PROJETO DE AVALIAÇÃO
ESTRUTURAS DE DADOS AVANÇADAS

LICENCIATURA EM ENGENHARIA DE SISTEMAS INFORMÁTICOS

Edgar Alexandre Vasquez Casal – a31026

Conteúdo

Resumo

Este relatório descreve a implementação completa do projeto da unidade curricular de Estruturas de Dados Avançadas (EDA), dividido em duas fases.
A **Fase 1** consistiu no desenvolvimento de um sistema de gestão de antenas e cálculo de efeitos nefastos, utilizando estruturas de dados dinâmicas (listas ligadas). Foram implementadas funcionalidades como inserção, remoção, carregamento de antenas a partir de ficheiros, e identificação de pontos de interferência.
A **Fase 2** estendeu o projeto com a utilização de grafos, representando as antenas como vértices e as ligações entre antenas de mesma frequência como arestas. Foram implementados percursos em profundidade (DFS), criação dinâmica de adjacências, e leitura/escrita do grafo em ficheiros binários.
O projeto foi desenvolvido integralmente em linguagem C, com ênfase na modularização, uso de memória dinâmica, e organização do código segundo boas práticas.

1.Introdução

Este projeto de avaliação individual da Unidade Curricular (UC) de **Estruturas de Dados Avançadas (EDA)**, lecionada no 2.º semestre do 1.º ano, tem como principal objetivo o reforço e aplicação dos conhecimentos adquiridos ao longo do semestre, através da resolução de um problema prático.

O projeto foi desenvolvido em duas fases distintas. A **Fase 1** teve como foco a utilização de estruturas de dados dinâmicas, nomeadamente listas ligadas, para a gestão de antenas e o cálculo de efeitos nefastos resultantes de interferência por alinhamento e frequência.

A **Fase 2** introduziu o conceito de grafos, representando cada antena como um vértice, e estabelecendo ligações (arestas) entre antenas com a mesma frequência. Esta fase permitiu aplicar algoritmos de percurso como a procura em profundidade (DFS) e trabalhar com armazenamento binário para guardar e recuperar o grafo.

A implementação foi realizada na linguagem de programação C, com enfoque na modularização, utilização de memória dinâmica, e documentação estruturada com recurso ao Doxygen. Este documento apresenta uma descrição detalhada da análise, implementação e testes realizados.

2. Análise e Especificação

O presente capítulo apresenta os requisitos funcionais do sistema, bem como as estruturas de dados utilizadas e a modelação lógica do funcionamento da aplicação.

2.1. Requisitos Funcionais

Fase 1 – Listas Ligadas

- Inserir e remover antenas com base nas coordenadas.
- Permitir o carregamento de antenas a partir de ficheiros.
- Calcular efeitos nefastos.
- Visualizar listas de antenas e de efeitos nefastos.
- Representar o mapa original com indicações dos pontos de interferência.

Fase 2 – Grafos

- Representar o sistema de antenas através de um grafo (lista de adjacências).
- Criar arestas (ligações) entre antenas com a mesma frequência.
- Implementar percursos sobre o grafo (DFS)
- Guardar e carregar o grafo num ficheiro binário.

2.2. Estruturas de Dados

Foram utilizadas estruturas de dados dinâmicas, maioritariamente listas ligadas, para representar tanto a lista de antenas como as relações entre elas.
Na **Fase 1**, recorreu-se a listas ligadas simples para armazenar antenas e pontos com efeitos nefastos.

Na **Fase 2**, as antenas passaram a ser representadas como vértices de um grafo, usando listas de adjacência, o que permitiu modelar ligações entre antenas com a mesma frequência e aplicar algoritmos como DFS (Depth-First Search).

Estrutura Antena

- **char freq:** frequência da antena.
- **int x, y:** coordenadas da antena**.**
- **struct Antena* prox:** apontador para a próxima antena.

```
typedef struct Antena {
    char freq;
    int x, y;
    struct Antena* prox;
}Antena;
```

A Figura 1 ilustra a estrutura Antena, que representa uma lista ligada de elementos, onde cada nó contém a frequência, as coordenadas da antena e um apontador para o nó seguinte na lista.

