**Instituto Politécnico do Cávado e do Ave**

**Escola Superior de Tecnologia**

# Estrutura de Dados Avançada

## Licenciatura em Engenharia de Sistemas Informáticos

## Trabalho Prático

1ºFase

Dani Carvalho da Cruz– a23016

maio de 2025

# *Índice de Ilustrações*

# Índice

# 1. Resumo

O presente trabalho foi desenvolvido no âmbito da unidade curricular de Estruturas de Dados Avançadas (EDA), sendo dividido em duas fases complementares.

Na primeira fase, recorreu-se à utilização de listas ligadas simples para representar uma cidade com antenas posicionadas numa matriz, cada uma sintonizada numa frequência específica.

Implementou-se a leitura da matriz a partir de ficheiro, bem como operações de inserção, remoção e listagem de antenas. Além disso, foi desenvolvido um algoritmo para identificar automaticamente localizações com efeito nefasto resultantes do alinhamento entre antenas da mesma frequência.

Na segunda fase, evoluiu-se para uma abordagem mais complexa com a utilização de grafos, onde cada antena corresponde a um vértice, e as arestas representam conexões entre antenas com a mesma frequência. Foram aplicados algoritmos clássicos de procura em profundidade (DFS) e largura (BFS), identificação de caminhos entre antenas, e análise de interações entre frequências distintas.

## 1.2 Erros encontrados na Fase 1

Durante o desenvolvimento da primeira fase do projeto, foram identificadas algumas implementações que mostraram estar incorretas, no dia de defesa com o professor.

Estes erros foram analisados e corrigidos na fase seguinte, contribuindo para uma melhoria significativa da estrutura do código.

### 1.2.1 Estruturas desnecessárias

Inicialmente, foram criadas estruturas como "*listaNefastas"* e "*listaAntenas"*, que acabaram por introduzir complexidade desnecessária no código.

Verificou-se que a utilização de uma única estrutura do tipo Antena, contendo os campos necessários, permitiria alcançar os mesmos objetivos com maior simplicidade e clareza.

A redundância estrutural dificultava a manutenção, ou seja, uma matriz conteria duas vezes o que na segunda fase equivale a "primeiraAntena". Isto complicava o código desnecessariamente, pois tinha duas estruturas diferentes que continham Antena* que tinham o mesmo objetivo no final.

### 1.2.2 Gestão de incorretas de Nefastas

Foi detetado um erro lógico no armazenamento das posições com efeito nefasto. Quando duas antenas da mesma frequência formavam uma combinação válida, apenas uma das posições nefastas era armazenada, em vez de ambas em cada antena

Este problema agravava-se em operações de remoção de antenas, onde algumas posições deveriam ser removidas em duplicado, mas eram tratadas como únicas, resultando que ao eliminar uma antena apenas uma das posições nefastas eram eliminadas pois não continha o par completo.

### 1.2.3 Retornos incorretos

Outro erro identificado foi o uso indevido de funções com retorno *void* em contextos onde era necessário devolver resultados ou estados.

Funções que deveriam comunicar valores de retorno relevantes para o fluxo lógico estavam a ser declaradas como *void*, comprometendo a reutilização do código e dificultando a deteção de erros.

O uso de *void* foi posteriormente reservado apenas para funções cujo único objetivo é exibir informação na consola por exemplo, através de *printf*, em conformidade com boas práticas de programação.

## 2. Introdução

A segunda fase do projeto da unidade curricular de Estruturas de Dados Avançadas (EDA) teve como principal objetivo a aplicação de conceitos de grafos para representar e analisar a disposição de antenas numa cidade.

Após a conclusão da fase inicial, que utilizava listas ligadas para armazenar e manipular dados das antenas, tornou-se necessário adotar uma abordagem mais robusta e relacional, permitindo explorar conexões entre antenas com a mesma frequência.

Nesta fase, cada antena passou a ser representada como um vértice num grafo, e as ligações entre antenas com a mesma frequência foram modeladas através de arestas. Esta estrutura permitiu realizar operações de análise mais avançadas, como percursos entre antenas e intersecções entre antenas de diferentes frequências.

