Fase 2 - Grafos

Adriano Veloso - 14815
Estruturas de Dados Avançadas

Conteúdo

Resumo

Esta segunda fase do projeto foca-se na transição do modelo baseado em listas ligadas simples para uma estrutura mais complexa de grafos. A informação sobre as antenas, obtida de um ficheiro de entrada, passa a ser organizada como um grafo onde as conexões representam a igualdade de frequência. São implementadas funcionalidades de procura em profundidade e largura (*DFS* e *BFS*), caminhos possíveis entre antenas, bem como a deteção de interseções entre pares de antenas com diferentes frequências.

1 Introdução

1.1 Contexto e Objetivos

Esta segunda fase do projeto propõe uma abordagem mais estruturada ao problema da Fase 1, através da construção de um grafo. Este grafo representa cada antena como um vértice e estabelece ligações (arestas) entre antenas com a mesma frequência.

Além da representação do grafo, esta fase também introduz a análise geométrica entre pares de antenas de diferentes frequências, identificando pontos de interseção que ocorrem entre segmentos definidos por pares de antenas.

1.2 Metodologia

Nesta fase do projeto, foi inicialmente feita uma análise aprofundada ao enunciado com o intuito de identificar claramente as funcionalidades exigidas, nomeadamente a representação por grafos, as procuras em profundidade e largura, a identificação de caminhos entre antenas da mesma frequência, e o cálculo geométrico de interseções entre pares formados por antenas de diferentes frequências. Após esta etapa, definiu-se a estrutura de dados mais apropriada para o contexto, optando-se pela representação do grafo através de listas de adjacência, devido à sua eficácia para grafos grandes e à simplicidade no armazenamento.

A implementação foi feita de forma incremental, começando pela criação básica das estruturas principais, avançando depois para as funções auxiliares necessárias à gestão do grafo, como inserção de vértices e arestas. Posteriormente foram desenvolvidas as funções mais complexas relacionadas às procuras. Durante o desenvolvimento, cada funcionalidade foi sujeita a testes, o que permitiu uma validação contínua e a deteção de erros.

Ao longo do processo de codificação, o código foi sendo documentado detalhadamente através de comentários, o que facilita a compreensão da lógica interna e promove uma manutenção mais simples e eficaz em fases posteriores do projeto.

1.3 Repositório Git

O código fonte da solução está presente no seguinte repositório:
https://github.com/Adrianoicv/Fase2_14815/

2 Enquadramento

2.1 Grafos

Um grafo é uma estrutura composta por vértices (neste projeto *nodes*) e arestas (neste projeto ligações). É usado para representar relações entre entidades, sendo muito utilizado em problemas de redes, caminhos e ligações. Na figura 1 está representado um grafo com 6 vértices e 7 arestas.

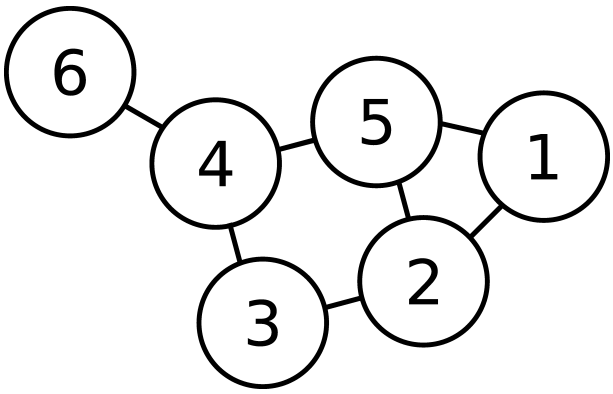

Figura 1: Exemplo de um grafo

2.2 Lista de Adjacência

Para armazenar as conexões entre antenas, a estrutura utilizada foi a lista de adjacência. Cada vértice possui uma lista ligada de vértices adjacentes, o que torna o armazenamento eficiente para grafos de grandes dimensões. Na tabela abaixo está representada a lista de adjacência de cada vértice do grafo da figura 1.

