitado[origem] = true;

caminho[indice] = origem; // Adiciona o vértice atual ao caminho

indice++;

if (origem == destino) {

// Se o vértice atual é o destino, imprime o caminho

for (int i = 0; i < indice; i++) {

printf("%d ", caminho[i]);

}

printf("\n");

// Calcula a soma dos valores dos vértices no caminho

int soma = 0;

for (int i = 0; i < indice; i++) {

soma += caminho[i];

}

\*somaCaminhos += soma;

}

else {

// Se o vértice atual não é o destino, procura as adjacências

Vertices\* verticeAtual = OndeEstaVerticeGrafo(g, origem);

Adjacencias\* adj = verticeAtual->proxAdj;

while (adj != NULL) {

if (!visitado[adj->id]) {

ProcuraProfundidadeRec(g, adj->id, destino, visitado, caminho, indice, somaCaminhos);

}

adj = adj->next;

}

}

indice--; // Decrementa o índice do caminho para backtracking

visitado[origem] = false; // Marca o vértice atual como não visitado para permitir outros caminhos

return g;

}

Grafo\* ProcuraProfundidade(Grafo\* g, int origem, int destino, int numVertices, int\* soma) {

bool\* visitado = (bool\*)malloc(numVertices \* sizeof(bool));

int\* caminho = (int\*)malloc(numVertices \* sizeof(int));

int somaCaminhos = 0;

for (int i = 0; i < numVertices; i++) {

visitado[i] = false; // Inicializa todos os vértices como não visitados

}

ProcuraProfundidadeRec(g, origem, destino, visitado, caminho, 0, &somaCaminhos);

\*soma = somaCaminhos;

return g;

}

#### Alínea 4: Soma máxima e caminhos entre dois vértices

Utilizar as estruturas e algoritmos desenvolvidos para encontrar o caminho que proporciona a maior soma possível dos inteiros na estrutura GR, seguindo a regra de conexão estabelecida. O programa deve fornecer tanto a soma máxima quanto o caminho.

Grafo\* DFSrec(Grafo\* g, int origem, int destino, bool\* visitado, int\* caminho, int indice, int\* somaCaminhos, int\* somaMaxima, int\* caminhoMaximo, int numVertices) {

visitado[origem] = true;

caminho[indice] = origem; // Armazena o ID do vértice atual no caminho

indice++;

if (origem == destino) {

// Calcula a soma dos pesos das arestas no caminho

int soma = 0;

for (int i = 0; i < indice - 1; i++) {

Vertices\* verticeAtual = g->inicioGrafo;

// Localiza o vértice atual na lista de vértices do grafo

while (verticeAtual != NULL && verticeAtual->id != caminho[i]) {

verticeAtual = verticeAtual->proxVertice;

}

// Verifica se o vértice atual foi encontrado

if (verticeAtual == NULL) {

return NULL;

}

Adjacencias\* adj = verticeAtual->proxAdj;

// Percorre as adjacências do vértice atual

while (adj != NULL) {

if (adj->id == caminho[i + 1]) {

soma += adj->peso;

break;

}

adj = adj->next;

}

}

// Verifica se a soma é maior que a soma máxima encontrada até agora

if (soma > \*somaMaxima) {

\*somaMaxima = soma;

// Atualiza o caminho máximo correspondente

for (int i = 0; i < indice; i++) {

caminhoMaximo[i] = caminho[i];

}

// Define -1 nos elementos restantes do caminho máximo

for (int i = indice; i < numVertices; i++) {

caminhoMaximo[i] = -1;

}

}

}

else {

// Se o vértice atual não é o destino, procura as adjacências

Vertices\* verticeAtual = g->inicioGrafo;

// Localiza o vértice atual na lista de vértices do grafo

while (verticeAtual != NULL && verticeAtual->id != origem) {

verticeAtual = verticeAtual->proxVertice;

}

// Verifica se o vértice atual foi encontrado

if (verticeAtual == NULL) {

return NULL;

