IPCA INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

ESCOLA SUPERIOR DE TECNOLOGIA

Relatório Técnico - Fase 1 e 2

LICENCIATURA DE SISTEMAS DE INFORMATICA

Unidade Curricular: Estruturas de Dados

Aluno: José Miguel Sousa

Nº31552

ESTRUTURA DE DADOS

RESUMO

O presente relatório descreve o desenvolvimento de um projeto realizado no âmbito da Unidade Curricular de Estruturas de Dados Avançadas (EDA), lecionada no 2.º semestre do 1.º ano da Licenciatura em Engenharia de Sistemas Informáticos. O objetivo central do projeto consiste na criação de uma aplicação em linguagem C capaz de simular a gestão de antenas numa área urbana, tendo em conta as suas coordenadas e frequências, e identificar automaticamente os efeitos nefastos provocados por fenómenos de ressonância entre antenas alinhadas com a mesma frequência.

A solução foi concebida com base na utilização de listas ligadas simples, de forma a garantir uma representação flexível, eficiente e dinâmica dos dados. As funcionalidades implementadas incluem o carregamento de antenas a partir de ficheiros de texto, a inserção e remoção de antenas em tempo de execução, o cálculo automático das localizações com efeito nefasto e a visualização dos dados em formato matricial.

O desenvolvimento seguiu uma metodologia iterativa e modular, promovendo a clareza, reutilização e manutenibilidade do código. Para apoiar a compreensão e manutenção do projeto, a documentação técnica foi gerada com a ferramenta Doxygen.

Este trabalho constitui uma aplicação prática dos conceitos abordados na unidade curricular, destacando-se pela implementação de uma estrutura de dados adaptada ao problema proposto, e reforçando a compreensão do impacto que as estruturas dinâmicas podem ter na resolução de problemas computacionais com cariz espacial e algorítmico.

Índice

Índice de figuras

Introdução

O presente capítulo apresenta o contexto e a motivação que sustentam o desenvolvimento deste trabalho, os seus objetivos principais, a metodologia adotada e a estrutura do documento. O projeto foi desenvolvido no âmbito da Unidade Curricular de Estruturas de Dados Avançadas (EDA), inserida no 2.º semestre do 1.º ano da Licenciatura em Engenharia de Sistemas Informáticos (LESI), da Escola Superior de Tecnologia do IPCA.

1.1 - Motivação

A realização deste trabalho surge no âmbito da disciplina de Estruturas de Dados, com o objetivo de aplicar, de forma prática, os conhecimentos adquiridos sobre listas ligadas e gestão dinâmica de memória. O enunciado proposto apresenta um problema concreto que exige o desenvolvimento de uma aplicação capaz de simular a colocação de antenas de telecomunicações numa matriz bidimensional, analisando a sua disposição e a frequência associada a cada uma.

1.2 - Enquadramento

O projeto insere-se na primeira fase do plano de avaliação da UC de EDA e consiste na resolução de um problema computacional utilizando exclusivamente listas ligadas. A aplicação desenvolvida visa representar as antenas numa estrutura dinâmica e identificar as localizações com efeito nefasto a partir de critérios geométricos definidos.

1.3 - Objetivos

Este projeto tem como principais objetivos:

- Aplicar conceitos de estruturas de dados dinâmicas, nomeadamente listas ligadas simples.
- Implementar um sistema para representar e manipular antenas numa matriz virtual.
- Calcular automaticamente localizações com efeito nefasto com base na ressonância de frequências.
- Representar os dados e resultados de forma clara na consola.
- Estruturar e documentar o código de forma modular e reutilizável.

1.4 - Metodologia de Trabalho

A abordagem seguida teve início com a análise dos requisitos definidos no enunciado do projeto. Após uma tentativa inicial baseada na utilização de uma matriz tridimensional, optou-se pela utilização de listas ligadas, dada a sua maior flexibilidade e eficiência na manipulação de dados dinâmicos.

O desenvolvimento seguiu uma metodologia iterativa e incremental, com divisão em módulos: definição das estruturas de dados, carregamento de antenas, cálculo de efeitos nefastos e funcionalidades de inserção/remover. O código foi modularizado e documentado com recurso ao Doxygen.

1.5 - Plano de Trabalho

O plano de trabalho da Fase 1 incluiu as seguintes etapas:

1. Definição das estruturas de dados e variáveis globais.
2. Leitura e armazenamento das antenas a partir de ficheiros.
3. Implementação das funcionalidades de inserção e remoção.
4. Cálculo e representação dos efeitos nefastos.
5. Testes com diferentes matrizes e refatoração do código.

O plano de trabalho da Fase 2 incluiu as seguintes etapas:

1. Definição das estruturas de dados e variáveis globais.
2. Leitura e armazenamento das antenas a partir de ficheiros.
3. implementação de operações sobre o grafo.
4. Desenvolvimento de algoritmos de exploração do grafo.
5. Execução de testes com matrizes de diferentes tamanhos e configurações.

1.6 - Estrutura do Documento

O presente documento está organizado da seguinte forma:

FASE 1

- **Capítulo 1**: Introdução ao projeto, objetivos, metodologia e plano de trabalho.
- **Capítulo 2**: Descrição das estruturas de dados e especificações da aplicação.
- **Capítulo 3**: Implementação das funcionalidades e explicação do código.
- **Capítulo 4**: Função Main
- **Capítulo 5**: Análise dos resultados e exemplos de execução.
- **Conclusão**

Fase 2

- **Objetivos:** Objetivos da fase 2.
- **Capítulo 1**: Analise e Especificação.
- **Capítulo 2**: Descrição das Funções do Programa.
- **Capítulo 3**: Função Main.
- **Capítulo 4**: Resultados e exemplos de execução.
- **Conclusão**

FASE 1

Capítulo 1 - Abordagem Inicial e Mudança de Estratégia

Este capítulo descreve a evolução da abordagem adotada para representar e gerir as antenas no sistema, iniciando com uma solução baseada em matrizes dinâmicas e culminando na adoção de uma estrutura de lista ligada.

Inicialmente, considerou-se a utilização de uma **matriz tridimensional de ponteiros** (antena***) para representar as antenas de forma bidimensional, permitindo acesso direto por coordenadas. Contudo, esta abordagem revelou-se ineficiente devido ao **elevado consumo de memória** — mesmo em posições sem antenas — e à **complexidade na gestão da memória dinâmica**, levando à sua substituição.

Como alternativa, foi adotada uma **estrutura de lista ligada**, onde cada antena é representada por um nó contendo a sua posição, frequência e um ponteiro para o próximo elemento. Esta abordagem revelou-se mais eficiente, flexível e de mais fácil manutenção, permitindo um desenvolvimento mais robusto da aplicação.

1.1-Abordagem Inicial

Numa fase inicial, considerou-se a utilização de uma matriz dinâmica de ponteiros para representar as antenas, estruturada sob a forma de um ponteiro triplo do tipo antena*** matriz. Esta abordagem visava facilitar o acesso direto às antenas com base nas suas coordenadas bidimensionais e permitir uma visualização intuitiva da distribuição espacial no terminal.

Para suportar esta ideia, foi desenvolvida a função ***criarMatrizAntenas***(int maxLinhas, int maxColunas), cuja responsabilidade era alocar dinamicamente uma matriz tridimensional de ponteiros. Cada célula da matriz continha um ponteiro para uma estrutura do tipo antena, que armazenava os atributos fundamentais: posição (linha e coluna), frequência e um ponteiro adicional next, com o intuito de possibilitar extensões futuras, como o armazenamento de múltiplas antenas na mesma posição.

```
1  antena*** criarMatrizAntenas(int maxLinhas, int maxColunas) {
2      antena*** matriz = (antena***)malloc(maxLinhas * sizeof(antena**));
3      if (!matriz) {
4          return NULL;
5      }
6
7      for (int i = 0; i < maxLinhas; i++) {
8          matriz[i] = (antena**)malloc(maxColunas * sizeof(antena*));
9          if (!matriz[i]) {
10             return NULL;
11         }
12         for (int j = 0; j < maxColunas; j++) {
13             matriz[i][j] = (antena*)malloc(sizeof(antena));
14             if (!matriz[i][j]) {
15                 return NULL;
16             }
17             matriz[i][j]->linha = i;
18             matriz[i][j]->coluna = j;
19             matriz[i][j]->frequencia = ' ';
20             matriz[i][j]->next = NULL;
21         }
22     }
23
24     return matriz;
25 }
```

Apesar de esta solução permitir um acesso direto e eficiente às antenas através das coordenadas da matriz, rapidamente se verificou que a complexidade associada à gestão de memória e à manipulação da estrutura ultrapassava os benefícios esperados. A alocação de um volume elevado de memória, mesmo para posições não ocupadas, resultava num desperdício significativo de recursos. Adicionalmente, a necessidade de garantir a correta libertação da memória para cada nível da matriz aumentava substancialmente o risco de erros e a dificuldade de manutenção.

Neste contexto, e considerando os objetivos do trabalho, concluiu-se que esta abordagem não se adequava de forma eficiente à resolução do problema, tendo sido abandonada em favor de uma solução mais simples e flexível.

1.2-Abordagem Final Adotada

Após a análise das limitações da abordagem baseada em matrizes dinâmicas, optou-se pela implementação de uma estrutura de **lista ligada**, mais eficiente e adequada à representação dinâmica do conjunto de antenas.

