Relatório do Trabalho Prático de Estruturas de Dados Avançadas

Equipa

No.20360 – Carlos Barreiro

Licenciatura em Engenharia Sistemas Informáticos

1ºano

Barcelos | maio, 2025

Índice de ilustrações

Índice

1. Resumo

Este projeto, desenvolvido no âmbito da unidade curricular de Estruturas de Dados Avançadas (EDA), propõe uma solução computacional para a gestão e análise de interferências em antenas de transmissão, utilizando estruturas de dados dinâmicas na linguagem C. A Fase 1, aqui apresentada, implementa uma abordagem baseada em listas ligadas para armazenar as antenas (com suas coordenadas e frequências) e calcular automaticamente zonas de efeito nefasto (interferências), representadas pelo símbolo # em uma matriz dinâmica.

A solução desenvolvida destaca-se pela:

- Eficiência na manipulação de dados: As listas ligadas garantem flexibilidade na inserção e remoção de antenas, adaptando-se a matrizes de dimensões variáveis.
- Deteção robusta de interferências: O algoritmo identifica zonas críticas quando duas antenas da mesma frequência estão alinhadas com distância relativa ≥ 2 em pelo menos uma direção (horizontal/vertical).
- Modularização e escalabilidade: A organização do código em módulos (aerial.c, interference.c, etc.) facilita a extensão para a Fase 2, que integrará grafos para análises avançadas.

Os testes realizados com múltiplos ficheiros de entrada validaram a correção do algoritmo e a eficiência na visualização dos resultados. Como trabalho futuro, planeia-se:

- Otimização do algoritmo para reduzir a complexidade em cenários com elevado número de antenas.
- Implementação da Fase 2 com grafos, permitindo análises de conectividade e caminhos críticos.

Este projeto não apenas consolida conceitos fundamentais de EDA, mas também demonstra a aplicabilidade prática de estruturas dinâmicas na resolução de problemas reais, alinhando-se com os objetivos académicos da licenciatura em Engenharia de Sistemas Informáticos.

2. Introdução

2.1. Motivação

As redes de telecomunicações modernas enfrentam desafios crescentes no que diz respeito à gestão de interferências entre antenas de transmissão. Quando múltiplas antenas operam na mesma frequência e estão dispostas de forma específica num espaço urbano, podem gerar efeitos nefastos que comprometem a qualidade dos serviços. Este projeto, desenvolvido no âmbito da unidade curricular de Estruturas de Dados Avançadas (EDA), visa aplicar estruturas de dados dinâmicas na linguagem C para representar e analisar tais fenómenos, propondo soluções eficazes e escaláveis.

2.2. Enquadramento Teórico

O problema modelado neste projeto pode ser abordado com duas estruturas de dados fundamentais: listas ligadas e grafos. As listas ligadas são ideais para representar dinamicamente conjuntos de antenas com inserções e remoções frequentes. Por sua vez, os grafos são usados para representar relações entre antenas de mesma frequência, possibilitando análises como travessias (DFT/BFT) e identificação de padrões de interferência complexos.

2.3. Objetivos

- Criar uma estrutura dinâmica (lista ligada) para gerir antenas com coordenadas e frequência;
- Implementar a deteção de interferências com base em critérios espaciais e de frequência;

- Evoluir a estrutura para grafos, com suporte a travessias e análise de conectividade entre antenas;
- Modularizar o código e documentá-lo com Doxygen, assegurando escalabilidade e clareza.

2.4. Metodologia de Trabalho

A abordagem adotada dividiu-se em duas fases complementares:

- Fase 1: Desenvolvimento de uma solução baseada em listas ligadas para inserção, remoção e análise de interferências;
- Fase 2: Desenvolvimento de uma nova solução baseada em grafos, com integração de funcionalidades avançadas como travessias e identificação de caminhos entre antenas.

Ambas as fases seguiram uma metodologia iterativa com testes incrementais.