## 3. Objetivos

- Conceber e implementar uma estrutura dinâmica para representar antenas e os seus efeitos nefastos.
- Identificar posições com efeito nefasto com base na disposição de antenas.
- Evoluir de uma abordagem linear (listas) para uma abordagem relacional (grafos).
- Aplicar algoritmos clássicos de procura e análise em grafos.
- Promover a modularidade, documentação e legibilidade do código desenvolvido.

# 4. Trabalho desenvolvido a nível Estrutural

## 4.1 Estrutura de Dados de *Grafo*

```
typedef struct
{
    struct Vertice* primVertice;
    int dimensao[2];
}Grafo;
```

***Figura 1 – Struct Grafo***

Estrutura principal que irá armazenar todas as informações em relação a um grafo. Esta estrutura bastante intuitiva, possui um apontador do tipo *Vertice* que será o primeiro vértice lido ou inserido e também a dimensão da matriz, linhas e colunas, que armazena a matriz inserida.

## 4.2 Estrutura de Dados de *Vertice*

```
typedef struct
{
    Dados dados;
    struct Vertice* proxVertice;
    struct Aresta* primAresta;
    bool visitado;
} Vertice;
```

***Figura 2 – Struct Vertice***

A estrutura *Vertice*, apresentada na *figura 2*, representa cada antena individual como um nó de um grafo. Este grafo foi construído com base em listas de adjacência, uma abordagem eficiente para representar ligações entre elementos.

Outra alteração feita, foi uma “dica” dada pelo professor que era dividir campos que definem um *Vertice,* como coordenadas e frequência em uma struct independente e por isso temos *Dados* que contém os seguintes campos:

#### 4.2.1 Estrutura de Dados de *Dados*

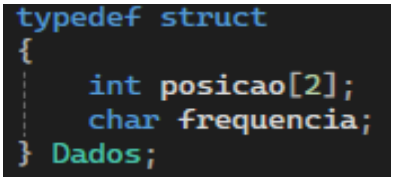


***Figura 3 – Struct Dados***

Contém um array de dois inteiros, posição, que representa as coordenadas da antena na matriz da cidade (linha e coluna), e um *char* chamado frequência, que indica a frequência de ressonância da antena. Esta estrutura é utilizada dentro de cada vértice do grafo para identificar a antena representada.

### 4.2 Estrutura de Dados de *Vertice* (continuação)

A ligação entre os vértices do grafo é feita através do campo *proxVertice*, um apontador para o próximo elemento na lista de vértices. Desta forma, todos os vértices do grafo estão interligados numa lista ligada simples, facilitando a iteração sequencial sobre todas as antenas representadas no sistema.

Além disso, cada vértice possui um apontador chamado *primAresta*, que aponta para a primeira aresta da lista de adjacência associada a esse vértice. Cada aresta representa uma ligação entre duas antenas com a mesma frequência, estabelecendo assim a conectividade dentro do grafo.

A estrutura das *Arestas*, por sua vez, deverá conter um apontador para o vértice de destino e um apontador para a próxima aresta da lista, formando uma sublista ligada de ligações para cada vértice.

Por fim, o campo *visitado* é uma variável booleano utilizado em algoritmos de procura como a busca em profundidade (DFS) ou busca em largura (BFS), para verificar se o vértice já foi visitado.

### 4.3 Estrutura de Dados de *Aresta*

```
typedef struct
{
    Vertice* adjVertice;
    struct Aresta* proxAresta;
    struct Nefasta* primNefasta;
}Aresta;
```

***Figura 4 – Struct Aresta***

A estrutura *Aresta* é usada para representar as conexões entre antenas no grafo, ou seja, as ligações entre vértices que partilham a mesma frequência.

O campo *adjVertice* aponta para o vértice adjacente, permitindo identificar qual é a antena ligada ao vértice de origem. Esta ligação reflete a existência de uma relação direta entre duas antenas com a mesma frequência de ressonância.