Nesta nova abordagem, cada antena é representada por um nó da lista ligada, contendo os seguintes campos: linha, coluna, frequência e um ponteiro para o próximo elemento da lista. A criação de novos nós é feita através da função ***CriarAntena***, que aloca dinamicamente memória para uma estrutura do tipo antena e inicializa os seus campos com os valores fornecidos.

```
1  antena* CriarAntena(int coluna, int linha, char frequencia) {
2      antena* aux;
3      aux = (antena*)malloc(sizeof(antena));
4      if (aux != NULL) {
5          aux->coluna = coluna;
6          aux->linha = linha;
7          aux->frequencia = frequencia;
8          aux->next = NULL;
9      }
10      return aux;
11  }
```

Em suma, a estrutura de lista ligada proporcionou uma solução mais simples, eficiente e alinhada com os princípios fundamentais da programação dinâmica em C. A clareza na organização do código e a redução da complexidade da gestão de memória permitiram um desenvolvimento mais robusto e modular da aplicação.

Capítulo 2 - Análise e Especificação

Este capítulo apresenta os requisitos funcionais do sistema, as estruturas de dados definidas para representar a informação, bem como uma descrição das variáveis globais utilizadas e um diagrama simplificado da estrutura do programa

2.1- Requisitos do Sistema

Os principais requisitos funcionais definidos para esta fase do projeto foram:

- Ler uma matriz de caracteres a partir de um ficheiro de texto e converter a informação em listas ligadas.
- Representar antenas com as suas coordenadas e frequência de emissão.
- Calcular automaticamente as localizações com efeito nefasto, com base na ressonância entre antenas da mesma frequência.
- Permitir inserção e remoção dinâmica de antenas.
- Apresentar os dados em formato matricial na consola.

2.2- Estruturas de Dados

Foram definidas duas estruturas principais, ambas implementadas através de listas ligadas simples:

- **antena**: Representa uma antena com os atributos linha, coluna, frequência (carácter), e um ponteiro para o próximo elemento.

```
1  typedef struct antena {
2      int coluna;
3      int linha;
4      char frequencia;
5      struct antena* next;
6  } antena;
```

- **nefasto**: Representa uma localização afetada por efeito nefasto, com as coordenadas nx e ny, e um ponteiro para o próximo elemento.

```
7 typedef struct nefasto {
8      int nx;
9      int ny;
10      struct nefasto* next;
11  } nefasto;
```

Estas estruturas encontram-se declaradas num ficheiro de cabeçalho próprio, permitindo a sua utilização em diferentes partes do programa.

2.3-Variaveis Globais

Foram utilizadas as seguintes variáveis globais:

```
int MAX_LINHAS = 0;
int MAX_COLUNAS = 0;
```

Estas variáveis armazenam as dimensões máximas da matriz de antenas, e são utilizadas para validar coordenadas e percorrer corretamente o espaço representado.

As variáveis globais são variáveis declaradas fora de qualquer função e que podem ser acedidas por qualquer parte do programa. Ao contrário das variáveis locais, que existem apenas dentro da função onde foram definidas, as variáveis globais mantêm o seu valor ao longo de toda a execução do programa.

2.4-Assinatura das funções

As assinaturas das funções usadas encontram-se no ficheiro ASSFunco.h.

```c
#pragma once
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h>
#include "Structs.h"


antena* CriarAntena(int coluna, int linha, char frequencia);
antena* CarregarAntenasDoFicheiro(const char* nome_ficheiro);
antena* procurarantena(antena* h, int l, int c);
antena* insereOrdenado(antena* inicio, antena* novo);
antena* removerantena(antena* h, int linha, int coluna);
void mostrarMatrizAntenas(antena* lista_antenas);
void libertarListaAntenas(antena* lista);
nefasto* criarEfeitoNefasto(int x, int y);
nefasto* calcularEfeitoNefastoFinal(antena* lista);
void mostrarMatrizNefastos(antena* lista_antenas, nefasto* lista_nefastos);
void libertarListaNefasto(nefasto* lnefasto);
```

Figura 1 Assinatura das Funções

2.5-Classe Diagram

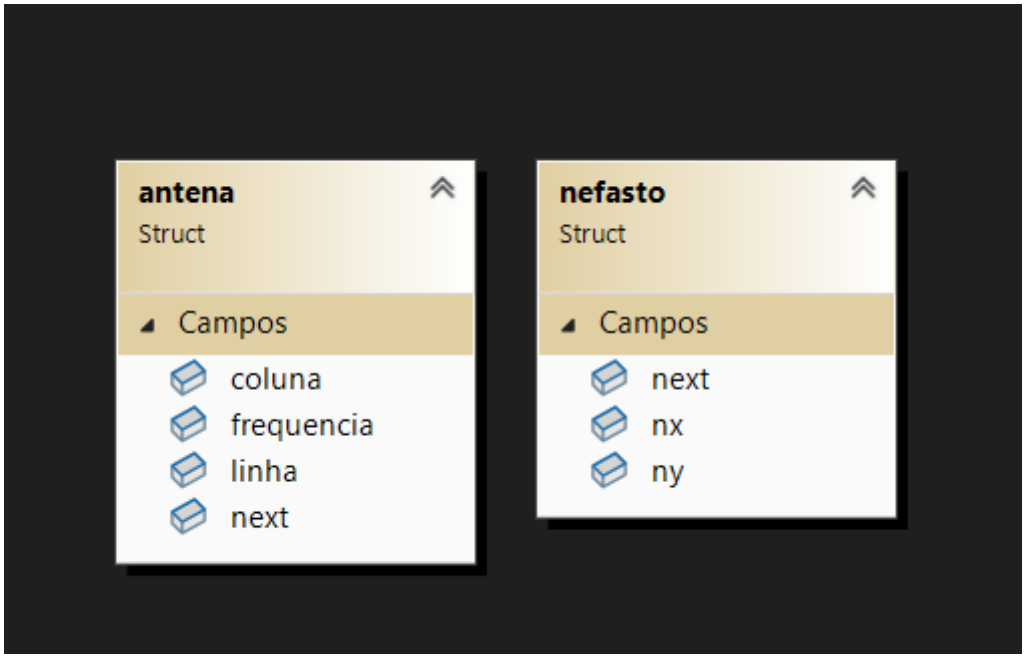


Figura 2 Classe Diagram

O diagrama de classes (Figura 2 Classe Diagram) representa a estrutura das principais entidades do projeto:

- **antena**: Estrutura que armazena a informação de cada antena, incluindo as suas coordenadas (**coluna** e **linha**), a **frequência** e um ponteiro (**next**) que liga ao próximo elemento da lista ligada.
- **nefasto**: Estrutura que representa as localizações afetadas pelo efeito nefasto, armazenando as coordenadas (**nx** e **ny**) e um ponteiro (**next**) para permitir a ligação entre os diferentes elementos na lista ligada.

Cada antena pode ser ligada à seguinte através do ponteiro next, o mesmo acontecendo com os efeitos nefastos. Esta abordagem permite percorrer dinamicamente todas as antenas e localizações afetadas, sem necessidade de recorrer a estruturas estáticas como matrizes.

Capítulo 3 - Funções do programa

Neste capítulo é apresentada a implementação das funcionalidades principais do projeto. Cada função foi desenvolvida com base nas estruturas de dados previamente especificadas e respeita os requisitos definidos na fase de análise. As operações foram estruturadas de forma modular, permitindo a separação lógica entre a criação, manipulação e visualização dos dados. Para além das funções de leitura e escrita, são também descritos os algoritmos para o cálculo dos efeitos nefastos e a gestão da memória. Abaixo, destacam-se os pontos mais importantes de cada função implementada:

3.1-Função criar Antena

A função ***CriarAntena*** tem como finalidade alocar dinamicamente memória para um novo nó da lista ligada de antenas. Esta operação é essencial para garantir a criação de uma estrutura válida que armazene a informação relativa à localização e à frequência de uma antena. Após a alocação, os campos da estrutura antena são devidamente inicializados, ficando o nó pronto a ser inserido na lista, respeitando a coerência e integridade dos dados mantidos pelo sistema.

```
antena* CriarAntena(int coluna, int linha, char frequência)
```

Esta função assegura uma alocação correta da memória e define os valores iniciais das coordenadas e da frequência da antena. Embora seja uma função simples, ela é essencial, pois garante a integridade da estrutura ligada.

3.2-Carregamento de Antenas a partir de Ficheiro

A função ***CarregarAntenasDoFicheiro*** permite a leitura de um ficheiro de texto que representa uma matriz de caracteres, onde os pontos (.) correspondem a espaços vazios e os outros caracteres a antenas:

```
antena* CarregarAntenasDoFicheiro(const char* nome_ficheiro)
```

Durante a leitura, cada antena identificada é transformada numa estrutura antena e adicionada à lista ligada. O controlo das dimensões da matriz também é assegurado, através das variáveis globais MAX_LINHAS e MAX_COLUNAS que são definidas aqui.

