Instituto Politécnico do Cávado e do Ave

Escola Superior de Tecnologia

Relatório do Projeto

Estruturas de Dados Avançadas

Aluno: Vitor Silveira Alonso de Rezende
Número: 31521
Curso: LESI
Unidade Curricular: Estruturas de Dados Avançadas
Docente: Luis Gonzaga Martins Ferreira

Maio de 2025

Conteúdo

Conteúdo **1**

1 Introdução **3**

2 Contextualização **4**

3 Objetivos **5**

4 Estruturas de Dados **6**
4.1 Vector2 . . . 6
4.2 Node . . . 6
4.3 Graph . . . 6
4.4 Vertex . . . 6
4.5 Edge . . . 7
4.6 Element . . . 7
4.7 Path . . . 7
4.8 Queue e Stack (implementações auxiliares) . . . 7

5 Etapa 1 – Leitura, Armazenamento e Deteção de Interferências **8**
5.1 Objetivo da Etapa . . . 8
5.2 Leitura do Ficheiro e Construção da Lista . . . 8
5.3 Deteção de Efeitos Nefastos . . . 8
5.4 Armazenamento e Exportação dos Resultados . . . 8
5.5 Visualização . . . 9
5.6 Desafios e Soluções . . . 9
5.7 Funções Principais da Etapa 1 . . . 9
5.8 Estruturas Utilizadas . . . 9

6 Etapa 2 – Representação em Grafo e Algoritmos de Busca **10**
6.1 Objetivo da Etapa . . . 10
6.2 Conversão da Lista para Grafo . . . 10
6.3 Estruturas da Etapa 2 . . . 10
6.3.1 Edge . . . 10
6.3.2 Vertex . . . 10
6.3.3 Graph . . . 10
6.3.4 Element e Path . . . 11
6.4 Algoritmos de Travessia . . . 11
6.4.1 Breadth-First Search (BFS) . . . 11
6.4.2 Depth-First Search (DFS) . . . 11
6.5 Percursos e Ressonância . . . 11
6.6 Visualização e Interface . . . 12
6.7 Armazenamento e Log . . . 12
6.8 Desafios e Soluções . . . 12
6.9 Algoritmos do Grafo . . . 12

7 Estrutura de Diretórios do Projeto **13**

8 Desafios Encontrados **14**
8.1 Gerenciamento de Memória 14
8.2 Evitar Ciclos Inesperados 14

9 Trabalhos Futuros **14**

10 Conclusão Final **15**

11 Bibliografia **16**

1 Introdução

Este projeto faz parte da avaliação individual da Unidade Curricular de Estruturas de Dados Avançadas (EDA). O objetivo é reforçar e aplicar conhecimentos adquiridos, especialmente no uso e manipulação de estruturas de dados dinâmicas na linguagem C. Além disso, a implementação exige modularização, armazenamento de dados em ficheiros e documentação utilizando Doxygen.

2 Contextualização

O projeto modela uma cidade com várias antenas, cada uma sintonizada numa frequência específica (representada por um caractere). A matriz a seguir ilustra um exemplo de disposição das antenas:

```
............
........0...
.....0......
.......0....
....0.......
......A.....
............
............
........A...
.........A..
............
............
```

Antenas de mesma frequência podem gerar efeitos nefastos (#), que aparecem em locais específicos dependendo da distância entre as antenas.

3 Objetivos

O projeto deve implementar as seguintes funcionalidades:

- Definição de uma estrutura de dados dinâmica (lista ligada) para armazenar as antenas e suas posições.
- Leitura dos dados de um ficheiro e armazenamento na estrutura de dados.
- Manipulação da lista: inserção e remoção de antenas.
- Identificação automática de locais com efeito nefasto e armazenamento desses locais.
- Exibição da matriz e dos dados em formato tabular no terminal.
- Identificar e modelar a distribuição de antenas numa matriz bidimensional.
- Detetar interferências causadas por antenas com a mesma frequência.
- Representar o problema como grafo e aplicar algoritmos clássicos de busca (DFS e BFS).
- Permitir análise de caminhos e componentes conexas entre antenas.