O campo *proxAresta* estabelece a continuidade da lista de adjacência, permitindo que múltiplas arestas sejam encadeadas para um mesmo vértice. Desta forma, é possível percorrer todas as ligações existentes a partir de uma antena, analisando o seu grau de conectividade.

Por fim, o campo *primNefasta* aponta para a primeira posição da lista de localizações com efeito nefasto resultantes da ligação entre as duas antenas conectadas.

### 4.4 Estrutura de Dados de *Nefasta*

```
typedef struct
{
    struct Nefasta* proxNefasta;
    Dados dados;
}Nefasta;
```

***Figura 5 - Nefasta***

A estrutura *Nefasta* é utilizada para representar uma posição com efeito nefasto resultante da ligação entre duas antenas.

Contém o campo *dados*, com a posição e frequência associadas, e um apontador *proxNefasta* para formar uma lista ligada de todas as posições nefastas associadas a uma aresta.

### 4.5 Estrutura de Dados de *Espera*

```
typedef struct
{
    Vertice* vertice;
    struct Espera* proxVertice;
} Espera;
```

***Figura 6 – Struct Espera***

A estrutura *Espera* é usada para implementar uma *fila* de vértices, geralmente utilizada nos algoritmos de *procura em largura (BFS)*. O campo *vertice* guarda um apontador para o vértice correspondente no grafo, enquanto *proxVertice* aponta para o próximo elemento da *queue*.

Através desta estrutura ligada é possível manter a ordem de visita dos vértices durante a execução do algoritmo, garantindo que os vértices mais próximos são explorados primeiro. Trata-se de uma estrutura auxiliar, temporária, usada apenas durante o processamento do grafo.

## 4.6 Estrutura de Dados de *VerticeFile*

```
typedef struct
{
    Dados dados;
    bool visitado;
}VerticeFile;
```

***Figura 7 – VerticeFile***

A estrutura *VerticeFile* foi criada para armazenar vértices num ficheiro binário de forma simplificada. Contém os campos essenciais: dados, que guarda a posição e frequência da antena, e visitado, que indica se o vértice já foi visitado.

Esta estrutura permite guardar e carregar o estado do grafo de forma eficiente, isolando apenas a informação relevante, sem incluir apontador.

## 4.7 Estrutura de Dados de *ArestaFile*

```
typedef struct
{
    int origemPos[2];
    int destinoPos[2];
} ArestaFile;
```

***Figura 8 – Struct ArestaFile***

A estrutura *ArestaFile* foi concebida para armazenar informações de arestas em ficheiros binários, sem depender de ponteiros. Contém dois arrays de inteiros: *origemPos* e *destinoPos*, que representam, respetivamente, as coordenadas da antena de origem e da antena de destino.

### 4.8 Estrutura de Dados de *NefastaFile*

```
typedef struct
{
    int posicao[2];
    char frequencia;
} NefastaFile;
```

***Figura 9 – Struct NefastaFile***

A estrutura *NefastaFile* serve para armazenar, em ficheiros binários o mesmo que arestas, as posições com efeito nefasto identificadas no grafo.

Contém um array *posicao* com as coordenadas da localização afetada e um *char* frequencia, que indica a frequência associada ao efeito. Esta estrutura permite guardar estas informações de forma leve e independente.

## 5. Funcionamento da Estruturas principais

### 5.1 Carregar vértices de um ficheiro Binário

A função *CarregarVertices* tem como objetivo ler de um ficheiro binário os vértices previamente guardados e reconstruir a lista ligada de vértices no grafo. Recebe como parâmetros um apontador para o grafo e o nome do ficheiro a ser lido.