3.3-Função Procura Antena

A função ***procurarantena*** tem como objetivo localizar uma antena específica na lista ligada fornecida, com base nas suas coordenadas de linha e coluna. Esta função percorre sequencialmente a lista ligada e compara, em cada iteração, os valores das coordenadas do nó atual com os fornecidos como parâmetros. Caso seja encontrada uma correspondência, a função devolve um ponteiro para o nó respetivo; caso contrário, retorna NULL, indicando que a antena procurada não se encontra na estrutura.

```
antena* procurarantena(antena* h, int l, int c)
```

3.4-Função Inserir Antena através de coordenadas

A função ***insereOrdenado*** é uma das principais funções deste projeto, foi concebida para inserir uma nova antena na lista encadeada de antenas, de forma a manter a ordem crescente com base nas coordenadas (linha e coluna). A ordenação é crucial para facilitar operações futuras, como a pesquisa e a remoção, e para garantir a integridade dos dados.

Aspetos relevantes a destacar da função:

Verificação de duplicação:
Antes de proceder à inserção, a função utiliza a função ***procurarantena*** para verificar se já existe uma antena com as mesmas coordenadas (linha e coluna). Se for encontrada, o novo nó é descartado (libertando a memória associada) para evitar duplicações e preservar a integridade da estrutura de dados.

I**nserção quando a lista está vazia:**

```
if (inicio == NULL) {
    inicio = novo;
    return inicio;
}
```

- Se a lista estiver vazia (ou seja, inicio for NULL), a nova antena se torna o primeiro elemento da lista. O ponteiro inicio é atualizado para apontar para novo.

Inserção no início da lista:

```
if ((novo->linha < inicio->linha) || (novo->linha == inicio->linha && novo->coluna < inicio->coluna)) {
    novo->next = inicio;
    inicio = novo;
    return inicio;
}
```

- A nova antena é inserida no início da lista se ela for "menor" do que a primeira antena da lista. A comparação é feita primeiro pela linha (novo->linha < inicio->linha), e se as linhas forem iguais, pela coluna (novo->coluna < inicio->coluna).
- Se a condição for verdadeira, o ponteiro next da nova antena (novo->next) aponta para o primeiro elemento da lista (inicio), e inicio é atualizado para apontar para novo.

Percorrer a lista para encontrar a posição correta:

```
antena* atual = inicio;
antena* anterior = atual;
```

```
while ((atual->linha < novo->linha || (atual->linha == novo->linha && atual->coluna < novo->coluna)) && atual->next != NULL) {
    anterior = atual;
    atual = atual->next;
}
```

- Se a nova antena não for inserida no início, a função começa a percorrer a lista. São usados para percorrer a lista dois ponteiros, atual e anterior, e localizar a posição correta onde a nova antena deve ser inserida.
- O laço while percorre a lista até encontrar uma antena cujo valor da linha e coluna seja maior do que o da nova antena ou até o final da lista (atual->next != NULL).

Inserção entre dois elementos existentes na lista:

```
if ((atual->linha == novo->linha && atual->coluna > novo->coluna)) {
    novo->next = atual;
    anterior->next = novo;
}
```

- Se encontramos um elemento com a mesma linha e uma coluna maior do que a da nova antena, a nova antena é inserida entre anterior e atual. O campo next de novo aponta para atual, e o campo next de anterior aponta para novo.

Inserção no final da lista:

```
else if (atual->linha < novo->linha && atual->coluna < novo->coluna) {
    atual->next = novo;
}
```

- Se a nova antena é maior do que todos os elementos da lista (tanto em linha quanto em coluna), ela é inserida no final da lista. O campo next de atual (o último elemento da lista) passa a apontar para a nova antena.

3.5-Função Remover Antena através de coordenadas

A função ***removerantena*** tem o objetivo de eliminar um nó específico da lista ligada de antenas, identificado pelas suas coordenadas (linha e coluna). Caso a antena se encontre na lista, o nó correspondente é removido e a memória alocada é libertada. Caso contrário, se a antena não for encontrada, a função retorna a lista inalterada. Abaixo, apresenta-se aspetos a destacar da função.

```
antena* removerantena(antena* h, int linha, int coluna)
```

Verificar se a antena a ser removida existe na lista:

```
if (procurarantena(h, linha, coluna) == NULL) return NULL;
```

- A função procurarantena é chamada para verificar se há uma antena na posição (linha, coluna).
- Se a antena não existir, a função retorna a lista original (h), pois não há nada para remover.

Remover a antena que está no início da lista:

```
if (h->linha == linha && h->coluna == coluna) {
    antena* aux = h;
    h = h->next;
    free(aux);
    return h;
}
```

- Se a antena a ser removida estiver no início da lista (h->linha == linha && h->coluna == coluna), um ponteiro auxiliar aux armazena o nó a ser removido.
- O ponteiro h é atualizado para apontar para o próximo nó (h = h->next).
- A memória ocupada pela antena removida é liberada (free(aux)).
- A função retorna o novo início da lista.

Remover a antena encontrada:

```
if (aux) {
    auxAnt->next = aux->next;
    free(aux);
}
```

- Se aux não for NULL, significa que encontramos a antena a ser removida.
- O ponteiro next do nó anterior (auxAnt) passa a apontar para o próximo nó após aux, removendo aux da lista.
- A memória da antena removida é liberada (free(aux)).

3.6-Função para Mostrar a Matriz das antenas

A função ***mostrarMatrizAntenas*** tem como objetivo apresentar uma representação visual, em formato matricial, das antenas armazenadas na lista ligada. Esta função itera sobre cada posição definida pela dimensão da matriz (determinada pelas variáveis globais MAX_LINHAS e MAX_COLUNAS) e, para cada par de coordenadas, utiliza a função ***procurarantena*** para verificar se existe uma antena presente.

Caso seja encontrada uma antena, é impressa a sua frequência; caso contrário, é exibido um ponto (.). O formato de impressão, com um espaço após cada carácter, garante o alinhamento correto entre as colunas da matriz.

```
void mostrarMatrizAntenas(antena* lista)
```

3.7-Função para libertar a memoria usada para Antenas

A função ***libertarListaAntenas*** é responsável por libertar toda a memória que foi alocada para a lista ligada de antenas, prevenindo assim possíveis fugas de memória e garantindo que o sistema se encontra numa situação limpa ao fim da sua execução. Para tal, a função percorre a lista de forma iterativa, nó a nó, e liberta a memória de cada elemento antes de prosseguir para o seguinte.

```
void libertarListaAntenas(antena* lista)
```

3.8-Função para criar o Efeito Nefasto

A função ***criarEfeitoNefasto*** tem como finalidade alocar dinamicamente memória para um novo nó da lista ligada de antenas. Esta operação é essencial para garantir a criação de uma estrutura válida que armazene a informação relativa à localização e à frequência de uma antena. Após a alocação, os campos da estrutura antena são devidamente inicializados, ficando o nó pronto a ser inserido na lista, respeitando a coerência e integridade dos dados mantidos pelo sistema.

```
criarEfeitoNefasto(int x, int y)
```

3.1.9-Função para Calcular o Efeito Nefasto

A função ***calcularEfeitoNefastoFinal*** é responsável por analisar a lista ligada de antenas e determinar os pontos onde se manifestam os efeitos nefastos. Este cálculo baseia-se na comparação de pares de antenas que operam na mesma frequência e na aplicação de um algoritmo que determina duas localizações simétricas (uma na direção de comparação e outra na direção oposta), considerando a diferença de posições entre as antenas. A função só adiciona um ponto nefasto à lista se este se encontrar dentro dos limites da matriz, definidos pelas variáveis globais MAX_LINHAS e MAX_COLUNAS.

O cálculo do efeito nefasto é realizado tendo em conta as coordenadas da primeira antena encontrada e a próxima, fazendo uma subtração da (x2-x1; y2-y1), e em seguida as coordenadas da primeira antena são subtraídas a esses valores e caso existam tais coordenadas o efeito nefasto é colocado, já a segunda é realizada uma soma e o efeito nefasto é colocado nas novas coordenadas.

```
nefasto* calcularEfeitoNefastoFinal(antena* lista)
```

Comparação entre Antenas:

```
if (atual->frequencia == comparar->frequencia) {
```

A condição if (atual->frequencia == comparar->frequencia) verifica se as antenas operam na mesma frequência.

Cálculo de Efeitos Nefastos:

Primeiro ponto nefasto (direção de comparar):

A diferença entre as linhas e as colunas das antenas é calculada.

```
int diff_linha = comparar->linha - atual->linha;

int diff_coluna = comparar->coluna - atual->coluna;
```

O primeiro ponto nefasto é calculado utilizando a diferença de posições.

```
int nefasto_linha = comparar->linha + diff_linha;

int nefasto_coluna = comparar->coluna + diff_coluna;
```

Segundo ponto nefasto (direção de atual):

O ponto nefasto na direção oposta à antena atual é calculado.

```
nefasto_linha = atual->linha - diff_linha;

nefasto_coluna = atual->coluna - diff_coluna;
```

Verificação dos Limites:

```
if (nefasto_linha >= 0 && nefasto_coluna >= 0 &&
    nefasto_linha < MAX_LINHAS && nefasto_coluna < MAX_COLUNAS) {
```

Antes de adicionar um ponto à lista de pontos nefastos, a função verifica se o ponto está dentro dos limites da matriz, utilizando as constantes MAX_LINHAS e MAX_COLUNAS.

Criação e Adição dos Pontos Nefastos:

```
nefasto* novo = criarEfeitoNefasto(nefasto_linha, nefasto_coluna);
novo->next = nefastos;
nefastos = novo;
```

Para o primeiro ponto, caso esteja dentro dos limites, é criado um novo ponto nefasto e é adicionado à lista nefastos.