Estrutura Nefasto

- **int x, y:** coordenadas do ponto de interferência ou efeitos nefastos**.**
- **struct Nefasto* prox:** apontador para o próximo ponto.

```
typedef struct Nefasto {
    int x, y;
    struct Nefasto* prox;
}Nefasto;
```

Figura - Representação da estrutura de dados Nefasto (Fase 1)

A figura 2 ilustra a estrutura Nefasto, que representa uma lista ligada de pontos de interferência, cada um contendo as suas coordenadas e o apontador para o próximo nó.

Estrutura Vértice

Representa uma antena com:

- **char freq:** frequência da antena.
- **int x, y:** coordenadas da antena**.**
- **int visitado:** campo auxiliar para percursos (DFS)
- **struct Aresta* adj:** lista ligada de adjacências (arestas)
- **struct Vertice* prox:** apontador para a próxima antena.

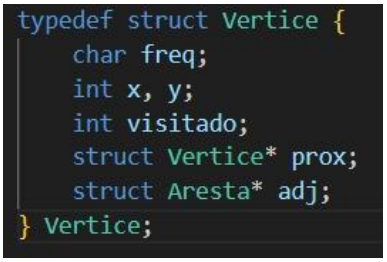


Figura 3 - Representação da estrutura de dados Vertice (Fase 2)

A Figura 3 ilustra a estrutura Vertice, utilizada para representar uma antena num grafo. Cada vértice armazena a frequência da antena, as suas coordenadas (x, y), um campo auxiliar visitado para percursos no grafo, um apontador para a sua lista de adjacências (adj), e um apontador para o próximo vértice na lista (prox), formando uma lista ligada de vértices

Estrutura Aresta

Representa uma ligação entre duas antenas:

- **struct Vertice* destino:** antena de destino
- **int peso:** valor da ligação (não usado ativamente no projeto)
- **struct Aresta* prox:** próxima ligação na lista de adjacências

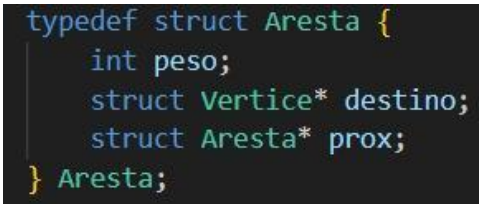


Figura 4 - Representação da estrutura de dados Aresta (Fase 2)

A Figura 4 representa a estrutura Aresta, que modela uma ligação entre duas antenas (vértices) com a mesma frequência.
Cada aresta contém um apontador para o vértice de destino (destino), um campo peso (não utilizado ativamente neste projeto, mas reservado para extensões), e um apontador para a próxima aresta (prox), formando assim uma lista ligada de adjacências.

Estrutura Grafo

Representa um grafo:

- **struct Vertice* h:** início da lista de vértices
- **int numVertices:** número de vértices

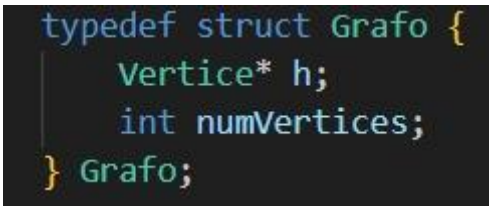


Figura 5 - Representação da estrutura de dados Grafo (Fase 2)

A Figura 5 mostra a estrutura Grafo, opcionalmente utilizada para encapsular o grafo como um todo.
Esta estrutura contém um apontador para o início da lista de vértices (h) e um inteiro que contabiliza o número total de antenas no grafo (numVertices).
Apesar de não ser essencial à lógica do programa, a sua utilização facilita a modularização do sistema.

2.3. Modelação e Arquitetura

O sistema está dividido em módulos independentes, com foco na reutilização e modularidade:

- **funcoes.c/h**: operações sobre antenas, adjacências, percursos e ficheiros.
- **main.c**: ponto de entrada do programa, onde são testadas todas as funcionalidades.
- **Organização por Fases**:
    - Fase 1 lida com listas ligadas e efeitos nefastos.
    - Fase 2 introduz grafos com percursos.