Durante a leitura, é utilizada uma estrutura auxiliar do tipo *VerticeFile*, que contém apenas os dados essenciais para armazenar um vértice em ficheiro (posição, frequência e estado de visita), já referido anteriormente.

```
Grafo* CarregarVertices(Grafo* grafo,char* file)
{
    FILE* ficheiro = fopen(file, "rb");
    if (ficheiro == NULL)
    {
        return false;
    }

    VerticeFile vfile;
    Vertice* novo = NULL;
    Vertice* anterior = NULL;

    grafo->primVertice = NULL;

    while (fread(&vfile, sizeof(VerticeFile), 1, ficheiro) == 1)
    {
        novo = (Vertice*)malloc(sizeof(Vertice));
        novo->dados = vfile.dados;
        novo->visitado = vfile.visitado;
        novo->primAresta = NULL;
        novo->proxVertice = NULL;

        if (grafo->primVertice == NULL)
        {
            grafo->primVertice = novo;
        }
        else
        {
            anterior->proxVertice = novo;
        }

        anterior = novo;
    }
    fclose(ficheiro);
    return true;
}
```

***Figura 10 - CarregarVertices***

## 5.2 Guardar vértices de um ficheiro Binário

A função *GuardarVertices* é responsável por gravar todos os vértices de um grafo num ficheiro binário. Para isso, abre o ficheiro no modo de escrita binária ("wb"). Se o ficheiro não for aberto corretamente, a função termina de imediato, retornando false.

Em seguida, percorre a lista ligada de vértices através do apontador *auxVertice*, começando no primeiro vértice do grafo. Para cada vértice encontrado, copia os dados relevantes para uma estrutura auxiliar do tipo *VerticeFile*, nomeadamente a posição, a frequência e o estado de visita.

## 5.3 Operações de Grafos

### 5.3.1 Criar Grafo

A função *CriarGrafo* tem como objetivo criar um novo grafo com as dimensões da matriz lidas ou inseridas pelo utilizador. Para isso, aloca memória dinâmica para a estrutura Grafo. Caso a alocação falhe, retorna false.

Os valores de linhas e colunas são armazenados no array dimensão, definindo o tamanho da matriz em que as antenas estarão posicionadas. O apontador *primVertice* é inicializado a *NULL*, indicando que o grafo ainda não possui vértices.

```
Grafo* CriarGrafo(int linhas, int colunas)
{
    Grafo* grafo = (Grafo*)malloc(sizeof(Grafo));
    if (grafo == NULL)
    {
        return false;
    }
    grafo->dimensao[0] = linhas;
    grafo->dimensao[1] = colunas;
    grafo->primVertice = NULL;
    return grafo;
}
```

***Figura 11 - CriarGrafo***

### 5.3.2 Criar Vértice

A função *CriarVertice* é responsável por criar um novo vértice, correspondente a uma antena, com base na sua posição (linha e coluna) e frequência.

Os dados do vértice são preenchidos: as coordenadas da antena são atribuídas ao campo posição, e a frequência é guardada no campo frequência. Os apontadores *proxVertice* e *primAresta* são inicializados a NULL, indicando que o vértice ainda não está ligado a outros nem possui arestas. O campo visitado também é definido como false, preparando-o para uso em algoritmos de procura.

```
Vertice* CriarVertice(Grafo* grafo, int linha, int coluna, char frequencia)
{
    Vertice* vertice = (Vertice*)malloc(sizeof(Vertice));
    if (vertice == NULL)
    {
        return false;
    }
    vertice->dados.posicao[0] = linha;
    vertice->dados.posicao[1] = coluna;
    vertice->dados.frequencia = frequencia;
    vertice->proxVertice = NULL;
    vertice->primAresta = NULL;
    vertice->visitado = false;
    AdicionarVertice(grafo, vertice);
    return vertice;
}
```

***Figura 12 - CriarVertice***

### 5.3.3 Eliminar Vértice

A função *EliminarVertice* tem como objetivo remover um vértice específico do grafo, identificado pelas suas coordenadas na matriz. Para isso, percorre a lista ligada de vértices até encontrar aquele que corresponde à posição indicada.