O mesmo processo é repetido para o segundo ponto nefasto calculado.

```
if (nefasto_linha >= 0 && nefasto_coluna >= 0 &&
    nefasto_linha < MAX_LINHAS && nefasto_coluna < MAX_COLUNAS) {
    nefasto* novo2 = criarEfeitoNefasto(nefasto_linha, nefasto_coluna);
    novo2->next = nefastos;
    nefastos = novo2;
}
```

3.10-Função para Mostrar Matriz do Efeito Nefasto

A função ***mostrarMatrizNefastos*** é responsável por apresentar, de forma visual, a disposição das antenas e dos pontos onde se manifestam os efeitos nefastos. Para além de identificar e imprimir as antenas (representadas pela sua frequência), a função distingue as posições onde se verifica a ocorrência do efeito nefasto, mostrando um símbolo específico (#), e ainda identifica as posições vazias (representadas por um ponto '.').

A implementação recorre a dois ciclos aninhados que percorrem todas as posições de uma matriz, cujas dimensões são determinadas pelas variáveis globais MAX_LINHAS e MAX_COLUNAS. Para cada posição da matriz, é verificada primeiramente a presença de uma antena, através da função ***procurarantena***. Se uma antena existir, imprime-se a sua frequência. Caso contrário, procede-se à verificação na lista de pontos nefastos; se existir um efeito nefasto nessa posição, imprime-se o símbolo #, e se não, imprime-se um ponto.

```
void mostrarMatrizNefastos(antena* lista_antenas, nefasto* lista_nefastos)
```

3.1.11-Função para libertar a memoria usada para Efeito Nefasto

A função ***libertarListaNefasto*** é responsável por libertar toda a memória que foi alocada para a lista ligada de efeitos nefastos. Esta operação é fundamental para evitar fugas de memória, garantindo que todo o espaço reservado dinamicamente seja devidamente recuperado ao final da execução do programa. A função percorre a lista nó a nó, utilizando um laço iterativo, e liberta cada elemento antes de avançar para o seguinte.

```
void libertarListaNefasto(nefasto* lnefasto) {
```

Capítulo 4 – Função Main

A função main() implementa um menu interativo que permite ao utilizador realizar várias operações relacionadas com antenas e efeitos nefastos. O menu oferece as seguintes opções:

1. **Carregar antenas de um ficheiro**
2. **Mostrar a matriz de antenas**
3. **Calcular e mostrar efeitos nefastos**
4. **Inserir uma nova antena**
5. **Remover uma antena**
6. **Sair do programa**

Declaração de Variáveis:

- **opcao**: Variável usada para armazenar a opção escolhida pelo utilizador.
- **lista_antenas**: Ponteiro para a lista de antenas.
- **lista_nefastos**: Ponteiro para a lista de pontos nefastos.
- **nome_ficheiro**: String para armazenar o nome do ficheiro a ser carregado.

Capítulo 5 – Resultados e exemplos de execução.

Menu

```
Menu:
1. Carregar antenas do ficheiro
2. Mostrar matriz de antenas
3. Calcular e mostrar efeitos nefastos
4. Inserir nova antena
5. Remover antena
0. Sair
Escolha uma opcao:
```

Figura 3 Menu do programa

Carregar antenas

```
Menu:
1. Carregar antenas do ficheiro
2. Mostrar matriz de antenas
3. Calcular e mostrar efeitos nefastos
4. Inserir nova antena
5. Remover antena
0. Sair
Escolha uma opcao: 1
Digite o nome do ficheiro: antenas.txt
Antenas carregadas com sucesso!
```

Figura 4 Função Carregar Antenas

Mostrar matriz de antenas

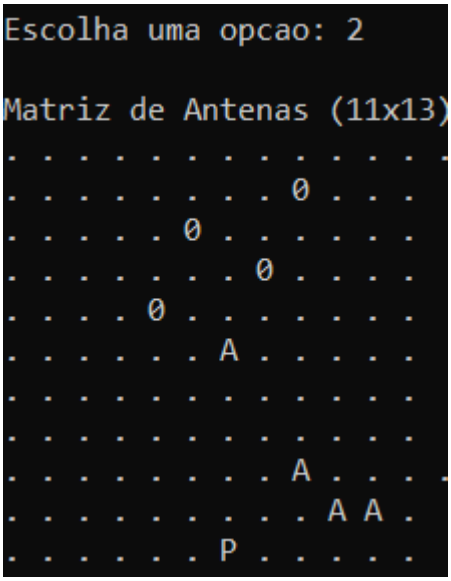


Figura 5 Função mostrar matriz de antenas

Calcular e mostra efeitos nefastos

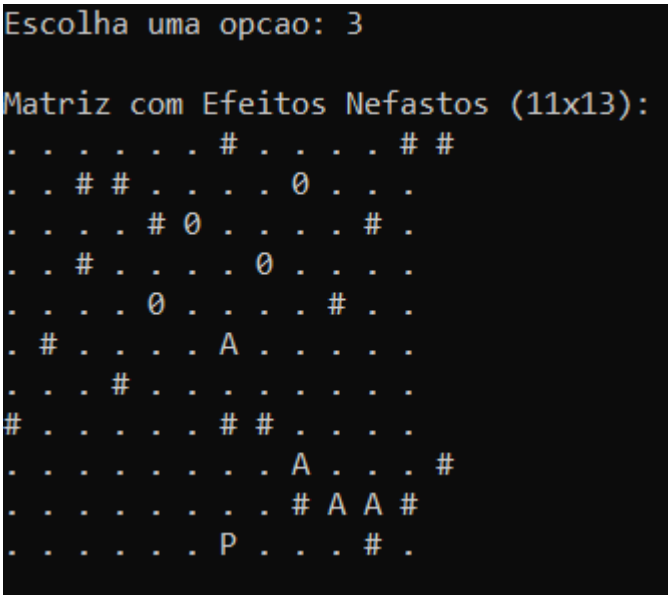


Figura 6 Função mostrar efeitos nefastos

Inserir nova antena

```
Escolha uma opcao: 4
Digite a linha: 1
Digite a coluna: 2
Digite a frequencia: B
Antena inserida com sucesso!

Menu:
1. Carregar antenas do ficheiro
2. Mostrar matriz de antenas
3. Calcular e mostrar efeitos nefastos
4. Inserir nova antena
5. Remover antena
0. Sair
Escolha uma opcao: 2

Matriz de Antenas (11x13):
. . . . . .
. . B . . . . . 0 . . .
. . . . . 0 . . . . . .
. . . . . 0 . . . .
. . . . 0 . . . . . .
. . . . . A . . . . .
. . . . . .
. . . . . .
. . . . . A . . . .
. . . . . A A .
. . . . . P . . . . .
```

Figura 7 Função inserir nova antena

Remover antena

```
Escolha uma opcao: 5
Digite a linha: 1
Digite a coluna: 2
Antena removida (se existia).

Menu:
1. Carregar antenas do ficheiro
2. Mostrar matriz de antenas
3. Calcular e mostrar efeitos nefastos
4. Inserir nova antena
5. Remover antena
0. Sair
Escolha uma opcao: 2

Matriz de Antenas (11x13):
. . . . . .
. . . . . 0 . . .
. . . . . 0 . . . . . .
. . . . . 0 . . . .
. . . . 0 . . . . . .
. . . . . A . . . . .
. . . . . .
. . . . . .
. . . . . A . . . .
. . . . . A A .
. . . . . P . . . . .
```

Figura 8 Função remover antena

Conclusão

A implementação da Fase 1 deste projeto proporcionou uma sólida consolidação dos conhecimentos teóricos sobre estruturas de dados dinâmicas, com ênfase nas listas ligadas, e permitiu aplicar esses conceitos de forma prática na linguagem C. Ao resolver o problema proposto, foi possível desenvolver uma compreensão mais aprofundada dos desafios e das soluções associadas à manipulação de dados em memória dinâmica, além de aprimorar as habilidades de programação e resolução de problemas. Esse processo não só reforçou os fundamentos das estruturas de dados, mas também contribuiu para uma maior confiança na aplicação dessas técnicas em contextos reais.

Fase 2

Objetivos da Fase 2

Na segunda fase do projeto, pretende-se aplicar **conceitos de teoria de grafos** em conjunto com **programação em C**, com o objetivo de representar e analisar a rede de antenas de uma cidade de forma mais avançada.

O foco principal é a definição e implementação de um **tipo de dados GR**, capaz de representar um grafo com um número arbitrário de vértices, onde cada vértice corresponde a uma antena e cada aresta liga antenas com **frequências de ressonância iguais**.

Para tal, o enunciado propõe a realização das seguintes tarefas:

- **Definição da estrutura de dados GR**, garantindo a flexibilidade necessária para representar um conjunto dinâmico de antenas e respetivas ligações, condicionadas pela igualdade de frequência de ressonância.
- **Implementação de algoritmos de exploração e análise sobre o grafo GR**, nomeadamente:
    - Algoritmo de **procura em profundidade (DFS)**, com o objetivo de listar todas as antenas acessíveis a partir de um determinado vértice;
    - Algoritmo de **procura em largura (BFS)**, com a mesma finalidade, percorrendo o grafo por níveis;
    - Identificação de **todos os caminhos possíveis** entre duas antenas selecionadas, evidenciando as sequências de ligações existentes;
    - **Listagem de intersecções** entre antenas com frequências distintas (A e B), indicando as coordenadas das posições onde coexistem.