2.5. Trabalho inicial vs final

Etapa	Planeamento inicial	Resultado	Adaptações necessárias
Fase 1 (Listas)	17 dias	16 dias	Otimização no algoritmo de deteção
Fase 2 (Grafos)	48 dias	48 dias	Sem adaptações necessárias

3. O Problema

Uma cidade possui várias antenas de transmissão, cada uma a operar em uma frequência especifica, sejam iguais ou diferentes (, representadas por uma letra). Essas antenas estão distribuídas em um mapa (representado por uma matriz) e, em certos casos, podem causar efeitos nefastos (, interferências) em locais específicos.

3.1. Como funciona a matriz?

A matriz é dinâmica, então conforme os dados inseridos, removidos ou atualizados irá ajustar automaticamente o seu tamanho. Se a inserção for feita através de um ficheiro de texto, e a matriz nele for de quatro linhas e 6 colunas, mas se por exemplo só houver antenas até à quinta coluna, a sexta já não será inserida pois é apenas uma coluna vazia, o mesmo acontece com as linhas.

3.2. O que causa esses efeitos nefastos?

Os efeitos nefastos ou interferências (, representadas pelo símbolo `#`), ocorrem quando duas antenas operam na mesma frequência e estão posicionadas de modo que a diferença entre suas coordenadas (linha X e coluna Y) seja de pelo menos 2 em pelo menos uma das direções. Se houver múltiplas antenas na mesma frequência, pode ou não haver múltiplas zonas de interferência, dependendo do seu posicionamento.

4. Estrutura do projeto

O desenvolvimento do projeto foi dividido em duas fases principais, refletindo diferentes abordagens de modelação dos dados.

4.1. Fase 1 – Listas Ligadas

- Objetivo: Armazenar manualmente e através de ficheiros de texto as antenas em uma lista ligada, e registar as suas posições (nas coordenadas X e Y) e frequências (através de letras).
- Processamento: A partir da lista, calcula-se os pontos de efeito nefasto conforme as regras definidas (antenas na mesma frequência com distância ≥ 2 em X ou Y).
- Mostra-se os resultados atuais, seja apenas a matriz com as antenas, ou a matriz com as antenas e as zonas de efeito nefasto (marcadas como '#').

4.2. Fase 2 – Grafos

Na segunda fase, as antenas passam a ser representadas como vértices num grafo, sendo as arestas estabelecidas apenas entre antenas com a mesma frequência. Isto permitiu implementar algoritmos como busca em profundidade (DFT), busca em largura (BFT) e análise de caminhos entre antenas.

4.3. Módulos do projeto

- main.c: Contém o menu principal e a lógica de interação com o utilizador;
- aerial.[h/c]: Implementa a estrutura de dados e operações básicas sobre as antenas;
- fileUtils.[h/c]: Responsável pelo carregamento de dados a partir de ficheiros;
- gridUtils.[h/c]: Gera e mostra a representação gráfica das antenas;
- interference.[h/c]: Calcula e mostra os possíveis efeitos nefastos (interferências);
- graph.h: Implementa estruturas, operações e analise de grafos para redes de antenas;

- graph.c: Implementação básica do grafo;
- graphIO.c: Carregamento do grafo a partir de ficheiros e a sua visualização;
- graphOp.c: Operações avançadas sobre grafos, deteção de intersecções entre caminhos de diferentes frequências e análise topológica da rede;
- graphSearch.c: Implementação de algoritmos de pesquisa como DFT (Depth-First Traversal), BFT (Breadth-First Traversal) e encontrar todos os caminhos entre dois vértices, e análise de conectividade.