Ao encontrar o vértice, é feito um segundo percurso por todos os outros vértices do grafo para remover eventuais arestas que apontem para o vértice a eliminar, garantindo assim que não ficam ligações inválidas.

```
Vertice* EliminarVertice(Grafo* grafo,int linha, int coluna)
{
    Vertice* auxVertice = grafo->primVertice;
    Vertice* eliminar = NULL;

    while (auxVertice != NULL)
    {
        if (auxVertice->dados.posicao[0] == linha && auxVertice->dados.posicao[1])
        {
            Vertice* outro = grafo->primVertice;
            while (outro != NULL)
            {
                if (outro != auxVertice)
                {
                    EliminarAresta(outro, auxVertice);
                }
                outro = outro->proxVertice;
            }

            if (eliminar == NULL)
            {
                grafo->primVertice = auxVertice->proxVertice;
            }
            else
            {
                eliminar->proxVertice = auxVertice->proxVertice;
            }
            free(auxVertice);
            return grafo->primVertice;
        }
        eliminar = auxVertice;
        auxVertice = auxVertice->proxVertice;
    }
    return NULL;
}
```

***Figura 13 - EliminarVertice***

### 5.3.4 Atualizar Vértice

A função *EditarVertice* permite modificar a frequência de uma antena localizada numa posição específica da matriz do grafo. Para isso, começa por remover o vértice existente na posição indicada, utilizando a função *EliminarVertice*.

Em seguida, cria um novo vértice com os mesmos valores de linha e coluna, mas com a nova frequência passada como argumento. Este novo vértice é automaticamente inserido no grafo através da função *CriarVertice* e o apontador para o vértice editado é devolvido. Esta abordagem simples e direta garante que os dados são atualizados sem necessidade de manipular diretamente os campos internos do vértice.

```
Vertice* EditarVertice(Grafo* grafo, int linha, int coluna, char frequencia)
{
    EliminarVertice(grafo, linha, coluna);
    Vertice* verticeEditado = CriarVertice(grafo, linha, coluna, frequencia);
    return verticeEditado;
}
```

***Figura 28 – EditarVertice***

***Figura 15 - CriarArestaFigura 29 – EditarVertice***

### 5.3.5 Criar Aresta

A função *CriarAresta* tem como objetivo criar uma nova ligação (aresta) entre dois vértices de um grafo, representando a ligação entre duas antenas com a mesma frequência. Para isso, aloca dinamicamente memória para uma nova estrutura do tipo Aresta.

Se o vértice de origem (*verticeAtual*) ainda não tiver nenhuma aresta, a nova aresta é simplesmente atribuída ao campo *primAresta*. Caso contrário, percorre-se a lista de arestas até ao fim para inserir a nova aresta na última posição.

```
Aresta* CriarAresta(Vertice* verticeAtual, Vertice* verticeInserido)
{
    Aresta* arestaCriada = (Aresta*)malloc(sizeof(Aresta));
    if (arestaCriada == NULL)
    {
        return false;
    }

    arestaCriada->adjVertice = verticeInserido;
    arestaCriada->proxAresta = NULL;
    arestaCriada->primNefasta = NULL;
    if (verticeAtual->primAresta == NULL)
    {
        verticeAtual->primAresta = arestaCriada;
    }
    else
    {
        Aresta* auxAresta = verticeAtual->primAresta;
        while (auxAresta->proxAresta != NULL)
        {
            auxAresta = auxAresta->proxAresta;
        }
        auxAresta->proxAresta = arestaCriada;
    }
    return arestaCriada;
}
```

***Figura 15 - CriarAresta***

### 5.3.6 Eliminar Aresta

A função *EliminarAresta* tem como objetivo remover uma ligação (aresta) entre dois vértices do grafo: *verticeOrigem* e *verticeDestino*. Para isso, percorre a lista de arestas do vértice de origem, utilizando o ponteiro *auxAresta*.