3. Implementação

O presente capítulo descreve, de forma detalhada, as principais funcionalidades implementadas no sistema, bem como o funcionamento interno de cada uma das operações previstas.

3.1. Fase 1 – Gestão de Antenas e Efeitos Nefastos

3.1.1. Carregamento de Antenas (a partir de ficheiro texto)

A função carregarAntenas(char* nomeFicheiro) lê um ficheiro de texto que representa um mapa de antenas, onde cada caractere pode indicar a presença de uma antena com uma determinada frequência. Esta função percorre o conteúdo do ficheiro, identifica os caracteres válidos (letras do alfabeto) e cria uma lista ligada, onde cada nó corresponde a uma antena com as respetivas coordenadas e frequência.

3.1.2. Inserção e Remoção de Antenas

As funções inserirAntena(Antena* h, char freq, int x, int y) e removerAntena(Antena* h, int x, int y) permitem adicionar uma nova antena ao início da lista ligada e remover uma antena existente com base nas suas coordenadas. A inserção é feita diretamente na cabeça da lista, enquanto a remoção percorre os nós até encontrar o que corresponde às coordenadas indicadas.

3.1.3. Cálculo de Efeitos Nefastos

A função efeitoNefasto(Antena* h) identifica pares de antenas com a mesma frequência e calcula os pontos onde ocorre interferência. O critério utilizado baseia-se em:
se a antena A está ao dobro da distância da antena B, o ponto de interferência encontra-se no ponto P = (2x1 - x2, 2y1 - y2). Os pontos válidos são armazenados numa nova lista ligada representada pela estrutura Nefasto.

3.1.4. Visualização

Foram implementadas três funções principais para visualização:
imprimirAntenas(Antena* h): apresenta a lista de antenas de forma tabular.
imprimirNefasto(Nefasto* h): apresenta os pontos de interferência de forma tabular.
imprimirAntenasNefasto(char* nomeFicheiro, Nefasto* h): representa o mapa original com marcações (#) nas posições dos efeitos nefastos.

Estas funções permitem verificar a estrutura dos dados carregados, bem como os resultados dos cálculos efetuados.

3.2. Fase 2 – Representação em Grafo

3.2.1. Estrutura do Grafo e Criação do Grafo

Nesta fase, as antenas são representadas como vértices de um grafo, onde apenas antenas com a mesma frequência podem estar ligadas entre si por arestas.
A estrutura Vertice representa uma antena, contendo os seus dados (frequência e coordenadas), uma lista ligada de adjacências (Aresta* adj) e campos auxiliares como visitado. A estrutura Aresta representa uma ligação entre duas antenas com a mesma frequência.
A criação do grafo é feita pela função criarGrafoDeFicheiro(), que lê um ficheiro de texto e cria dinamicamente os vértices com base nas posições das antenas. Cada antena é inserida de forma ordenada numa lista ligada.

3.2.2. Criação de Adjacências entre Antenas

Após o carregamento das antenas, a função criarAdjacencia() permite estabelecer ligações (arestas) entre pares de antenas com a mesma frequência. A verificação é feita entre todos os pares de vértices, e caso tenham a mesma frequência, é criada uma aresta que liga esses dois vértices.
Cada aresta criada é inserida na lista de adjacências do respetivo vértice através da função inserirAdjacencia(). Deste modo, forma-se um grafo não dirigido com listas de adjacência.

3.2.3. Percurso em Profundidade (DFS)

Para explorar o grafo, foi implementada a função depthFirstTraversal() que executa um percurso em profundidade (DFS). O algoritmo visita todos os vértices alcançáveis a partir de uma antena inicial, seguindo recursivamente as suas adjacências.
Durante a travessia, cada vértice é marcado como visitado usando o campo visitado, e a função resetVisitado() permite reiniciar esse estado para novos percursos.