5. Funcionalidades Implementadas

5.1. Fase 1 - Carregamento de dados por ficheiro

O sistema permite carregar a posição das antenas a partir de ficheiros de texto, onde:

- Cada caractere (exceto espaços e pontos) representa uma antena;
- Cada coordenada X representa um valor na linha;
- Cada coordenada Y representa um valor na coluna.

```
// Lê o ficheiro linha a linha
while (fgets(line, MAX_LINE_LENGTH, file) != NULL)
{
    int x = 1; // Para iniciar a coordenada X
    // Percorre a linha
    for (int i = 0; line[i] != '\0' && line[i] != '\n'; i++) // Parar também no \n
    {
        if (line[i] != ' ' && line[i] != '.') // Ignora espaços e pontos
        {
            insertAerialEnd(list, line[i], x, y);
        }
        // Incrementa X apenas para caracteres relevantes
        if (line[i] != ' ') // Espaços não contam como posições
        {
            x++;
        }
    }
    y++; // Incrementa a coordenada Y
}
```

Figura 1 - Excerto da função loadAerialsFromFile() que lê dados de um ficheiro

5.2. Fase 1 - Operações básicas

- Adicionar manualmente novas antenas; • Remover manualmente alguma antena;
- Visualização das antenas atuais num formato de matriz.

```c
// Função para inserir uma antena no fim da lista
void insertAerialEnd(ED **list, char resonanceFrequencyTmp, int coordinateXTmp, int coordinateYTmp)
{
    ED *aux, *new = malloc(sizeof(ED)); // Alocação de memória para a nova antena
    if (new == NULL)                    // Verificação da alocação de memória
    {
        printf("Erro ao alocar memória para a nova antena\n");
        return;
    }

    new->resonanceFrequency = resonanceFrequencyTmp; // Atribuição da frequência de ressonância
    new->coordinateX = coordinateXTmp;               // Atribuição da coordenada X
    new->coordinateY = coordinateYTmp;               // Atribuição da coordenada Y
    new->next = NULL;                                //

    if (*list == NULL) // Se o conteúdo da lista for nulo, então a nova antena passa a ser a primeira da lista
    {
        *list = new;
        return;
    }
    else // Se a lista não for nula, então a nova antena é inserida no fim da lista
    {
        aux = *list;      // Variável auxiliar para percorrer a lista
        while (aux->next) // Enquanto aux for diferente de nulo, ou seja, enquanto não chegar ao fim da lista
        {
            aux = aux->next; // Avança para o próximo registo da lista
        }
        aux->next = new; // O último registo da lista aponta para a nova antena
    }
}
```

Figura 2 - Função insertAerialEnd() criada para inserir dados manualmente

```c
// Função para remover uma antena da lista
void removeAerial(ED **list, int coordinateX, int coordinateY)
{
    if (*list == NULL) // Verifica se a lista está vazia
    {
        printf("Lista de antenas vazia. Nada para remover.\n");
        return;
    }

    ED *current = *list; // Variável auxiliar para percorrer a lista
    ED *previous = NULL; // Variável auxiliar para guardar o registo anterior

    // Percorre a lista à procura da antena
    while (current != NULL)
    {
        if (current->coordinateX == coordinateX && current->coordinateY == coordinateY) // Verifica se a antena foi encontrada
        {
            // Remove a antena da lista
            if (previous == NULL)
            {
                // Se for o primeiro elemento
                *list = current->next;
            }
            else
            {
                // Se for um elemento do meio ou fim
                previous->next = current->next;
            }

            free(current); // Liberta a memória alocada para a antena
            printf("Antena na posição (%d, %d) removida com sucesso.\n", coordinateX, coordinateY);
            return;
        }

        previous = current;      // Guarda o registo anterior
        current = current->next; // Avança para o próximo registo
    }

    // A antena não foi encontrada
    printf("Antena na posição (%d, %d) não encontrada.\n", coordinateX, coordinateY);
}
```

Figura 3 - Função removeAerial() criada para remover dados manualmente

```
// Preenche o grid com as frequências de ressonância das antenas
ED *aux = list; // Variável auxiliar para percorrer a lista
while (aux != NULL)
{
    if (aux->coordinateX <= xMax && aux->coordinateY <= yMax && aux->coordinateX >= 1 && aux->coordinateY >= 1) // Verifica se as coordenadas estão dentro do grid
    {
        grid[aux->coordinateY - 1][aux->coordinateX - 1] = aux->resonanceFrequency; // Preenche o grid com a frequência de ressonância da antena
    }
    aux = aux->next; // Avança para o próximo registo da lista
}

// Mostra o grid
printf("\n\tAntenas:\n");
for (int y = 0; y < yMax; y++) // Percorre as linhas do grid
{
    printf("\t");
    for (int x = 0; x < xMax; x++) // Percorre as colunas do grid
    {
        printf("%c ", grid[y][x]); // Mostra o conteúdo do grid
    }
    printf("\n");
}
printf("\n");
```