Capítulo 1 - Análise e Especificação

Este capítulo apresenta a **especificação funcional do sistema**, incluindo os **requisitos funcionais identificados**, as **estruturas de dados** concebidas para representar a informação relevante, bem como uma descrição das **variáveis globais utilizadas**. Adicionalmente, é incluído um **diagrama simplificado da estrutura geral do programa**, com o intuito de proporcionar uma visão clara da organização e interligação dos seus principais componentes.

1.1 – Requisitos do Sistema

Os principais requisitos funcionais definidos para esta fase do projeto foram:

- Ler uma matriz de caracteres a partir de um ficheiro de texto e converter a informação em listas ligadas.

- Procura em profundidade a partir de uma determinada antena, listando na consola as coordenadas das antenas alcançadas.

- Procura em largura a partir de uma determinada antena, listando na consola as coordenadas das antenas alcançadas.

- Identificar todos os caminhos possíveis entre duas determinadas antenas, listando na consola as sequências de arestas interligando as antenas.

- Dadas duas frequências de ressonância distintas A e B, listar na consola todas as intersecções de pares de antenas com ressonância A e B, indicando as respectivas coordenadas.

1.2- Estruturas de Dados

Foram definidas três estruturas principais, ambas implementadas através de listas ligadas simples:

- **Vertice:** Representa um vértice (antena) com os atributos linha, coluna, frequência(carácter), um parâmetro para saber se foi visitado, um ponteiro para o próximo elemento, e um ponteiro para adjacência.

```
1. typedef struct Vertice {
2.       char frequencia;
3.       int linha;
4.       int coluna;
5.       bool visitado;
6.       struct Vertice* next;
7.       struct ADJ* adnext;
8.
9. }Vertice;
```

- **ADJ:** Representa as adjacências de um vértice, tendo como parâmetros a frequência, linha, coluna, e um apontador para a próxima adjacência.

```
1. typedef struct ADJ {
2.       char freque;
3.       int l;
4.       int c;
5.
6.       struct ADJ* prox;
7.
8. }ADJ;
```

- **Graph:** Representa a estrutura do grafo tendo como parâmetros um apontador para a struct Vertice e um int representando o Número de Vértices.

```
1. typedef struct Graph {
2. Vertice* InicioGraph;
3. int NumeroVertices;
4. }Graph;
```

1.3- Variáveis Globais

Foram utilizadas as seguintes variáveis globais:

```
int MAX_COLUNAS = 0;
int MAX_LINHAS = 0;
```

Estas variáveis armazenam as dimensões máximas da matriz de antenas, e são utilizadas para validar coordenadas e percorrer corretamente o espaço representado.
As variáveis globais são declaradas fora de qualquer função e podem ser acedidas por qualquer parte do programa. Ao contrário das variáveis locais, que existem apenas dentro da função onde foram definidas, as variáveis globais mantêm o seu valor ao longo de toda a execução do programa.

Foi também definido os seguintes limites:

```
#define MAX_CAMINHO 10000
#define MAX 1000
```

Estas constantes definem o tamanho máximo permitido para o caminho percorrido, utilizadas em algumas funções do programa.
São usadas para limitar a quantidade de dados que podem ser registados ou seguidos, garantindo que o programa não ultrapasse os limites de memória ou entre em loops infinitos.

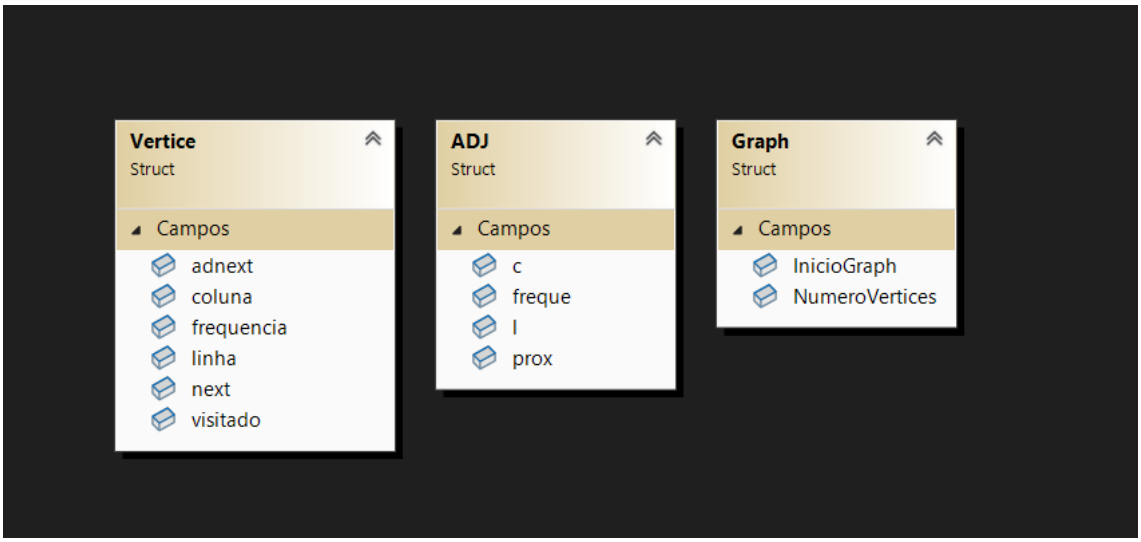
1.4- Assinatura das funções

As assinaturas das funções usadas encontram-se no ficheiro Assinaturas.h

```
Vertice* CriarVertice(char frequencia, int linha, int coluna);
Vertice* CarregarAntenasDoFicheiro(const char* nome_ficheiro);
bool ExisteVertice(Vertice* inicio, int l, int c);
Vertice* ProcurarVertice(Vertice* h, int l, int c);
Vertice* InsereOrdenado(Vertice* inicio, Vertice* novo, bool* res);
void ShowGraph(Vertice* graph);
Vertice* RemoverVertice(Vertice* h, int linha, int coluna, bool* res);
Vertice* LibertarVertice(Vertice* lista);


ADJ* CriaAdj(char frequencia, int linha, int coluna);
ADJ* InsereAdj(ADJ* ListaAdja, int l, int c, char frequencia);
bool ConstruirAdjacencias(Vertice* lista);
bool ConstruirAdjacenciasTodas(Vertice* lista);
void ApagaADJ(ADJ* AdjacenciaApag);
ADJ* ElimiminaAdj(ADJ* ListAdj, int l, int c, bool* res);


Graph* CriaGraph();
Graph* InsereVerticeGraph(Graph* h, Vertice* novo, bool* res);
Graph* InsereAdjGraph(Graph* h, int lOrigem, int cOrigem, int lDestino, int cDestino, bool* res);
Graph* EliminaAdjGraph(Graph* g, int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, bool* res);
Vertice* OndeEstaVerticeGraph(Graph* g, int linha, int coluna);
bool ExisteVerticeGraph(Graph* g, int linha, int coluna);
Graph* EliminaVerticeGraph(Graph* g, int linha, int coluna, bool* res);
void ShowGaphLA(Graph* g);

bool InserirQueue(int fila[], int* fim, int valor);
int RemoverQueue(int fila[], int* inicio);
bool VaziaQueue(int fim, int inicio);
void ApanhaVertice(Vertice* v);
bool BreadthFirstTraversalLA(Graph* g, int linhan, int colunan);
Vertice* ResetVertices(Vertice* inicio);

bool isStackEmpty(int top);
bool push(int stack[], int* top, int valor);
int pop(int stack[], int* top);
Vertice* GetAdjNaoVisitado(Vertice* lstVertices, int linha, int coluna);
bool DepthFirstTraversal(Graph* g, int linOrigem, int colOrigem);
void ImprimirCaminho(int* linhas, int* colunas, int tamanho);
bool  MostrarTodosCaminhosDFS(Graph* g, Vertice* atual, Vertice* destino, int* caminhoLinhas, int* caminhoColunas,
int profundidade);
bool ProcurarTodosOsCaminhos(Graph* g, int linOrigem, int colOrigem, int linDestino, int colDestino);
void CalcularInterferenciasAB(Graph* grafo, char freqA, char freqB);
```

1.5-Classe Diagram

O diagrama de classes representa a estrutura das principais entidades do projeto:



Capítulo 2 – Funções do Programa

Neste capítulo é apresentada a implementação das funcionalidades principais do projeto. Cada função foi desenvolvida com base nas estruturas de dados previamente especificadas e respeita os requisitos definidos na fase de análise. As operações foram estruturadas de forma modular, permitindo a separação lógica entre a criação, manipulação e visualização dos dados. Para além das funções de criação e eliminação, são também descritos os algoritmos para a procura em largura e profundidade e a ressonância entre duas frequências distintas. Abaixo, destacam-se os pontos mais importantes de cada função implementada:

2.1-Função criar Vértice

A função ***CriarVerice*** tem como finalidade alocar dinamicamente memória para um novo nó da lista ligada de vértices. Esta operação é essencial para garantir a criação de uma estrutura válida que armazene a informação relativa à localização e à frequência de um vértice. Após a alocação, os campos da estrutura vértice são devidamente inicializados, ficando o nó pronto a ser inserido na lista, respeitando a coerência e integridade dos dados mantidos pelo sistema.

- **Vertice* CriarVertice(char frequencia, int linha, int coluna);**

2.2 – Função Carregar Antenas Do Ficheiro

Esta função é a mesma utilizada na fase 1.