3.2.4. Gravação e Leitura de Ficheiro Binário

Com o objetivo de guardar o estado atual do grafo, foram implementadas duas funções: gravarFicheiroBinario() e lerFicheiroBinario().

- gravarFicheiroBinario() percorre todos os vértices e suas adjacências, guardando os dados num ficheiro binário, incluindo o número de adjacências de cada antena.
- lerFicheiroBinario() reconstrói a lista de vértices a partir do ficheiro binário, criando novamente as antenas e restaurando as suas ligações.

4. Organização do Código

O projeto foi estruturado em ficheiros separados por responsabilidades, promovendo modularidade e reutilização de código entre fases distintas.

- funcoes.c / funcoes.h: Contêm todas as funções de manipulação de antenas, arestas e grafos (criação, inserção, DFS, gravação em ficheiro, etc.).
- main.c: Responsável apenas pelos testes ao sistema e chamadas às funções do módulo.
  O main.c não contém lógica de estrutura de dados, promovendo separação entre lógica e interface.
- Makefile: Utilizado para compilar o projeto com suporte à criação de uma biblioteca estática (libfuncoes.a). Esta abordagem melhora a organização e permite compilação mais rápida.

Na Fase 2, foi criada uma biblioteca estática com todas as funções relacionadas com grafos, antenas e adjacências. Essa biblioteca é compilada como libfuncoes.a e permite uma melhor organização, reutilização e manutenção do código.

5. Testes Realizados

Para validar o funcionamento do sistema, foram realizados diversos testes práticos diretamente na função main. Estes testes cobrem os principais casos de uso e funcionalidades desenvolvidas

5.1. Testes da Fase 1 – Listas Ligadas

Teste 1: Inserção, Remoção e Impressão de Antenas

Neste teste, foram inseridas manualmente várias antenas na lista ligada. Em seguida, foi feita a remoção de uma antena por coordenadas e a impressão dos dados em formato tabular.

```c
int main () {
    Antena* lista = NULL;
    Nefasto* listaEfeitoNefasto = NULL;


    //Teste de criar/inserir antenas numa lista ligada, remoção de uma antena e imprimir antenas de uma lista ligada em forma tabular
    lista = inserirAntena(lista,'A', 0, 0);
    lista = inserirAntena(lista,'A', 0, 1);
    lista = inserirAntena(lista,'A', 1, 1);
    lista = inserirAntena(lista,'B', 1, 2);
    imprimirAntenas(lista);
    printf("\n");

    removerAntena(lista, 1, 1);
    imprimirAntenas(lista);
    printf("\n");
```

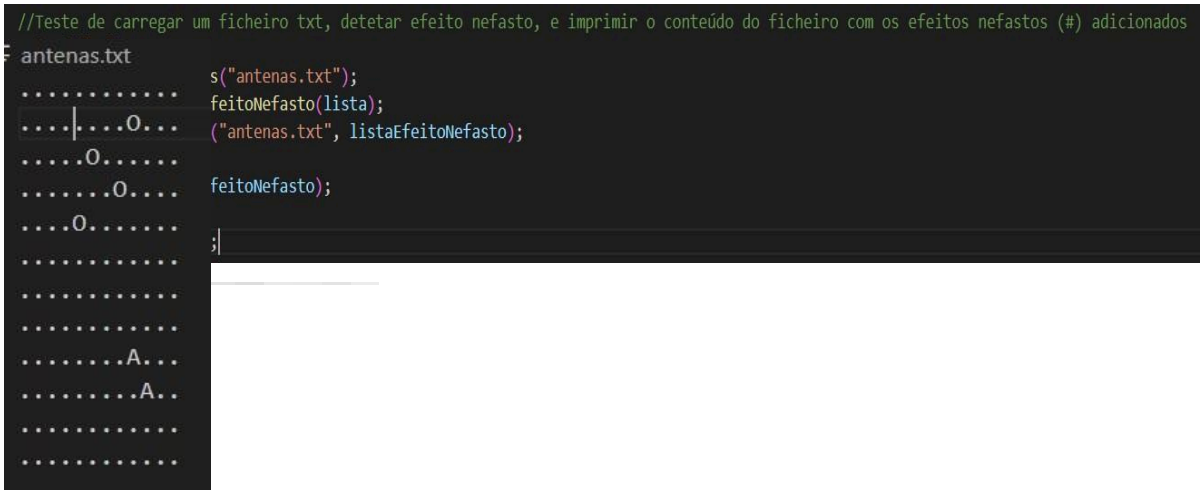
O resultado demonstrou que a lista é atualizada corretamente depois da remoção e que os dados são apresentados de forma legível.