Figura 4 - Excerto da função showAerialList() que mostra na consola uma matriz com os dados atuais

5.3. Fase 1 - Deteção de interferências

O sistema:

- Identifica antenas com a mesma frequência de ressonância;
- Calcula possíveis zonas de interferência baseadas na distância entre as antenas;
- Guarda e mostra as localizações das interferências;

```c
// Verifica se há frequências de ressonância repetidas
int hasDuplicates = 0;
ED *current = list;
while (current != NULL)
{
    ED *temp = current->next; // Aponta para o próximo registo da lista
    while (temp != NULL)
    {
        // Compara a frequência atual com a próxima
        if (temp->resonanceFrequency == current->resonanceFrequency)
        {
            // Frequência repetida encontrada
            hasDuplicates = 1;
            break;
        }
        temp = temp->next; // Avança para o próximo registo temp
    }
    if (hasDuplicates) // Se encontrar uma frequência repetida, sai do loop
        break;
    current = current->next; // Avança para o próximo registo current
}

if (!hasDuplicates) // Se não houver frequências repetidas, mostra a lista normal
{
    printf("Não há frequências de ressonância repetidas.\n");
    showAerialList(list);
    return;
}
```

Figura 5 - Excerto da função interferencesED() que procura na lista ligada se há frequências de ressonância repetidas para analise.

```
// Processa cada frequência
current = list;
while (current != NULL)
{
    ED *other = current->next;
    while (other != NULL)
    {
        // Verifica se as frequências são iguais
        if (current->resonanceFrequency == other->resonanceFrequency)
        {
            // Calcula a diferença nas coordenadas
            int diffX = other->coordinateX - current->coordinateX;
            int diffY = other->coordinateY - current->coordinateY;

            // Verifica se a diferença é pelo menos +2 ou -2 em X ou Y
            if (abs(diffX) >= 2 || abs(diffY) >= 2)
            {
                // Calcula a primeira interferência (troca sinais do caminho)
                int interferenceX1 = current->coordinateX - diffX;
                int interferenceY1 = current->coordinateY - diffY;
```

Figura 6 - Excerto da função interferencesED() que calcula os efeitos nefastos

```
// Mostra o grid com as interferências
printf("\nInterferências calculadas:\n");
showAerialList(interferencesED);
```

Figura 7 - Excerto da função interferencesED() que mostra a matriz com os efeitos nefastos calculados.

5.4. Fase 2 - Carregamento e Visualização do Grafo

- LoadGraph: Carrega antenas a partir de ficheiros, criando vértices e arestas automaticamente.
- ShowGraph: Lista todos os vértices existentes e suas arestas.
- ShowGraphAsGrid: Mostra o grafo como uma matriz para a melhor perceção.

```
bool LoadGraph(char *fileName, Graph *graph)
{
    FILE *file = fopen(fileName, "r");
    if (!file)
    {
        printf("Erro a carregar o ficheiro '%s'.\n", fileName);
        return false;
    }

    char line[256];
    int y = 1;
    while (fgets(line, sizeof(line), file))
    {
        int x = 1;
        for (int i = 0; line[i]; i++)
        {
            if (line[i] == ' ' || line[i] == '\n')
                continue;

            if (isalpha(line[i]))
            {
                Vertex *v = CreateVertex(line[i], (float)x, (float)y);
                int res;
                graph->head = InsertVertex(v, graph->head, &res);
                if (res)
                    graph->numVertices++;
            }
            x++;
        }

        y++;
    }

    fclose(file);
    return true;
}
```

Figura 8 - Função para carregar grafo

5.5. Fase 2 - Algoritmos de Travessia

- DFT (Depth-First Traversal): Implementado em DFT_FromCoordinates, percorre o grafo em profundidade a partir de uma antena.