4 Estruturas de Dados

Nesta seção, são apresentadas as principais estruturas utilizadas ao longo do projeto. Elas foram desenhadas para permitir uma manipulação eficiente da matriz de antenas, detecção de interferências, e posteriormente, a conversão para grafo e aplicação de algoritmos de busca.

4.1 Vector2

```c
typedef struct Vector2 {
    int x;
    int y;
} Vector2;
```

Estrutura auxiliar utilizada para representar coordenadas bidimensionais (x, y) na matriz. É reutilizada em várias outras estruturas, como em vértices do grafo ou nós da lista.

4.2 Node

```c
typedef struct Node {
    Vector2 pos;
    char value;
    struct Node* next;
} Node;
```

Representa um elemento da lista ligada que armazena uma antena. Contém a posição na matriz, o valor (frequência ou marcador especial) e um ponteiro para o próximo nó da lista. É a estrutura central da Etapa 1.

4.3 Graph

```c
typedef struct Graph {
    int count;
    Vertex* vertices;
} Graph;
```

Estrutura principal que representa o grafo. Armazena o número total de vértices e um ponteiro para a lista encadeada de vértices. Utilizada na Etapa 2 para representar antenas válidas e suas conexões.

4.4 Vertex

```c
typedef struct Vertex {
    int id;
    Vector2 pos;
    char value;
    Edge* edges;
    int seen;
    struct Vertex* next;
} Vertex;
```

Cada antena válida é convertida em um vértice. Armazena um identificador único (`id`), a posição na matriz, a frequência (`value`), uma lista de arestas (`edges`), um campo auxiliar `seen` (para algoritmos de busca) e o ponteiro para o próximo vértice.

4.5 Edge

```
typedef struct Edge {
    float weight;
    struct Vertex* dest;
    struct Edge* next;
} Edge;
```

Representa uma aresta que conecta dois vértices. Contém o peso (geralmente a distância entre os vértices), o destino da ligação e o ponteiro para a próxima aresta. É utilizada para construir a lista de adjacência do grafo.

4.6 Element

```
typedef struct Element {
    Vertex* item;
    struct Element* next;
} Element;
```

Elemento de uma lista auxiliar usada nos algoritmos de travessia de grafos (BFS e DFS). Armazena um ponteiro para o vértice visitado e o próximo elemento da lista.

4.7 Path

```
typedef struct Path {
    Element* first;
    int max;
} Path;
```

Representa um caminho no grafo, usado para armazenar sequências de vértices visitados durante as buscas. O campo `max` indica o tamanho máximo do percurso.

4.8 Queue e Stack (implementações auxiliares)

Embora não explicitamente mostradas no código principal, as travessias em largura (BFS) e profundidade (DFS) utilizam estruturas de `fila` e `pilha`, respectivamente, implementadas com listas encadeadas ou chamadas recursivas.

- **Queue (Fila)**: usada na BFS para visitar vértices por camadas.
- **Stack (Pilha ou Recursão)**: usada na DFS para visitar vértices em profundidade.

Essas estruturas são implementadas de forma modular, reutilizando `Element` para armazenar os vértices a serem processados.

5 Etapa 1 – Leitura, Armazenamento e Deteção de Interferências

5.1 Objetivo da Etapa

O objetivo principal da primeira etapa foi representar e organizar a informação das antenas presentes numa matriz bidimensional, detetando os efeitos nefastos causados pela proximidade entre antenas com a mesma frequência. Esta etapa foca-se na leitura dos dados, criação da estrutura de dados inicial (lista ligada), e aplicação das regras para identificar interferências.

5.2 Leitura do Ficheiro e Construção da Lista

O sistema inicia com a leitura de um ficheiro de entrada que representa uma matriz, onde cada célula pode conter ou não uma antena. A função `ReadListFile` percorre o conteúdo do ficheiro e armazena, numa lista ligada, as posições e frequências das antenas válidas encontradas.

```
Node* ReadListFile(const char* filename);
```

Cada linha do ficheiro é interpretada como uma linha da matriz, e os caracteres são lidos como possíveis antenas. Caso uma célula contenha uma antena (representada por um caractere válido), é criado um nó com a sua posição `(x, y)` e frequência.