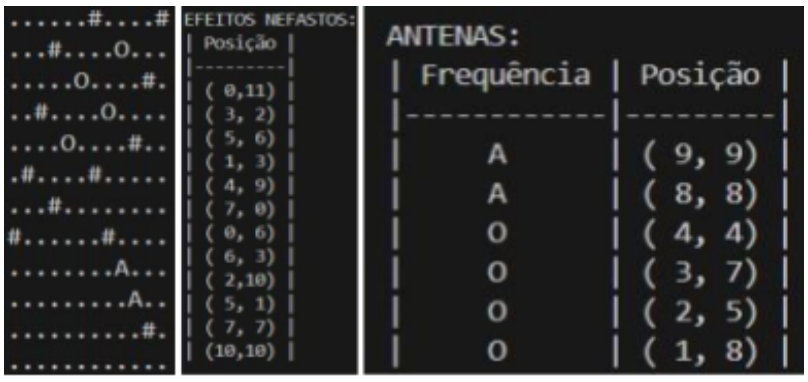
```
root@quedas:/mnt/c/Users/edgar/Desktop/IPCA/
ANTENAS:
| Frequência | Posição |
|------------|---------|
|     B      | ( 1, 2) |
|     A      | ( 1, 1) |
|     A      | ( 0, 1) |
|     A      | ( 0, 0) |

ANTENAS:
| Frequência | Posição |
|------------|---------|
|     B      | ( 1, 2) |
|     A      | ( 0, 1) |
|     A      | ( 0, 0) |
```

A Figura 6 mostra o código executado para o Teste 1, enquanto a Figura 7 apresenta o respetivo output obtido.

Teste 2: Carregamento de Ficheiro, Cálculo de Efeitos Nefastos e Visualização

Este teste envolveu o carregamento de um ficheiro antenas.txt com o mapa de antenas, seguido do cálculo dos efeitos nefastos e da visualização dos mesmos no mapa.





A Figura 9 apresenta o conteúdo do ficheiro antenas.txt, utilizado como entrada neste teste. Figura 8 mostra o código executado para o Teste 2, e a Figura 10 apresenta o resultado da visualização com efeitos nefastos sobre o mapa original.
Os testes realizados comprovaram que os efeitos nefastos foram corretamente identificados e representados com o caractere #, assim como a impressão das listas correspondentes, que está funcional.

5.1. Testes da Fase 1 – Listas Ligadas

Teste 3 – Criação do Grafo a partir de Ficheiro e Visualização das Antenas Carregadas

Foi utilizada a função criarGrafoDeFicheiro() para converter o conteúdo do ficheiro antenas.txt numa lista ligada de vértices (antenas).

```c
int main () {
    Grafo* grafo = NULL;
    Vertice* lista = NULL;
    bool resultado;

    // 1. Criar grafo a partir do ficheiro de texto
    printf("Carregar o grafo do ficheiro 'antenas.txt'\n");
    grafo = criarGrafoDeFicheiro("antenas.txt");
    if (grafo == NULL || grafo->h == NULL) {
        printf("Erro ao carregar grafo.\n");
        return 1;
    }

    lista = grafo->h;

    // 2. Mostrar as antenas carregadas
    printf("\n--- Antenas carregadas ---\n");
    if (!mostrarAntenas(lista)) {
        printf("Lista de antenas vazia.\n");
    }
```