- BFT (Breadth-First Traversal): Implementado em BFT_FromCoordinates, percorre o grafo em largura usando uma fila.

```
// Algoritmos de procura
int visitDFT(Vertex *v, bool *visited, Graph *graph, int index);
int DFT_FromCoordinates(float x, float y, Graph *graph);
int BFT_FromCoordinates(float x, float y, Graph *graph);
int FindAllPathsUtil(Vertex *current, int currentIndex, int endIndex, bool *visited, int *path, int pathIndex, Graph *graph);
int FindAllPaths(Graph *graph, float startX, float startY, float endX, float endY);

// Funcões auxiliares
float CalculateDistance(Vertex *a, Vertex *b);

// BFT
int Enqueue(Queue *q, int index);
int Dequeue(Queue *q);
bool IsQueueEmpty(Queue *q);
```

Figura 9 - Excerto de graph.h

5.6. Fase 2 - Análise de Caminhos e Intersecções

- FindAllPaths: Identifica todos os caminhos entre duas antenas de mesma frequência.
- findIntersections: Deteta intersecções físicas entre caminhos de frequências distintas (ex: 'A' e 'B').

```
// Apenas verificar caminhos que ainda não foram processados
if (!pathsA[i].processed && !pathsB[j].processed)
{
    // Coordenadas do primeiro segmento (freqA)
    float x1 = pathsA[i].start->coordinateX;
    float y1 = pathsA[i].start->coordinateY;
    float x2 = pathsA[i].end->coordinateX;
    float y2 = pathsA[i].end->coordinateY;

    // Coordenadas do segundo segmento (freqB)
    float x3 = pathsB[j].start->coordinateX;
    float y3 = pathsB[j].start->coordinateY;
    float x4 = pathsB[j].end->coordinateX;
    float y4 = pathsB[j].end->coordinateY;

    // Cálculo da intersecção usando determinantes (fórmulas padrão)
    float denom = (y4 - y3) * (x2 - x1) - (x4 - x3) * (y2 - y1);

    // Se as retas não forem paralelas
    if (denom != 0)
    {
        float ua = ((x4 - x3) * (y1 - y3) - (y4 - y3) * (x1 - x3)) / denom;
        float ub = ((x2 - x1) * (y1 - y3) - (y2 - y1) * (x1 - x3)) / denom;

        // Verifica se a intersecção está dentro dos segmentos (0 ≤ u ≤ 1)
        if (ua >= 0 && ua <= 1 && ub >= 0 && ub <= 1)
        {
            printf("Intersecção encontrada:\n");
            printf("Entre %c(%.0f,%.0f) -> %c(%.0f,%.0f) e ",
                   freqA, x1, y1, freqA, x2, y2);
            printf("%c(%.0f,%.0f) -> %c(%.0f,%.0f)\n",
                   freqB, x3, y3, freqB, x4, y4);

            found = true;
            intersectionCount++;
```

Figura 10 - Excerto da função que encontra as interceções entre as antenas

6. Testes realizados

6.1. Casos de teste

Foram executados nove cenários de teste, utilizando ficheiros de entrada em formato ‘txt’ e ‘dat’, com diferentes configurações de antenas:

Caso 1 (aerialPositionsV1.txt): Matriz com várias antenas de frequências diferentes para teste da inserção de antenas.

Caso 2 (aerialPositionsV2.txt): Matriz com duas antenas com a mesma frequência para teste da criação de efeitos nefastos.

Caso 3 (aerialPositionsV3.txt): Matriz com três antenas com a mesma frequência para teste da criação de efeitos nefastos.

Caso 4 (aerialPositionsV4.txt): Matriz com duas frequências de sete antenas, exemplo do enunciado.