Ao mesmo tempo, mantém um apontador auxiliar chamado eliminar para guardar a referência da aresta anterior. Durante o percurso, se encontrar uma aresta cujo vértice adjacente (*adjVertice*) seja igual ao vértice de destino, verifica se esta é a primeira aresta da lista. Se for, atualiza o apontador *primAresta* do vértice de origem para apontar para a aresta seguinte. Caso contrário, ajusta o encadeamento da lista para remover a aresta encontrada.

```
Aresta* EliminarAresta(Vertice* verticeOrigem, Vertice* verticeDestino)
{
    Aresta* auxAresta = verticeOrigem->primAresta;
    Aresta* eliminar = NULL;

    while (auxAresta != NULL)
    {
        if (auxAresta->adjVertice == verticeDestino)
        {
            if (eliminar == NULL)
            {
                verticeOrigem->primAresta = auxAresta->proxAresta;
            }
            else
            {
                eliminar->proxAresta = auxAresta->proxAresta;
            }
            free(auxAresta);
            return verticeOrigem->primAresta;
        }
        eliminar = auxAresta;
        auxAresta = auxAresta->proxAresta;
    }
    return NULL;
}
```

***Figura 16 - EliminarAresta***

### 5.3.7 Encontrar Vértice

A função *EncontrarVertice* tem como objetivo localizar um vértice no grafo com base nas suas coordenadas (linha e coluna). Para isso, percorre a lista ligada de vértices, começando no primeiro vértice do grafo, e compara a posição de cada um com os valores fornecidos como parâmetro.

Se encontrar um vértice cuja linha e coluna correspondam, retorna imediatamente esse vértice. Caso contrário, continua a percorrer a lista até ao final. Se não encontrar nenhum vértice com as coordenadas indicadas, retorna NULL.

```
Vertice* EncontrarVertice(Grafo* grafo, int linha, int coluna)
{
    Vertice* auxVertice = grafo->primVertice;
    while (auxVertice != NULL)
    {
        if (auxVertice->dados.posicao[0] == linha && auxVertice->dados.posicao[1])
        {
            return auxVertice;
        }
        auxVertice = auxVertice->proxVertice;
    }
    return NULL;
}
```

***Figura 17 – EncontrarVertice***

### 5.3.8 Percorrer Grafo

A função *PercorrerGrafo* tem como finalidade percorrer todos os vértices do grafo e identificar quais deles partilham a mesma frequência que um vértice específico passado como parâmetro. Durante a iteração, compara-se a frequência de cada vértice com a do vértice alvo, ignorando o próprio vértice.

Sempre que encontra um vértice com frequência igual, cria-se uma aresta entre esse vértice e o vértice de entrada através da função *CriarAresta*. Em seguida, é chamada a função *CriarPosicaoNefasta*, que calcula e associa as posições com efeito nefasto resultantes da ligação entre os dois vértices

```
bool PercorrerGrafo(Grafo* grafo, Vertice* vertice)
{
    Vertice* auxVertice = grafo->primVertice;
    while (auxVertice != NULL)
    {
        if (auxVertice->dados.frequencia == vertice->dados.frequencia && auxVertice != vertice)
        {
            CriarAresta(auxVertice, vertice);
            CriarPosicaoNefasta(auxVertice, vertice, grafo);
        }
        auxVertice = auxVertice->proxVertice;
    }
    return true;
}
```

***Figura 18 - PercorrerGrafo***

### 5.3.9 Mostrar Grafo

A função *MostraGrafo* tem como objetivo apresentar de forma organizada todos os elementos do grafo na consola. Inicia mostrando a lista de vértices através da função *ListaVertices*.

De seguida, percorre a lista ligada de vértices do grafo. Para cada vértice, invoca duas funções: *ListaArestas*, que imprime as ligações (arestas) entre esse vértice e *ListaNefastas* que apresenta as posições com efeito nefasto associadas a essas ligações.

```
void MostraGrafo(Grafo* grafo)
{
    Vertice* auxVertice = grafo->primVertice;
    ListaVertices(grafo);
    printf("\n");
    while (auxVertice != NULL)
    {
        ListaArestas(auxVertice);
        printf("\n");
        ListaNefastas(auxVertice);
        printf("\n");
        auxVertice = auxVertice->proxVertice;
    }
}
```

***Figura 19 - MostraGrafo***

# 6. Algoritmos de Procura

## 6.1 Algoritmo DFS (Depth First Search)

### 6.1.1 Procura no grafo

A função *PercorrerGrafoDFS* tem como objetivo percorrer o grafo utilizando o algoritmo de *busca em profundidade* (DFS).