Figura 11 - Código executado no teste 3

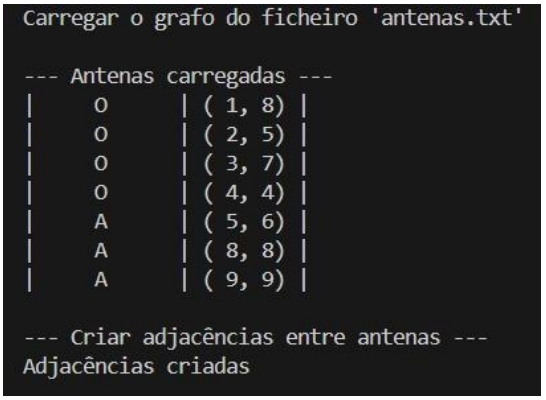


Figura 12 - Resultado do carregamento de antenas e criação de adjacências

A Figura 11 apresenta o código responsável por executar as primeiras funcionalidades da Fase 2. Inicialmente, é criado um grafo a partir da leitura do ficheiro antenas.txt, contendo o mapa de antenas. Em seguida, as antenas carregadas são apresentadas em formato tabular através da função mostrarAntenas, permitindo validar visualmente se o ficheiro foi corretamente processado.

A Figura 12 mostra o resultado da execução do código representado na Figura 11. São exibidas todas as antenas carregadas a partir do ficheiro e é confirmada a criação das adjacências entre antenas com a mesma frequência. Esta verificação assegura que os dados foram interpretados e estruturados corretamente em memória.

Teste 4 – Criação de Adjacências entre Antenas

Para todas as antenas com a mesma frequência, foram criadas ligações (arestas) entre elas com a função criarAdjacencia(), seguidas de inserirAdjacencia(). Verificou-se corretamente a estrutura de grafos com mostrarAdjacencias().

```
    // 3. Criar adjacências entre antenas com a mesma frequência
printf("\n--- Criar adjacências entre antenas ---\n");
Vertice* v1 = lista;
while (v1 != NULL) {
    Vertice* v2 = v1->prox;
    while (v2 != NULL) {
        Aresta* nova = criarAdjacencia(v1, v2);
        if (nova != NULL) {
            inserirAdjacencia(v1, nova);
            //printf("Adjacência criada de (%d,%d) até (%d,%d)\n", v1->x, v1->y, v2->x, v2->y);
        }

        Aresta* nova2 = criarAdjacencia(v2, v1);
        if (nova2 != NULL) {
            inserirAdjacencia(v2, nova2);
            //printf("Adjacência criada de (%d,%d) para (%d,%d)\n", v2->x, v2->y, v1->x, v1->y);
        }

        v2 = v2->prox;
    }
    v1 = v1->prox;
}
printf("Adjacências criadas\n");

// 4. Mostrar adjacências de todas as antenas
printf("\n--- Adjacências de cada antena ---\n");
Vertice* atual = lista;
while (atual != NULL) {
    printf("Antena %c com as coordenadas (%d,%d) tem adjacências com:\n", atual->freq, atual->x, atual->y);
    if (!mostrarAdjacencias(atual)) {
        printf("Sem adjacências.\n");
    }
    atual = atual->prox;
}
```