5.3 Deteção de Efeitos Nefastos

Após a construção da lista, é necessário verificar se existem efeitos nefastos entre antenas. Para isso, são implementadas duas funções:

- `NoiseCheck` – Percorre a lista e compara pares próximos de antenas com a mesma frequência, aplicando a regra da distância para sinalizar locais afetados.
- `NoiseCheckAlt` – Variante exaustiva que compara todos os pares possíveis da lista, assegurando uma cobertura completa das combinações.

```
Node *NoiseCheck(Node *st);
Node *NoiseCheckAlt(Node *st);
```

O critério utilizado considera que uma antena provoca um efeito nefasto se estiver a mais do dobro da distância de outra antena da mesma frequência, resultando numa marcação especial na matriz.

5.4 Armazenamento e Exportação dos Resultados

Com os dados processados e os efeitos nefastos identificados, a lista atualizada pode ser exportada novamente para um ficheiro através da função `SaveList`, permitindo ao utilizador observar as alterações resultantes da análise.

```
void SaveList(const char* filename, Node* st);
```

Este ficheiro de saída segue o mesmo formato do ficheiro de entrada, sendo fácil de interpretar e reutilizar para etapas futuras.

5.5 Visualização

O estado atual da matriz (com ou sem efeitos nefastos) pode ser visualizado diretamente no terminal através da função:

```
void DrawMatrix(Node *st);
```

Cada célula é desenhada com base nos nós da lista, permitindo ao utilizador ter uma representação visual clara da distribuição e interferência das antenas.

5.6 Desafios e Soluções

Durante esta etapa, surgiram diversos desafios importantes:

- Garantir a correta leitura do ficheiro e interpretação dos dados.
- Criar uma estrutura de lista eficiente e dinâmica para armazenar dados da matriz.
- Implementar regras de interferência que respeitem distâncias relativas entre antenas.
- Diferenciar visualmente células afetadas por interferência.

Estas dificuldades foram resolvidas com testes manuais, simulações de casos limites, e refatoração progressiva das funções, com foco em modularidade e clareza.

5.7 Funções Principais da Etapa 1

Abaixo está uma síntese das funções mais relevantes desta etapa:

- `ReadListFile` – Leitura da matriz do ficheiro.
- `SaveList` – Escrita da lista em ficheiro de saída.
- `NoiseCheck` e `NoiseCheckAlt` – Deteção de interferências.
- `DrawMatrix` – Visualização gráfica da matriz.

5.8 Estruturas Utilizadas

Para representar os dados nesta fase inicial, utiliza-se a estrutura de lista ligada:

```
typedef struct Node {
    int x, y;
    char value;
    struct Node *next;
} Node;
```

Esta estrutura permite armazenar dinamicamente as posições das antenas na matriz e associar facilmente efeitos nefastos mediante as análises realizadas.

6 Etapa 2 – Representação em Grafo e Algoritmos de Busca

6.1 Objetivo da Etapa

A segunda etapa do projeto visa evoluir a estrutura de dados, anteriormente baseada apenas em listas ligadas, para uma representação em grafo. A ideia principal é transformar as antenas válidas (sem interferência) em vértices de um grafo, permitindo assim aplicar algoritmos de travessia e análise como BFS (Breadth-First Search) e DFS (Depth-First Search). Estes algoritmos são úteis para identificar percursos entre antenas, componentes conectados e pares com ressonância.

6.2 Conversão da Lista para Grafo

A lista de nós criada na Etapa 1 é percorrida, e cada antena válida (não nefasto) é transformada num vértice. Vértices são ligados por arestas se estiverem a uma distância de 2 unidades e tiverem a mesma frequência.

```
bool CopyListToGraph(Node* st, Graph* gr);
```

Este processo cria conexões entre as antenas através da estrutura `Edge`, representando uma ligação entre antenas vizinhas da mesma frequência.