Vértice	**Lista de Adjacência**
1	$\rightarrow 2 \rightarrow 5$
2	$\rightarrow 1 \rightarrow 3 \rightarrow 5$
3	$\rightarrow 2 \rightarrow 4$
4	$\rightarrow 3 \rightarrow 5 \rightarrow 6$
5	$\rightarrow 1 \rightarrow 2 \rightarrow 4$
6	$\rightarrow 4$

Tabela 1: Listas de adjacência do grafo representado na figura 1

2.3 Procura em Profundidade (DFS) e Procura em Largura (BFS)

Foram implementadas duas formas clássicas de procura em grafos: a procura em profundidade (*Depth-First Search*, DFS) e a procura em largura (*Breadth-First Search*, BFS).

A procura em profundidade (DFS) é uma técnica que explora tão profundamente quanto possível cada ramo antes de retroceder (*backtracking*). Esta estratégia utiliza recursividade ou um *stack* explícito para manter o controlo dos vértices a visitar. É particularmente útil para encontrar caminhos, verificar conectividade e identificar componentes conexos em grafos [2].

Por outro lado, a procura em largura (BFS) explora todos os vértices adjacentes ao vértice inicial antes de avançar para o nível seguinte. Para isso, recorre a uma fila (estrutura *FIFO - First-In First-Out*), de forma a garantir que os vértices sejam visitados por níveis, do mais próximo ao mais distante. Esta técnica é especialmente eficaz para encontrar o caminho mais curto num grafo, assim como verificar a conectividade geral entre vértices [1].

Ambas as técnicas são fundamentais no contexto deste projeto, pois permitem alcançar todas as antenas direta ou indiretamente conectadas a partir de uma antena inicial, ao fornecer métodos eficientes e organizados para percorrer o grafo.

2.4 Interseções Entre Pares

Para identificar a interseção entre pares de antenas de frequências diferentes, foi desenvolvido um algoritmo que funciona seguindo estes passos:

- São selecionados pares de antenas da mesma frequência para formar segmentos;
- Cada par da frequência A é comparado com todos os pares da frequência B;
- Se os pares se cruzarem num ponto inteiro e dentro dos limites do segmento, considera-se uma interseção.

A fórmula de interseção entre dois pares é baseada no sistema linear das retas. A figura 2 mostra como foi interpretado o problema da interseção.

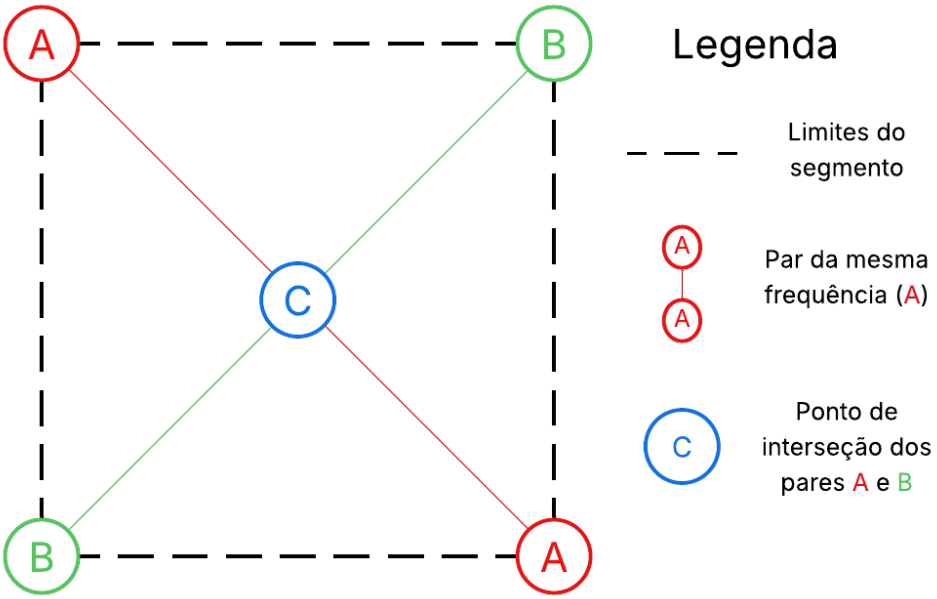

Figura 2: Exemplo de uma interseção entre pares

3 Trabalho Desenvolvido

Nesta fase, o projeto evoluiu de uma simples lista ligada para uma representação mais complexa e versátil: um grafo com listas de adjacência. Todas as antenas são transformadas em vértices e as suas ligações (caso tenham a mesma frequência) representam as arestas do grafo. A seguir descreve-se detalhadamente o trabalho implementado, incluindo estruturas de dados e funções.