Figura 13 - Código executado no Teste 4

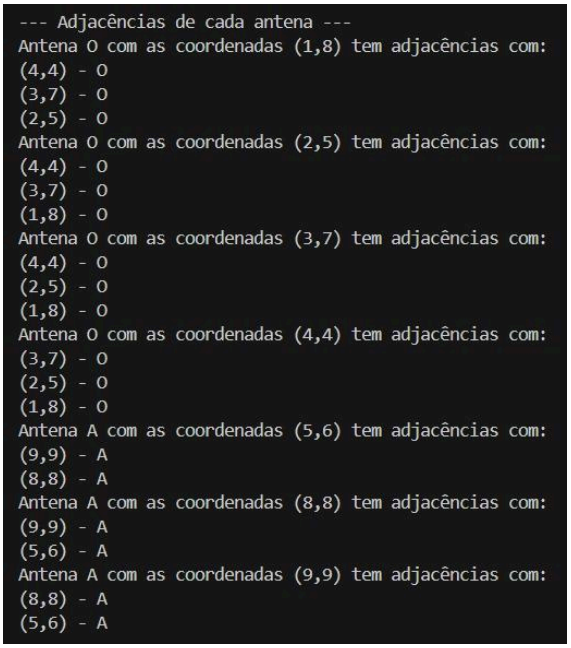


Figura 14 - Saída do terminal com a lista de adjacências por antena

A Figura 13 apresenta o código responsável pela criação das adjacências entre antenas com a mesma frequência. O processo percorre todos os pares possíveis de vértices e, caso tenham a mesma frequência, cria e insere as respetivas arestas na lista de adjacências de cada vértice. Em seguida, é realizada a visualização de todas as adjacências, listando para cada antena outras antenas com a mesma frequência interligadas.

A Figura 14 mostra o resultado da execução do código da Figura 13. Cada antena é listada com as suas coordenadas e as coordenadas das antenas com que está ligada. O teste comprova que a criação das adjacências está a funcionar corretamente, com base na correspondência de frequências, e que o grafo foi corretamente construído com listas ligadas de adjacência.

Teste 5 – Percurso em Profundidade (DFS)

Foi testada a função depthFirstTraversal() para verificar a conectividade entre antenas com a mesma frequência. O campo visitado foi corretamente atualizado e posteriormente reiniciado com resetVisitado().

```c
// 5. DFS - Percurso em profundidade
printf("\n--- DFS (Depth First Traversal) ---\n");
resetVisitado(lista);
if (!depthFirstTraversal(lista)) {
    printf("Erro: antena de origem inválida.\n");
} else {
    printf("DFS concluído.\n");
}

// 6. Remover antena (por exemplo, coordenada 1,8)
printf("\n--- Remover antena na posição (1,8) ---\n");
lista = removerAntena(lista, 1, 8, &resultado);
if (resultado) {
    printf("Antena removida com sucesso.\n");
} else {
    printf("Antena não encontrada.\n");
}

// 7. Mostrar antenas após remoção
printf("\n--- Lista atualizada de antenas ---\n");
if (!mostrarAntenas(lista)) {
    printf("Lista vazia.\n");
}
```

Figura 15 - Código executado no Teste 5

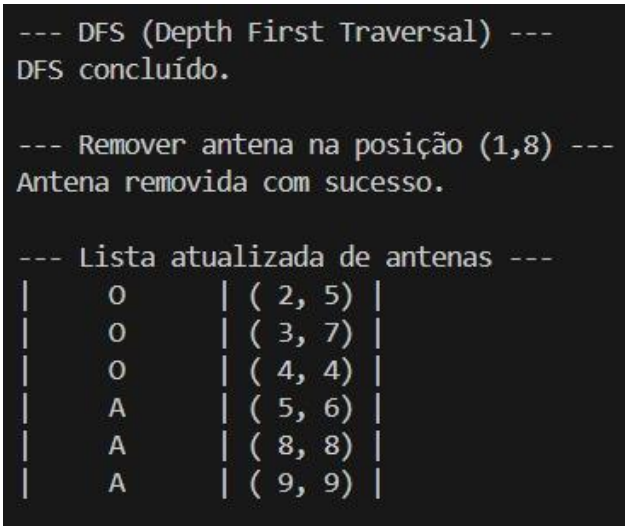


Figura 16 - Saída do terminal com o resultado do Teste 5

A Figura 15 apresenta o código responsável por realizar três operações consecutivas:

- Um percurso em profundidade (DFS) a partir do primeiro vértice da lista, marcando como visitados todos os vértices conectados por adjacências.
- A remoção de uma antena com coordenadas específicas (neste caso, (1,8)).
- A visualização da lista de antenas atualizada após a remoção.

Estas operações demonstram a integração entre as funcionalidades de percorrer, modificar e apresentar os dados manipulados pelo grafo. O percurso DFS é útil para analisar a conectividade entre vértices, enquanto a função de remoção permite alterar dinamicamente a estrutura da rede de antenas.