2.3 - Verificar Existência de Vértice

A função ***ExisteVertice*** tem como objetivo confirmar se um vértice com determinadas coordenadas já foi criado. Isto é especialmente útil na verificação de redundâncias e integridade da estrutura de dados.

Durante a construção do grafo, esta função é utilizada para evitar duplicações de vértices e garantir que cada antena do mapa seja representada uma única vez.

```
bool ExisteVertice(Vertice* inicio, int l, int c)
```

2.4 - Procurar Vértice

A função ProcurarVertice tem como função localizar e retornar um ponteiro para o vértice com coordenadas específicas dentro da lista ligada de vértices. Caso não seja encontrado, retorna NULL.

Este procedimento é frequentemente utilizado durante a criação de adjacências e na execução de algoritmos de travessia (como BFS e DFS), onde é necessário acessar rapidamente os dados de um vértice com base na sua posição no mapa.

A eficiência da função está diretamente relacionada à simplicidade da estrutura de lista ligada utilizada.

```
Vertice* ProcurarVertice(Vertice* h, int l, int c)
```

2.5 - Inserir Vértice de Forma Ordenada

Esta função tem por objetivo inserir um novo vértice na lista de vértices mantendo a ordem crescente, primeiro por linha e depois por coluna. Além disso, previne a inserção de vértices duplicados, usando a função **ExisteVertice** como mecanismo de verificação.

A ordenação é relevante para facilitar futuras buscas e para garantir uma organização lógica da estrutura de dados. Ao manter os vértices ordenados, as travessias do grafo e as representações visuais tornam-se mais previsíveis e compreensíveis.

```
Vertice* InsereOrdenado(Vertice* inicio, Vertice* novo, bool* res)
```

2.6 - Mostrar Grafo

A função ShowGraph percorre a lista de vértices existente e imprime na consola os detalhes de cada antena, como a sua posição (linha e coluna) e a frequência correspondente. Para cada vértice, a função também percorre a sua lista de adjacências, apresentando as ligações com outras antenas.

Este procedimento é útil para depuração, validação da estrutura de dados e análise visual da topologia da rede. Ao visualizar as conexões diretamente, o programador e o utilizador podem confirmar se as adjacências foram corretamente estabelecidas e se a estrutura representa fielmente os dados do ficheiro.

```
void ShowGraph(Vertice* graph);
```

2.7 - Remover Vértice

A função **RemoverVertice** é responsável por eliminar um vértice específico da lista ligada de vértices. Esta operação não se limita a remover apenas o nó principal: também garante a remoção de todas as adjacências que apontam para esse vértice a partir de outros vértices da lista.

Trata-se de uma função crítica para a manutenção da integridade do grafo, evitando ligações inválidas ou memória não libertada após a remoção de um nó. O uso interno de outras funções auxiliares como **RemoverAdjacenciasParaVertice** e **ElimiminaAllAdj** torna o processo modular, limpo e mais fácil de manter.

```
Vertice* RemoverVertice(Vertice* h, int linha, int coluna, bool* res);
```

2.8 - Libertar Vértices

Esta função percorre toda a lista ligada de vértices e liberta a memória associada a cada nó. Para cada vértice, ela também chama **ElimiminaAllAdj** para garantir que todas as listas de adjacência sejam devidamente eliminadas antes de libertar o vértice em si.

A função é usada no final da execução do programa (em main) e sempre que é necessário libertar a estrutura de forma segura e completa. Essa gestão eficiente de memória evita fugas e assegura a boa prática de programação, especialmente em contextos com alocação dinâmica.

```
Vertice* LibertarVertice(Vertice* lista);
```

2.9 – Criar Adjacência

A função **CriaAdj** tem por finalidade criar e inicializar um novo nó de adjacência, que representa uma ligação entre dois vértices. Este nó contém a posição (linha e coluna) do vértice destino e a sua frequência.

O uso de adjacências torna possível representar as conexões entre antenas de forma eficiente, permitindo construir listas de vizinhança para cada vértice, conforme os princípios de grafos representados por listas de adjacência.

```
ADJ* CriaAdj(char frequencia, int linha, int coluna);
```

2.10 – Inserir Adjacência

A função **InsereAdj** adiciona uma nova adjacência à lista de adjacências de um vértice, desde que ainda não exista uma adjacência para o mesmo destino (linha e coluna). Esta verificação é feita antes da inserção, garantindo que não haja duplicações nas ligações.

A lógica desta função permite manter a lista de adjacência consistente, refletindo corretamente as conexões entre vértices. A inserção ocorre no final da lista, o que preserva a ordem de criação e evita reordenações desnecessárias.

```
ADJ* InsereAdj(ADJ* ListaAdja, int l, int c, char frequencia);
```

2.11 – Construir Adjacências

A função **ConstruirAdjacencias** percorre a lista de vértices e, para cada vértice, procura outros vértices com a mesma frequência. Quando encontra, cria uma adjacência entre eles, ligando apenas antenas com frequências idênticas.

Este comportamento permite modelar corretamente a topologia da rede com base nas regras do enunciado, que estipula que apenas antenas com a **mesma frequência de ressonância** devem estar conectadas. Assim, esta função cria uma representação fiel das conexões válidas entre antenas operando na mesma frequência.

```
bool ConstruirAdjacencias(Vertice* lista);
```

2.12 – Construir Adjacências para Todos os Vértices

Diferente da anterior, **ConstruirAdjacenciasTodas** liga todos os vértices entre si, independentemente da frequência. A função é utilizada na criação do **grafo completo**, que representa todas as antenas, ignorando diferenças de frequência.

```
bool ConstruirAdjacenciasTodas(Vertice* lista);
```

2.13 – Apagar Adjacência

A função **ApagaADJ** é simples, mas essencial: recebe um ponteiro para uma adjacência e liberta a memória associada a ela. É utilizada como parte integrante de outras funções, como **ElimiminaAdj** ou **ElimiminaAllAdj**, garantindo que os nós de adjacência sejam corretamente destruídos.

O seu uso evita **fugas de memória**, que poderiam comprometer a estabilidade do programa em execuções prolongadas ou com grandes volumes de dados.

```
void ApagaADJ(ADJ* AdjacenciaApag);
```

2.14 – Eliminar Adjacência

Esta função percorre a lista de adjacências de um vértice e remove a adjacência que corresponde às coordenadas fornecidas. Após a remoção, ajusta os ponteiros da lista para manter a coerência da estrutura ligada.

É uma função crítica para operações de manutenção do grafo, como a remoção de vértices (onde todas as adjacências relacionadas devem ser eliminadas). Ao incluir um parâmetro booleano res, a função informa ao chamador se a operação foi bem-sucedida.

```
ADJ* ElimiminaAdj(ADJ* ListAdj, int l, int c, bool* res);
```

2.15 – Eliminar Todas as Adjacências

A função **ElimiminaAllAdj** percorre toda a lista de adjacências de um vértice e elimina todos os seus elementos, libertando a memória ocupada. Esta função é usada, por exemplo, durante a eliminação de um vértice, para garantir que as ligações desse nó sejam completamente removidas.

A remoção completa das adjacências é essencial para manter a integridade da estrutura de dados e para evitar **ligações órfãs** que apontariam para vértices inexistentes.

```
ADJ* ElimiminaAllAdj(ADJ* listAdj, bool* res);
```

2.16 – Remover Adjacências para Vértice Específico

A função **RemoverAdjacenciasParaVertice** percorre todos os vértices da lista e remove, de cada um, qualquer adjacência que aponte para um vértice específico, identificado pelas suas coordenadas (linha e coluna).

Esta função é fundamental para garantir que, ao remover um vértice, todas as ligações (arestas) que outros vértices mantinham com ele também sejam eliminadas. Caso contrário, poderiam surgir referências inválidas na estrutura, o que comprometeria a coerência e segurança do grafo.

```
Vertice* RemoverAdjacenciasParaVertice(Vertice* lista, int linha, int coluna, bool* res);
```

2.17 – Criar Grafo

A função **CriaGraph** inicializa a estrutura do grafo, alocando memória para um novo objeto do tipo Graph. Inicializa os campos com valores padrão: o ponteiro para o início da lista de vértices (InicioGraph) é definido como NULL e o contador de vértices (NumeroVertices) é posto a zero.

```
Graph* CriaGraph();
```

2.18 – Inserir Vértice no Grafo

A função **InsereVerticeGraph** insere um novo vértice numa estrutura de grafo previamente criada. Internamente, a função usa **InsereOrdenado** para manter a lista ligada de vértices ordenada, garantindo organização lógica do grafo.

```
Graph* InsereVerticeGraph(Graph* h, Vertice* novo, bool* res);
```

2.19 – Inserir Adjacência no Grafo

Esta função adiciona uma ligação entre dois vértices num grafo, ou seja, insere uma adjacência do vértice de origem para o vértice de destino.

```
Graph* InsereAdjGraph(Graph* h, int lOrigem, int cOrigem, int lDestino, int cDestino, bool* res);
```

2.20 – Eliminar Adjacência no Grafo

A função **EliminaAdjGraph** permite eliminar uma ligação específica entre dois vértices no grafo. Essa operação é comum em cenários onde se pretende simular a quebra de uma ligação (por exemplo, falha de comunicação entre duas antenas).

```
Graph* EliminaAdjGraph(Graph* g, int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, bool* res);
```

2.21 – Encontrar Vértice no Grafo

A função **OndeEstaVerticeGraph** procura um vértice com coordenadas específicas dentro de um grafo. Internamente, esta função é apenas um invólucro da função **ProcurarVertice**, aplicada à lista de vértices contida na estrutura Graph.

```
Vertice* OndeEstaVerticeGraph(Graph* g, int linha, int coluna);
```

2.22 – Verificar Existência de Vértice no Grafo

A função **ExisteVerticeGraph** verifica se um vértice com determinadas coordenadas está presente na estrutura do grafo. Assim como na função anterior, ela faz uso de **ExisteVertice** para percorrer a lista de vértices do grafo.