6.3 Estruturas da Etapa 2

6.3.1 Edge

```
typedef struct Edge {
    float weight;
    struct Vertex *dest;
    struct Edge *next;
} Edge;
```

Representa uma aresta, armazenando o peso (distância), destino e ligação para a próxima aresta.

6.3.2 Vertex

```
typedef struct Vertex {
    int id;
    Vector2 pos;
    char value;
    Edge *edges;
    int seen;
    struct Vertex *next;
} Vertex;
```

Cada antena torna-se um vértice do grafo. O campo `seen` é utilizado nos algoritmos de travessia para evitar ciclos.

6.3.3 Graph

```c
typedef struct Graph {
    int count;
    Vertex *vertices;
} Graph;
```

Estrutura principal do grafo, armazenando o número de vértices e a lista de vértices.

6.3.4 Element e Path

```c
typedef struct Element {
    Vertex *item;
    struct Element *next;
} Element;

typedef struct Path {
    Element *first;
    int max;
} Path;
```

Utilizadas para armazenar e organizar os percursos durante as travessias como BFS e DFS.

6.4 Algoritmos de Travessia

Foram implementados dois algoritmos clássicos de grafos:

6.4.1 Breadth-First Search (BFS)

```c
Element* GraphBFS(Graph* gr, Vertex* start);
```

Este algoritmo percorre o grafo em largura, utilizando uma fila para visitar os vértices vizinhos em camadas. É útil para encontrar o caminho mais curto (menor número de arestas).

6.4.2 Depth-First Search (DFS)

```c
Element* GraphDFS(Graph* gr, Vertex* start);
```

Explora o grafo em profundidade, utilizando uma pilha (ou recursão) para visitar vértices até o fim antes de recuar. Ideal para encontrar caminhos profundos e detectar ciclos.

6.5 Percursos e Ressonância

Após a travessia, os caminhos podem ser analisados para encontrar padrões de ressonância, como pares de antenas A-B seguidas de B-A. A função `ShowResonancePairs` identifica e exibe esses pares:

```c
void ShowResonancePairs(Element* list);
```

6.6 Visualização e Interface

Toda a interação com o utilizador foi projetada para terminal. Foram adicionadas opções no menu para converter a lista em grafo, realizar buscas, mostrar o grafo filtrado, e exibir os resultados de travessias.

```
void Menu(Node* st, Graph* gr);
void DrawMatrix(Node *st);
void ShowGraph(Graph *gr, char filter);
```

6.7 Armazenamento e Log

A etapa 2 também incorpora funcionalidades de registo em ficheiros de log, permitindo armazenar ações do utilizador e erros para futura análise.

```
void InitLog();
void Log(const char *format, ...);
```

6.8 Desafios e Soluções

Durante esta etapa, enfrentaram-se desafios como:

- Garantir que as ligações entre vértices não duplicassem arestas.
- Implementar BFS e DFS de forma modular sem causar vazamento de memória.
- Identificar corretamente a distância entre antenas usando a métrica Euclidiana.
- Separar corretamente os nós inválidos (nefastos) para evitar erros na travessia.

Todos os desafios foram superados com testes modulares, verificação de memória com ferramentas como `valgrind`, e uma estrutura bem dividida entre leitura, construção, visualização e travessia.

6.9 Algoritmos do Grafo

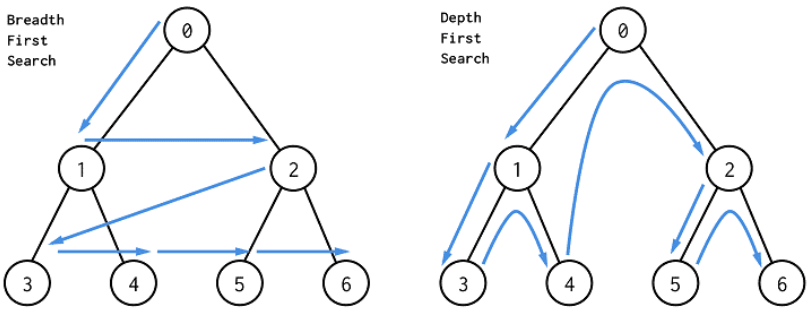

Figura 1: Representação gráfica dos alforitimos de busca em largura e profundidade.