3.1 Estrutura de Dados

As estruturas de dados utilizadas são definidas em ***graph.h***:

```
typedef struct Antenna {
    int x, y;
    char freq;
} Antenna;
```

Uma ***Antenna*** guarda uma posição e frequência.

```
typedef struct Node {
    Antenna data;
    struct Node* next;
} Node;
```

Um ***Node*** representa uma ligação a outra antena (aresta).

```
typedef struct Vertex {
    Antenna antenna;
    Node* adjList;
} Vertex;
```

Um ***Vertex*** contém uma antena e a sua lista de adjacências.

```
typedef struct Graph {
    Vertex* vertices;
    int size;
    int capacity;
} Graph;
```

Graph contém todos os vértices (antenas).

3.2 Funções para Representação de Grafos

createGraph

Cria um grafo novo com capacidade inicial definida.

Declaração:

```
Graph* createGraph();
```

Esta função aloca dinamicamente a estrutura genérica do grafo em memória, incluindo o vetor inicial de vértices. O vetor tem uma capacidade inicial definida por uma constante, o que permite futuras expansões dinâmicas com recurso à função *realloc*. Esta função é essencial, pois inicializa corretamente os parâmetros como *size* e *capacity*.

addVertex

Adiciona uma antena ao grafo como novo vértice.

Declaração:

```
void addVertex(Graph* g, Antenna a);
```

Esta função adiciona dinamicamente um vértice novo ao grafo. Antes da adição, é verificada a capacidade do vetor; se a capacidade máxima é atingida, duplica-se essa capacidade com *realloc*, para evitar erros de memória. Cada novo vértice é inicializado com uma antena específica e a respetiva lista de adjacência inicialmente vazia.

addEdge

Cria uma aresta entre dois vértices.

Declaração:

```
void addEdge(Graph* g, int i, int j);
```

Esta função adiciona uma nova aresta, ou seja, uma ligação entre dois vértices existentes no grafo *g->vertices[i]* para *g->vertices[j]*. A operação é feita ao criar um novo *node* que é inserido no início da lista ligada adjacente ao vértice de origem. Esta operação é rápida eeficiente em grafos dinâmicos.

loadGraphFromFile

Carrega um grafo a partir de um ficheiro *.txt*.

Declaração:

```
void loadGraphFromFile(Graph* g, const char* filename);
```

Esta função realiza a leitura e carregamento das antenas a partir de um ficheiro de texto. Utiliza *fgets* para ler o ficheiro linha a linha, tratando cada caractere individualmente. Ao detetar letras ou números (frequeência da antena), cria-se um objeto antena com as respetivas coordenadas e frequência, chamando depois ***addVertex***. Após a leitura completa do ficheiro, invoca-se a função ***linkSameFrequency*** para estabelecer as ligações apropriadas. Esta função é responsável por inicializar todo o estado inicial do grafo.

printGraph

Imprime todas as antenas e as suas ligações (grafo).

Declaração:

```
void printGraph(const Graph* g);
```

Esta função apresenta, na consola, o estado atual do grafo. Cada antena (vértice) é impressa, seguida da sua lista de adjacências. Mesmo sendo uma funcionalidade pedida no enunciado, foi muito útil como ferramenta de depuração, ao ajudar na verificação visual das ligações do grafo após alterações.

3.3 Funções de Procura

dfs

Faz uma procura em profundidade (*Depth-First Search*) a partir de uma antena.

Declaração:

```
void dfs(Graph* g, int startIndex);
```

Esta função realiza uma procura em profundidade a partir de uma antena inicial [2]. Internamente, usa uma função auxiliar recursiva (*dfsReach*), que explora ao máximo cada caminho antes de retroceder. Este método usa um vetor auxiliar (*visited*) para registar os vértices já visitados, o que garante que nenhum vértice seja visitado mais do que uma vez.

bfs

Faz uma procura em largura (*Breadth-First Search*) a partir de uma antena.