Para isso, percorre a lista ligada de vértices do grafo e, sempre que encontra um vértice que ainda não foi visitado, chama a função *PercorrerArestasDFS* para iniciar a exploração a partir desse vértice.

```
bool PercorrerArestasDFS(Vertice* vertice)
{
    if (vertice->visitado == true)
    {
        return false;
    }
    else
    {
        vertice->visitado = true;

        Aresta* auxAresta = vertice->primAresta;
        MostrarCaminho(vertice);
        while (auxAresta != NULL)
        {
            Vertice* auxVertice = auxAresta->adjVertice;
            if (auxVertice->visitado != true)
            {
                PercorrerArestasDFS(auxAresta->adjVertice);
            }
            auxAresta = auxAresta->proxAresta;
        }
    }
    return true;
}
```

***Figura 20 - PercorrerArestasDFS***

A função *PercorrerArestasDFS* implementa a busca em profundidade (DFS) a partir de um vértice específico do grafo. Inicialmente, verifica se o vértice já foi visitado. Se sim, retorna false, interrompendo a chamada recursiva.

Caso contrário, marca o vértice como visitado e chama a função MostrarCaminho, que imprime caminho percorrido. De seguida, percorre todas as arestas ligadas a esse vértice usando uma lista de adjacência.

```
bool PercorrerGrafoDFS(Grafo* grafo)
{
    Vertice* auxVertice = grafo->primVertice;

    while (auxVertice != NULL)
    {
        if (auxVertice->visitado != true)
        {
            PercorrerArestasDFS(auxVertice);
        }
        auxVertice = auxVertice->proxVertice;
    }
    return true;
}
```

***Figura 321 - PercorrerGrafoDFS***

### 6.1.2 Procura em Vertice

A função *PercorrerVerticeDFS* realiza uma busca em profundidade (DFS) entre dois vértices: um de partida (*primVertice*) e um de destino (*segVertice*), com o objetivo de determinar se existe um caminho entre eles.

Primeiro, verifica se o vértice de partida já foi visitado. Se tiver sido, retorna false de imediato. Caso contrário, marca-o como visitado e chama a função *MostrarCaminho*, que imprime o vértice visitado.

Em seguida, verifica se o vértice atual tem a mesma posição que o vértice de destino. Se sim, retorna true, indicando que o caminho foi encontrado.

Caso contrário, percorre todas as arestas ligadas ao vértice atual. Para cada aresta, obtém o vértice adjacente (*proxVertice*) e, se este ainda não tiver sido visitado, chama recursivamente a própria função para continuar a procura a partir desse vértice.

```
bool PercorrerVerticeDFS(Vertice* primVertice, Vertice* segVertice)
{
    if (primVertice->visitado != false)
    {
        return false;
    }
    else
    {
        primVertice->visitado = true;
        MostrarCaminho(primVertice);

        if (primVertice->dados.posicao[0] == segVertice->dados.posicao[0] && primVertice->dados.posicao[1] == segVertice->dados.posicao[1])
        {
            return true;
        }

        Aresta* auxAresta = primVertice->primAresta;
        while (auxAresta != NULL)
        {
            Vertice* proxVertice = auxAresta->adjVertice;

            if (proxVertice->visitado != true)
            {
                PercorrerVerticeDFS(proxVertice, segVertice);
                return true;
            }
            auxAresta = auxAresta->proxAresta;
        }
        return false;
    }
}
```

***Figura 22 - PercorrerVerticeDFS***

## 6.2 Algoritmo BFS (Breath First Search)

### 6.2.1 Procura por Grafo

A função *PercorrerGrafoBFS* realiza uma busca em largura (BFS) em todo o grafo, garantindo que todos os vértices são percorridos, mesmo que o grafo não seja totalmente conectado (ou seja, tenha componentes isolados).