}

Adjacencias\* adj = verticeAtual->proxAdj;

// Percorre as adjacências do vértice atual

while (adj != NULL) {

if (!visitado[adj->id]) {

DFSrec(g, adj->id, destino, visitado, caminho, indice, somaCaminhos, somaMaxima, caminhoMaximo, numVertices);

}

adj = adj->next;

}

}

// Backtracking

indice--;

visitado[origem] = false;

return g;

}

void encontrarCaminhoMaiorSoma(Grafo\* g, int origem, int destino, int numVertices) {

bool\* visitado = (bool\*)malloc(sizeof(bool) \* numVertices);

int\* caminho = (int\*)malloc(sizeof(int) \* numVertices);

// Inicializa todos os vértices como não visitados

for (int i = 0; i < numVertices; i++) {

visitado[i] = false;

}

int somaCaminhos = 0;

int somaMaxima = 0;

int\* caminhoMaximo = (int\*)malloc(sizeof(int) \* numVertices);

DFSrec(g, origem, destino, visitado, caminho, 0, &somaCaminhos, &somaMaxima, caminhoMaximo, numVertices);

//Mostra soma máxima e o caminho correspondente

printf("Soma máxima: %d\n", somaMaxima);

printf("Caminho correspondente: ");

for (int i = 0; i < numVertices; i++) {

if (caminhoMaximo[i] != -1) {

printf("%d ", caminhoMaximo[i]);

}

}

printf("\n");

}

# 3.Análise Refletiva

### Abordagens exploradas, mas deixadas de lado

No início do projeto, não estava a utilizar bibliotecas então fazias as funções todas num .c e testava-as no main, porém tinha a intenção de no final dividir as funções em 3 bibliotecas diferentes: uma para os vértices, uma para as arestas e outra para os grafos e foi aí que surgiu o problema. Nas funções dos grafos, eu utilizo muitas funções das arestas e dos vertices em recursividade o que levou a implementar apenas uma biblioteca com todas as funções.

### Observações

Não consegui enviar o ficheiro sem .obj, .exe, .lib, visto que, posteriormente quando abria a solução dizia que não encontrava o projeto da biblioteca e então decidi enviar tudo, uma vez que dessa maneira funcionava perfeitamente.

Para além disso, ao usar este template tem partes que falam em mestrado que não são possíveis de remover.

# 4.Conclusões

Neste trabalho, abordei o problema de calcular o somatório máximo possível de inteiros a partir de uma matriz de inteiros, considerando regras específicas de conexão entre os elementos.

Utilizei conceitos avançados de teoria dos grafos e programação em C para desenvolver uma solução eficiente e flexível.

Comecei por definir uma estrutura de dados para representar o grafo, que suporta grafos orientados e um número variável de vértices. Em seguida, modelei o problema utilizando grafos, onde cada elemento da matriz de inteiros foi representado como uma aresta no grafo.

Implementei operações para adicionar/remover vértices e arestas, bem como funções para importar dados de uma matriz de inteiros a partir de um ficheiro de texto e criar o grafo correspondente.

Além disso, criei funções para guardar o grafo em formato binário e carregá-lo posteriormente. garantindo a preservação dos dados de forma eficiente.

Criei algoritmos de procura em profundidade para identificar todos os caminhos possíveis e calcular a soma dos valores dos vértices num determinado caminho.

Por fim, executei testes com matrizes de diferentes dimensões e complexidades para validar a eficiência da solução desenvolvida, demonstrando a sua capacidade de lidar com uma variedade de cenários.

Em suma, este trabalho não apenas abordou os aspetos teóricos da teoria dos grafos, mas também proporcionou uma implementação prática e robusta, oferecendo uma solução completa para o problema proposto.

# 6.Referências Bibliográficas

* Código produzido nas aulas
* Notes on Data Structures and Programming Techniques (James Aspnes. 2015)
* Data Structures Using C (Reema Thareja, 2nd Editions)
* Fundamentos sobre Grafos (Professor Morais da Silva)

# 7.Anexos