Este tipo de verificação é essencial para assegurar que certas operações (como inserção de adjacências ou início de uma travessia) sejam realizadas apenas em vértices válidos. Isso evita erros de execução ou comportamentos inesperados.

```
bool ExisteVerticeGraph(Graph* g, int linha, int coluna);
```

2.23 – Eliminar Vértice no Grafo

Esta função elimina um vértice da estrutura Graph com base nas suas coordenadas. Para isso, invoca internamente a função **RemoverVertice**, que trata da eliminação do vértice e de todas as suas adjacências (tanto as que ele contém como as que o apontam).

Após a remoção bem-sucedida, o contador de vértices do grafo (NumeroVertices) é decrementado, refletindo a alteração na estrutura.

```
Graph* EliminaVerticeGraph(Graph* g, int linha, int coluna, bool* res);
```

2.24 – Mostrar Grafo com Lista de Adjacências

A função ShowGaphLA imprime o grafo, listando os vértices e respetivas adjacências. Na prática, é uma chamada direta para ShowGraph, aplicada ao campo InicioGraph da estrutura Graph.

```
void ShowGaphLA(Graph* g);
```

2.25 – Funções auxiliares QUEUE

2.25.1 - Inserir na QUEUE

Insere um elemento no final da queue.

```
bool InserirQueue(int fila[], int* fim, int valor)
```

2.25.2 – Remover da QUEUE

Remove e retorna o elemento do início da queue e devolve-o.

```
int RemoverQueue(int fila[], int* inicio)
```

2.25.3 – Verificar se a QUEUE está vazia

Verifica se a queue está vazia.

```
bool VaziaQueue(int fim, int inicio)
```

2.26 – Mostra a antena alcançada

A função **ApanhaVertice** é uma função auxiliar que imprime, no terminal, a posição de um vértice (antena) alcançado durante uma travessia. Esta função é usada dentro dos algoritmos de busca em largura (BFS) e profundidade (DFS) para informar o utilizador sobre quais antenas foram visitadas ao longo do percurso.

```
void ApanhaVertice(Vertice* v)
```

2.27 – Função de Procura em Largura

A função **BreadthFirstTraversalLA** implementa a travessia em largura (BFS – Breadth First Search), começando por uma antena especificada e explorando todos os seus vizinhos de forma **nivelada** (por camadas).

```
bool BreadthFirstTraversalLA(Graph* g, int linhan, int colunan)
```

2.28 – Função para resetar o campo visitado

A função **ResetVertices** percorre a lista de vértices e define o campo visitado de todos os vértices como false. Isso é necessário sempre que se pretende executar uma nova travessia (BFS ou DFS), para garantir que todos os nós estejam disponíveis para serem novamente percorridos.

```
Vertice* ResetVertices(Vertice* inicio)
```

2.29 – Funções auxiliares para Stack

2.29.1 – Função para ver se a Stack esta vazia

Verifica se a Stack está vazia, usada em buscas do tipo profundidade (DFS).

```
bool isStackEmpty(int top)
```

2.29.2 – Função para inserir na Stack

Empilha um elemento na pilha.

```
bool push(int stack[], int* top, int valor)
```

2.29.3 – Função para remover da Stack

Desempilha e retorna o elemento do topo da pilha.

```
int pop(int stack[], int* top)
```

2.29.4 – Função para obter vértice não visitado

Retorna o vértice adjacente ainda não visitado para continuar a busca em profundidade.

```
Vertice* GetAdjNaoVisitado(Vertice* lstVertices, int linha, int coluna)
```

2.30 –Percorrer o grafo em Profundidade

A função **DepthFirstTraversal** executa a travessia em profundidade (DFS – Depth First Search), começando num vértice de origem e explorando o mais fundo possível antes de retroceder.
A implementação é **iterativa**, utilizando uma pilha (stack) para armazenar os vértices à medida que são visitados. Isso evita o uso de recursão e permite um maior controlo sobre os dados.

```
bool DepthFirstTraversal(Graph* g, int linOrigem, int colOrigem)
```

2.31 – Imprimir caminho

A função ImprimirCaminho recebe dois arrays de inteiros (linhas e colunas) e um tamanho. Estes arrays representam, respetivamente, as coordenadas dos vértices que formam um caminho identificado entre duas antenas. A função percorre os arrays e imprime as posições formatadas, separando-as com o símbolo "→".

Esta função é utilizada após a identificação de um caminho válido, permitindo apresentar ao utilizador a sequência completa de antenas envolvidas.

```
void ImprimirCaminho(int* linhas, int* colunas, int tamanho)
```

2.32 – Mostrar Todos os Caminhos entre dois vértices

A função **MostrarTodosCaminhosDFS** utiliza uma abordagem recursiva de DFS para descobrir **todos os caminhos possíveis** entre dois vértices de um grafo. Durante a execução, vai construindo o caminho atual em arrays auxiliares e, quando atinge o destino, imprime o caminho completo com a função **ImprimirCaminho**.

```
bool  MostrarTodosCaminhosDFS(Graph* g, Vertice* atual, Vertice* destino, int* caminhoLinhas, int* caminhoColunas, int profundidade) {
```

2.33 - Procurar Todos os Caminhos

Esta função serve como **ponto de entrada** para a funcionalidade de descoberta de caminhos. Recebe as coordenadas de origem e destino, procura os vértices correspondentes, inicializa os arrays de caminho e invoca **MostrarTodosCaminhosDFS**.

```
bool ProcurarTodosOsCaminhos(Graph* g, int linOrigem, int colOrigem, int linDestino, int colDestino)
```

2.34 - Calcular Interferências Entre Frequências A e B

A função **CalcularInterferenciasAB** analisa a possibilidade de **interferência entre antenas de duas frequências distintas**. Percorre os vértices do grafo e compara todos os pares possíveis de vértices cujas frequências correspondem a freqA e freqB.

Uma interferência é identificada se as antenas estiverem **alinhadas horizontalmente, verticalmente ou diagonalmente**, ou seja, se estiverem sobre a mesma linha, coluna ou diagonal (mesmo valor absoluto de diferença entre linha e coluna).

```
void CalcularInterferenciasAB(Graph* grafo, char freqA, char freqB)
```

Capítulo 3 – Função MAIN

A função main() do programa tem como objetivo gerir a interação com o utilizador e orquestrar o funcionamento de um sistema de antenas representado por grafos. A estrutura do programa segue uma abordagem modular, permitindo criar, manipular e visualizar grafos que representam antenas com diferentes frequências de emissão.

1. Carregamento do Ficheiro

Logo no início, o programa solicita ao utilizador o nome de um ficheiro de texto que contém a informação das antenas:

```
printf("Nome do ficheiro a carregar (ex: mapa.txt): ");
scanf("%s", nomeFicheiro);
```

Esse ficheiro é lido pela função **CarregarAntenasDoFicheiro**, que devolve uma **lista ligada de vértices**, onde cada vértice representa uma antena com os seguintes atributos:

Frequência (ex: 'A', 'B', etc.),

Posição (linha e coluna).

Caso a leitura falhe, o programa termina com uma mensagem de erro.

2. Criação das Estruturas de Dados

São criadas duas principais estruturas para armazenar os grafos:

grafoCompleto: contém todas as antenas do ficheiro, independentemente da frequência;

grafosPorFrequencia[256]: vetor que pode conter até 256 grafos distintos, um para cada frequência possível (caracter ASCII).

3. Menu Interativo

O programa entra num ciclo do-while com um menu que permite ao utilizador escolher entre várias opções. Estas opções incluem:

Opção 1 - Criar grafo com todas as antenas

Cria o grafo completo, inserindo cada antena na estrutura e estabelecendo as adjacências entre elas com base na sua proximidade.

Opção 2 - Criar grafos por frequência

Separa as antenas por frequência e cria um grafo para cada uma. Cada grafo é construído apenas com antenas que operam na mesma frequência.

Opção 3 - Visualizar o grafo completo

Exibe na consola a estrutura do grafo completo, mostrando cada vértice e os seus vizinhos.

Opção 4 - Visualizar grafo por frequência

Permite visualizar apenas o grafo de uma frequência específica, indicada pelo utilizador.

Opções 5 e 6 - Busca em largura e profundidade

Realiza uma travessia no grafo por frequência, a partir de uma antena origem escolhida pelo utilizador. A travessia pode ser feita:

Em **largura** (BFS – Breadth First Search),

Em **profundidade** (DFS – Depth First Search).

Estas técnicas ajudam a identificar quais antenas estão conectadas a partir de uma origem.

Opção 7 - Procurar todos os caminhos entre duas antenas

Procura todos os caminhos possíveis entre duas antenas no grafo completo, com base nas suas posições.

Opção 8 - Calcular interferência entre antenas

Verifica todas as combinações de pares de antenas de duas frequências diferentes, permitindo analisar possíveis interferências.