7 Estrutura de Diretórios do Projeto

A estrutura de diretórios do projeto foi organizada para separar claramente os diferentes componentes do desenvolvimento, incluindo o código-fonte, documentação e configuração. Abaixo é apresentada a árvore de diretórios:

Árvore Geral

```
Tp_EDA/
  .vscode/              # Configurações do Visual Studio Code
  doc/                  # Documentação geral do projeto
  doxdoc/               # Documentação gerada automaticamente (Doxygen)
    html/               # Ficheiros HTML
      search/           # Dados de pesquisa da documentação
    latex/              # Ficheiros LaTeX para PDF da documentação
  src/                  # Código-fonte e artefatos de build
    .vscode/            # Configurações específicas da IDE para src/
```

Conteúdo da Pasta `src/`

```
src/
  example.txt           # Ficheiro de exemplo de entrada
  file.txt              # Ficheiro principal de entrada
  func.c                # Implementação das funções principais
  func.h                # Cabeçalho de func.c
  func.o                # Objeto compilado de func.c
  interface.c           # Funções de interface com o utilizador
  interface.h           # Cabeçalho da interface
  interface.o           # Objeto compilado de interface.c
  log.txt               # Log da execução atual
  log_old.txt           # Log anterior (backup)
  main.c                # Função principal
  main.exe              # Executável gerado
  makefile              # Script de compilação
  .vscode/
    settings.json       # Configurações do VS Code para debug/build
```

Observações

- A pasta `src/` centraliza todo o código-fonte, ficheiros de entrada e saída, bem como os objetos e o executável final.
- A documentação gerada via Doxygen pode ser encontrada em `doxdoc/html` para navegação via navegador e `doxdoc/latex` para geração de PDF.
- A pasta `.vscode/` contém configurações úteis para ambiente de desenvolvimento, facilitando a compilação e execução no Visual Studio Code.

8 Desafios Encontrados

Durante o desenvolvimento, surgiram dificuldades como:

8.1 Gerenciamento de Memória

C foi escolhido pela eficiência, mas exige controle manual de alocação e liberação, o que levou à criação de funções específicas de `FreeList` e `FreeGraph`.

8.2 Evitar Ciclos Inesperados

Durante a aplicação de BFS/DFS, era necessário garantir que cada vértice fosse visitado apenas uma vez, sendo implementado o campo `seen` nos vértices.

9 Trabalhos Futuros

Este projeto pode ser expandido de diversas formas:

- Implementação de algoritmos de caminho mínimo como Dijkstra.
- Suporte a diferentes potências de antena e zonas de interferência.
- Interface gráfica para visualização dinâmica da matriz e do grafo.
- Simulações em tempo real com dados gerados automaticamente.

10 Conclusão Final

Este projeto permitiu consolidar os conhecimentos em estruturas dinâmicas, algoritmos de grafos e modularização de código em C. A evolução natural da lista ligada para a representação em grafo demonstrou como a escolha de estrutura de dados impacta diretamente na eficiência e possibilidades do sistema.

As funcionalidades desenvolvidas cobrem desde o armazenamento, análise e visualização dos dados até à sua manipulação e busca inteligente com algoritmos clássicos. O projeto reforça práticas de engenharia de software como documentação, modularidade, tratamento de erros e interação com ficheiros e utilizador.

11 Bibliografia

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms.
- ISO/IEC 9899:2011, *Programming languages – C*.
- Doxygen Documentation. `https://www.doxygen.nl/index.html`.
- Sedgewick, R. & Wayne, K. (2011). Algorithms. 4th Edition.