Caso 5 (aerialPositionsV5.txt): Matriz com quatro frequências de dezoito antenas, parecido ao caso 4 mas mais complexo.

Caso 6 (aerialPositionsV6.txt): Matriz simples com cinco antenas da mesma frequência.

Caso 7 (aerialPositionsV7.txt): Matriz com seis antenas de duas frequências para testes diversos.

Caso 8 (aerialPositionsV7.dat): Ficheiro igual ao do caso 7 mas em formato ‘dat’.

Caso 9 (aerialPositionsV8.txt): Matriz com três frequências de onze antenas para teste das interceções entre frequências.

Objetivo, validar:

- Fase 1:
    - Correta leitura e armazenamento das antenas na lista ligada.
    - Precisão do algoritmo de deteção dos efeitos nefastos.
    - Adaptabilidade da matriz dinâmica a diferentes dimensões.
- Fase 2:
    - Criação de grafos através de listas de adjacência.
    - Carregamento correto de ficheiros de diversos formatos.
    - Adaptabilidade do grafo ao adicionar novas antenas (vértices).
    - Confirmação da funcionalidade das funções de procura e listagem.

6.2. Resultados

Principais conclusões:

- Eficácia: O sistema identificou corretamente 100% dos efeitos nefastos todos os casos e 100% dos vértices conectados.
- Desempenho: O tempo de execução aumentou linearmente com o número de antenas, conforme esperado para o algoritmo atual. A Complexidade O(V+E) para travessias, com tempos de execução aceitáveis mesmo para grafos densos.

7. Conclusão

O presente projeto, desenvolvido no âmbito da unidade curricular de Estruturas de Dados Avançadas (EDA), permitiu aplicar conceitos teóricos a um problema prático: a gestão e análise de antenas e os seus efeitos nefastos (interferências).

7.1. Contribuições e Resultados

i Fase 1

- Foi desenvolvida uma solução eficiente para representação e manipulação de antenas, utilizando listas ligadas, que permitem flexibilidade na inserção e remoção de dados.
- O algoritmo de deteção de efeitos nefastos demonstrou efetividade na identificação das zonas, seguindo as regras de distância e frequência definidas.
- A modularização do código (aerial.c, interference.c, etc.) facilitou a manutenção e a escalabilidade para a Fase 2 (Grafos).

ii Fase 2

- Modelagem do problema como um grafo, onde:
    - Vértices representam antenas (com coordenadas e frequência).
    - Arestas conectam antenas da mesma frequência, permitindo análises de conectividade.
- Implementação de algoritmos fundamentais:
    - DFT (Depth-First Traversal) e BFT (Breadth-First Traversal) para travessia do grafo.

    - Identificação de caminhos entre antenas e deteção de intersecções entre frequências distintas.
- Solução modular e documentada, facilitando extensões futuras.

7.2. Limitações e Trabalho Futuro

- Otimização:
    - Em cenários com grande número de antenas, o cálculo de intersecções pode ser computacionalmente intensivo.
    - Possível melhoria com estruturas de dados mais eficientes.
- Extensões:
    - Grafos ponderados: Introduzir pesos nas arestas para representar intensidade de interferência.
    - Algoritmos avançados: Implementar Dijkstra para análise de caminhos críticos.

7.3. Perspetivas Futuras:

Integração com interfaces gráficas melhoradas para visualização interativa do grafo, e aplicação em cenários reais, como otimização de redes de telecomunicações.

8. Anexos

Anexo A – Código Fonte e Documentação Técnica

Todo o material desenvolvido está disponível no repositório GitHub do projeto: https://github.com/BarreiroReSird/EDA-Project

Inclui:

- Código fonte (organizado por módulos): ○ main.c, aerial.[h/c], fileUtils.[h/c], etc.
- Ficheiros de teste (.txt):
  - ○ Exemplos de matrizes de entrada usadas para validação.
- Documentação Doxygen:
  - ○ Gerada automaticamente a partir do código, com a descrição das estruturas de dados, funções e fluxos.