4. Libertação de Memória

Ao final da execução, toda a memória alocada dinamicamente (lista de vértices, grafos) é libertada para evitar fugas de memória, utilizando as funções adequadas como **LibertarVertice**() e **free**().

Capítulo 4 - Resultados e exemplos de execução.

4.1 – Carregar Ficheiro

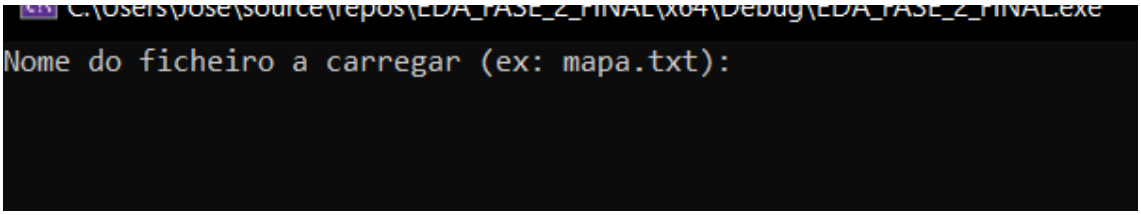


Figura 9 Carregar Ficheiro FASE 2

4.2 - Menu

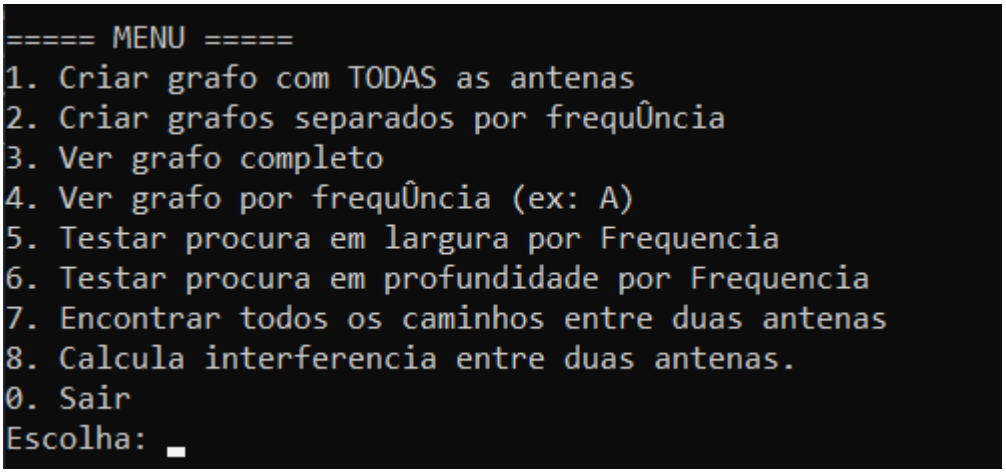


Figura 10 Menu FASE 2

4.3 - Criar Grafo de Todas as frequências

```
Escolha: 1
Criar grafo com TODAS as antenas...
Grafo completo criado com sucesso!
```

Figura 11 Criar Grafo Total FASE 2

4.4 - Criar Grafo por frequência

```
Escolha: 2
Criar grafos por frequencia...
Grafos por frequencia criados com sucesso!
```

Figura 12 Criar Grafo Frequências FASE 2

4.5 - Mostrar Grafo Completo

```
Escolha: 3

--- Grafo Completo ---
Antena em (0, 7) - Frequencia: C
                Adjacente em (2, 2) - Frequencia: B
                Adjacente em (2, 6) - Frequencia: B
                Adjacente em (5, 1) - Frequencia: B
                Adjacente em (6, 3) - Frequencia: A
                Adjacente em (6, 6) - Frequencia: A
Antena em (2, 2) - Frequencia: B
                Adjacente em (0, 7) - Frequencia: C
                Adjacente em (2, 6) - Frequencia: B
                Adjacente em (5, 1) - Frequencia: B
                Adjacente em (6, 3) - Frequencia: A
                Adjacente em (6, 6) - Frequencia: A
Antena em (2, 6) - Frequencia: B
                Adjacente em (0, 7) - Frequencia: C
                Adjacente em (2, 2) - Frequencia: B
                Adjacente em (5, 1) - Frequencia: B
                Adjacente em (6, 3) - Frequencia: A
                Adjacente em (6, 6) - Frequencia: A
Antena em (5, 1) - Frequencia: B
                Adjacente em (0, 7) - Frequencia: C
                Adjacente em (2, 2) - Frequencia: B
                Adjacente em (2, 6) - Frequencia: B
                Adjacente em (6, 3) - Frequencia: A
                Adjacente em (6, 6) - Frequencia: A
Antena em (6, 3) - Frequencia: A
                Adjacente em (0, 7) - Frequencia: C
                Adjacente em (2, 2) - Frequencia: B
                Adjacente em (2, 6) - Frequencia: B
                Adjacente em (5, 1) - Frequencia: B
                Adjacente em (6, 6) - Frequencia: A
Antena em (6, 6) - Frequencia: A
                Adjacente em (0, 7) - Frequencia: C
                Adjacente em (2, 2) - Frequencia: B
                Adjacente em (2, 6) - Frequencia: B
                Adjacente em (5, 1) - Frequencia: B
                Adjacente em (6, 3) - Frequencia: A
```

Figura 13 Mostrar Grafo Completo FASE 2

4.6 - Mostrar Grafo de uma frequência especifica

```
Escolha: 4
Indique a frequÛncia (ex: A): B

--- Grafo da frequencia 'B' ---
Antena em (2, 2) - Frequencia: B
                Adjacente em (2, 6) - Frequencia: B
                Adjacente em (5, 1) - Frequencia: B
Antena em (2, 6) - Frequencia: B
                Adjacente em (2, 2) - Frequencia: B
                Adjacente em (5, 1) - Frequencia: B
Antena em (5, 1) - Frequencia: B
                Adjacente em (2, 2) - Frequencia: B
                Adjacente em (2, 6) - Frequencia: B
```

Figura 14 Mostrar Grafo por Frequência FASE 2

4.7 – Procura em Largura

```
Escolha: 5
Indique a frequencia do grafo para busca em largura (ex: A): B
Linha da antena de origem: 2
Coluna da antena de origem: 2
Procura em Largura na frequencia 'B' a partir de (2, 2):
Antena alcanþada: (2, 2)
Antena alcanþada: (2, 6)
Antena alcanþada: (5, 1)
```

Figura 15 Procurar por largura na mesma Frequência FASE 2

4.8 – Procura em Profundidade

```
Escolha: 6
Indique a frequÛncia do grafo para busca em profundidade (ex: A): B
Linha da antena de origem: 2
Coluna da antena de origem: 2

Procura em Profundidade na frequÛncia 'B' a partir de (2, 2):
Antena alcanþada: (2, 2)
Antena alcanþada: (2, 6)
Antena alcanþada: (5, 1)
```

Figura 16 Procurar por profundidade na mesma frequência FASE 2

4.9 – Procura todos os caminhos possíveis entre duas frequências

```
Escolha: 7
Linha da antena de origem: 0
Coluna da antena de origem: 7
Linha da antena de destino: 6
Coluna da antena de destino: 3

Procurando todos os caminhos entre (0, 7) e (6, 3):
Caminho encontrado:
(0, 7) -> (2, 2) -> (2, 6) -> (5, 1) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (2, 6) -> (5, 1) -> (6, 6) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (2, 6) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (2, 6) -> (6, 6) -> (5, 1) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (2, 6) -> (6, 6) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (5, 1) -> (2, 6) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (5, 1) -> (2, 6) -> (6, 6) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (5, 1) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (5, 1) -> (6, 6) -> (2, 6) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (5, 1) -> (6, 6) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (6, 6) -> (2, 6) -> (5, 1) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (6, 6) -> (2, 6) -> (6, 3)
Caminho encontrado:
(0, 7) -> (2, 2) -> (6, 6) -> (5, 1) -> (2, 6) -> (6, 3)
```

Figura 17 Procura de todos os caminhos possíveis entre duas frequências FASE 2

4.10 – Interseções entre duas frequências

```
Escolha: 8
Indique a primeira frequÛncia (ex: A): A
Indique a segunda frequÛncia (ex: B): B

Intersecþ§es (pares A-B):
Antena A (6,3) Antena B (2,2)
Antena A (6,3) Antena B (2,6)
Antena A (6,3) Antena B (5,1)
Antena A (6,6) Antena B (2,2)
Antena A (6,6) Antena B (2,6)
Antena A (6,6) Antena B (5,1)
```

Figura 18 Interseções FASE 2

Conclusão

A segunda fase do projeto permitiu consolidar e aplicar os conceitos fundamentais da **teoria dos grafos** na resolução de um problema prático: a análise e representação de uma rede de antenas com base nas suas localizações e frequências de ressonância. A implementação em linguagem C reforçou o domínio de estruturas dinâmicas como **listas ligadas**, **pilhas**, **filas** e o uso de **ponteiros**, essenciais para a gestão eficiente de memória e flexibilidade estrutural.

Com esta fase, foram atingidos todos os objetivos propostos, reforçando a capacidade de análise algorítmica sobre grafos e o desenvolvimento de software orientado a estruturas de dados dinâmicas.

GITHUB

Fase 1: https://github.com/JoseSousa31552/TrabalhoEDAFase1

Fase 2: https://github.com/JoseSousa31552/EDA_FASE_2