A Figura 16 mostra o resultado da execução, indicando que o percurso DFS foi concluído com sucesso, a antena da posição (1,8) foi removida corretamente, e a lista resultante mostra as restantes antenas. A funcionalidade de remoção comprovadamente atualiza a estrutura de forma adequada, como evidenciado pela ausência da antena removida na visualização final

Teste 6 – Gravação e Leitura de Ficheiro Binário

A estrutura do grafo foi guardada com gravarFicheiroBinario() e lida novamente com lerFicheiroBinario(). Confirmou-se que os dados das antenas e das suas adjacências foram mantidos corretamente entre execuções.

```
// 8. Gravar para ficheiro binário
printf("\n--- Gravar antenas no ficheiro binário ---\n");
if (gravarFicheiroBinario(lista)) {
    printf("Ficheiro 'grafo.bin' criado com sucesso.\n");
} else {
    printf("Erro ao gravar ficheiro binário.\n");
}

// 9. Ler novamente do ficheiro binário
printf("\n--- Ler ficheiro binário ---\n");
bool resLeitura = false;
Vertice* listaLida = lerFicheiroBinario("grafo.bin", &resLeitura);

if (resLeitura) {
    printf("Antenas lidas do ficheiro binário:\n");
    mostrarAntenas(listaLida);
} else {
    printf("Erro ao ler do ficheiro binário.\n");
}
```

Figura 17 - Código executado no Teste 6

```
--- Gravar antenas no ficheiro binário ---
Ficheiro 'grafo.bin' criado com sucesso.

--- Ler ficheiro binário ---
Antenas lidas do ficheiro binário:
|     0      | ( 2, 5) |
|     0      | ( 3, 7) |
|     0      | ( 4, 4) |
|     A      | ( 5, 6) |
|     A      | ( 8, 8) |
|     A      | ( 9, 9) |
```

Figura 18 - Output da gravação e leitura de antenas do ficheiro binário

Este teste avalia a capacidade do sistema de gravar a lista de antenas num ficheiro binário (grafo.bin) e, posteriormente, reconstruí-la corretamente a partir desse ficheiro. A **Figura 17** apresenta o trecho de código utilizado para gravar e ler o ficheiro binário, enquanto a **Figura 18** mostra o resultado da sua execução. Para cada antena, foram gravados a frequência e as coordenadas, seguidos dos dados das antenas adjacentes (também com frequência e coordenadas). Após a leitura, as antenas foram corretamente restauradas e listadas, confirmando que o processo foi bem-sucedido.

6. Conclusões

O desenvolvimento deste projeto permitiu consolidar conhecimentos em estruturas de dados dinâmicas, nomeadamente listas ligadas, bem como aprofundar competências na linguagem de programação C. A resolução do problema proposto, centrado na deteção de efeitos nefastos entre antenas com a mesma frequência, demonstrou a aplicabilidade de conceitos teóricos em cenários práticos.

Existem algumas melhorias ao nível de implementação como:

- Separar a funcionalidade de criação de antenas da função de inserção, permitindo maior clareza no código.

- Reduzir o uso de funções com retorno void, e promover o uso de mensagens de erro que facilitem a validação.

- Evitar o uso de printf dentro das funções de lógica, para a separação entre lógica e apresentação de dados.

7. Referências

W3Schools. (2025). C Pointers. em:
https://www.w3schools.com/c/c_pointers.php
ChatGPT (versão GPT-4), a partir de:
https://chat.openai.com/
Estruturando. (2017). Estrutura de Dados em C - Lista Simplesmente Encadeada - Criar, Inserir Inicio, Imprimir [Vídeo]. Youtube.
https://www.youtube.com/watch?v=Uk8v7gB2rHk
Programe seu Futuro. (2022). Curso de Programação C | Lista encadeada, lista duplamente encadeada e lista circular | aula 242 [Vídeo]. Youtube.
https://www.youtube.com/watch?v=biTMaMxWLRc&t

8. Anexos

Anexo A: Link para o repositório com o código-fonte completo:
https://github.com/edgarcasal/projeto_EDA.git

Anexo B: Documentação Técnica gerada automaticamente com Doxygen (index.html).