Declaração:

```
void bfs(Graph* g, int startIndex);
```

Esta função implementa uma procura em largura, que explora os vértices por níveis a partir de uma antena inicial [1]. Utiliza uma estrutura de dados auxiliar tipo fila (*int* q = malloc(sizeof(int) * g->size)*) para guardar a ordem correta de travessia dos vértices. O uso desta estrutura é essencial para garantir a ordem *FIFO* (*First-In, First-Out*), característica fundamental desta procura. A função também assegura que apenas antenas da mesma frequência são visitadas durante a procura, de forma a evitar percorrer ligações indesejadas.

findAllPaths

Encontra todos os caminhos possíveis entre duas antenas com a mesma frequência.

Declaração:

```
void findAllPaths(Graph* g, int startIndex, int endIndex);
```

Esta função encontra todos os caminhos possíveis entre duas antenas com a mesma frequência, com recurso a *backtracking*. Implementa um método *DFS* modificado que regista os caminhos encontrados num vetor auxiliar. Cada vez que o vértice de destino é visitado, o caminho é mostrado na consola. Depos, a função retrocede para explorar caminhos alternativos. A implementação usa uma estratégia de "marcação"e "desmarcação"(*backtracking*) de vértices visitados, o que garante uma travessia completa e sem redundâncias.

findIntersections

Procura interseções entre pares de duas frequências distintas.

Declaração:

```
void findIntersections(Graph* g, char freq1, char freq2);
```

Explicação:

- Recolhe todas as antenas da frequência 1 e 2;
- Para cada par de antenas de cada grupo, forma dois segmentos;
- Verifica se as retas se cruzam num ponto inteiro;
- Se sim, imprime o ponto como interseção;

Esta foi a função mais complexa implementada nesta fase, que determina as interseções geométricas entre pares de antenas de duas frequências distintas. A implementação segue as seguintes etapas:

1. São separados dois conjuntos de antenas com frequências fornecidas;
2. Calcula-se o ponto de interseção entre cada par formado pelas antenas destes conjuntos utilizando a fórmula matemática de interseção de retas:

$$px = \frac{(x_1y_2 - y_1x_2)(x_3 - x_4) - (x_1 - x_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

$$py = \frac{(x_1y_2 - y_1x_2)(y_3 - y_4) - (y_1 - y_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

3. Verifica-se se o ponto obtido é um ponto inteiro, recorrendo à função auxiliar *isInteger*.
4. Se o ponto for inteiro, confirma-se que este se encontra efetivamente dentro dos limites dos segmentos em análise com recurso à função auxiliar *isBetween*.
5. Se o ponto cumprir todas as condições, é impresso na consola como uma interseção válida.

3.4 Funções Auxiliares

isInteger

Esta função verifica se um valor é inteiro; serve como auxiliar à função *findIntersections*, para não devolver uma interseção com coordenadas inválidas (com casas decimais).

isBetween

Esta função verifica se um ponto está entre outros dois; serve como auxiliar à função *findIntersections*, para não devolver uma interseção de retas que não esteja "dentro"dos dois pares de antenas.

linkSameFrequency

Esta função cria arestas entre todos os pares de antenas que partilham a mesma frequência. Implementa-se através de um ciclo que compara cada par único de vértices. Sempre que a frequência coincide, cria-se uma ligação (com *addEdge*).

4 Conclusão

Esta segunda fase do projeto permitiu aprofundar significativamente os conceitos associados à representação dinâmica de dados através de grafos. Ao utilizar listas de adjacência, foi possível implementar uma solução eficiente para gerir grandes conjuntos de antenas, representar claramente as suas ligações e permitir operações de procura.

A implementação das buscas em profundidade (DFS) e largura (BFS) demonstrou-se particularmente eficaz para percorrer e explorar as ligações entre antenas com a mesma frequência. Adicionalmente, a função que encontra todos os caminhos possíveis entre dois vértices abriu caminho a uma análise detalhada de conectividade dentro do grafo.

Como possíveis melhorias para futuras etapas, destaca-se a otimização dos algoritmos geométricos e a implementação adicional de estratégias que permitam identificar zonas de conflito ou zonas nefastas entre as antenas de forma mais eficiente.

Resumindo, esta fase consolidou os conhecimentos práticos e teóricos sobre grafos e algoritmos de procura, e criou uma base sólida para a continuidade de trabalho em cenários mais complexos e realistas.

Referências

[1] kartik (GeeksforGeeks). *Breadth First Search or BFS for a Graph*. 2025. URL: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.

[2] kartik (GeeksforGeeks). *Depth First Search or DFS for a Graph*. 2025. URL: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/.