A função percorre a lista de vértices do grafo usando o ponteiro auxVertice. Para cada vértice que ainda não foi visitado (visitado != true), invoca a função PercorrerVerticeBFS, responsável por realizar o algoritmo BFS a partir desse ponto. O segundo parâmetro dessa função é

NULL, o que sugere que, nesta versão, não há um vértice de destino (possivelmente usado noutras variações da função).

Este método garante que todos os vértices e todas as suas ligações são analisados. No final do processo, a função retorna true, indicando que o percurso em largura foi concluído com sucesso.

```
bool PercorrerGrafoBFS(Grafo* grafo)
{
    Vertice* auxVertice = grafo->primVertice;

    while (auxVertice != NULL)
    {
        if (auxVertice->visitado != true)
        {
            PercorrerVerticeBFS(auxVertice, NULL);
        }
        auxVertice = auxVertice->proxVertice;
    }

    return true;
}
```

***Figura 23 - PercorrerGrafoBFS***

### 6.2.2 Pesquisa por Vértice

A função *PercorrerVerticeBFS* implementa o algoritmo de busca em largura (BFS) a partir de um vértice de origem, com a possibilidade de parar caso atinja um vértice de destino. BFS é realizada usando uma estrutura de fila (Espera), típica desta abordagem.

Inicialmente, o vértice de origem é marcado como visitado e colocado na fila. Em seguida, enquanto existirem elementos na fila, o vértice na frente é removido e comparado com o destino. Se a posição do vértice atual for igual à do destino, a função termina imediatamente, retornando true.

```
bool PercorrerVerticeBFS(Vertice* origem, Vertice* destino)
{
    origem->visitado = true;

    Espera* queue = (Espera*)malloc(sizeof(Espera));
    queue->vertice = origem;
    queue->proxVertice = NULL;

    while (queue != NULL)
    {
        Vertice* atual = queue->vertice;
        Espera* remover = queue;
        queue = queue->proxVertice;
        free(remover);

        MostrarCaminho(atual);

        if (destino && atual->dados.posicao[0] == destino->dados.posicao[0] && atual->dados.posicao[1] == destino->dados.posicao[1])
        {
            return true;
        }
        else
        {
            queue = PercorrerArestasBFS(atual->primAresta, queue);
        }
    }
    return false;
}
```

***Figura 24 - PercorrerVerticeBFS***

# 7. Conclusão

A fase 2 do projeto de Estruturas de Dados Avançadas representou uma evolução significativa em relação à fase anterior, tanto ao nível da complexidade da estrutura de dados utilizada como das funcionalidades implementadas. A transição de uma abordagem baseada em listas ligadas para a utilização de grafos permitiu explorar um novo nível de abstração e eficiência na representação de ligações entre antenas com a mesma frequência.

Foi possível aplicar com sucesso conceitos fundamentais da teoria dos grafos, como a definição de vértices e arestas, bem como algoritmos de procura em profundidade (DFS) e em largura (BFS). Estas técnicas foram essenciais para permitir a navegação, análise e deteção de padrões dentro do grafo, como caminhos possíveis entre antenas e a identificação de interações entre frequências distintas.

Durante o desenvolvimento, foi necessário repensar e corrigir decisões feitas na fase anterior, como o uso desnecessário de estruturas redundantes e erros na gestão de posições nefastas. Estas correções não só melhoraram a estrutura interna do código, como também permitiram consolidar aprendizagens importantes sobre eficiência, modularidade e clareza na programação.

A implementação de funcionalidades complementares, como o armazenamento e carregamento de grafos a partir de ficheiros binários, reforçou o domínio da manipulação de memória dinâmica e da persistência de dados em C.

Em suma, este projeto permitiu aplicar de forma prática os conhecimentos adquiridos ao longo da unidade curricular, promovendo uma maior compreensão sobre estruturas dinâmicas e algoritmos de grafos.