Relatório Fase 2 – Grafos
EDA

Adelino Daniel da Rocha Vilaça

Nº 16939 – Regime Pós-laboral

Professor

Luís Gonzaga Martins Ferreira

Github: https://github.com/danielvilaca/EDA

Ano letivo 2024/2025

Licenciatura em Engenharia de Sistemas Informáticos

Escola Superior de Tecnologia

Instituto Politécnico do Cávado e do Ave

RESUMO

O presente documento servirá de documentação desde a implementação do código em cumprimento do enunciado, como para demonstrar os resultados obtidos na execução do mesmo.

O foco do projeto é sedimentar os conhecimentos relativos à definição e manipulação de estruturas de dados dinâmicas na linguagem de programação C, documentando tudo em Doxygen.

ÍNDICE

ÍNDICE DE FIGURAS

Glossário

Array - Estrutura de dados linear que armazena elementos do mesmo tipo em posições contíguas de memória, permitindo acesso direto por índice (Cormen et al., 2009).

Aresta - Elemento de ligação entre dois vértices de um grafo. Pode ser direcionada (grafo dirigido) ou bidirecional (grafo não dirigido) (Sedgewick & Wayne, 2011).

Backtracking - Técnica de exploração recursiva que permite reverter decisões anteriores, típica em algoritmos de procura exaustiva, como listagem de caminhos (Cormen et al., 2009).

BFS (Breadth-First Search) - Algoritmo de procura em largura que visita todos os vértices a uma determinada "distância" (em número de arestas) antes de avançar para os seguintes (Sedgewick & Wayne, 2011).

DFS (Depth-First Search) - Algoritmo de procura em profundidade que visita recursivamente os vértices do grafo, explorando o mais a fundo possível antes de retroceder (Cormen et al., 2009).

Grafo - Estrutura de dados composta por vértices (nós) e arestas (ligações). Pode ser representado por matriz de adjacência ou listas de adjacência (GoodRich et al., 2011).

Vértice (ou nodo / nó) - Elemento de um grafo que representa uma entidade ou ponto de ligação. No projeto, corresponde a uma antena com coordenadas e frequência (Sedgewick & Wayne, 2011).

1. Introdução

A Fase 2 do projeto da unidade curricular de Estruturas de Dados Avançadas (EDA) propõe a aplicação de conceitos de grafos, utilizando a linguagem de programação C, para modelar e resolver um problema computacional com base na localização de antenas numa cidade.

Cada antena é representada por um carácter (Exemplo: “A”) numa matriz, correspondente à sua frequência de ressonância, e a sua posição na grelha determina a sua localização espacial.

O enunciado estabelece que as antenas devem ser transformadas em vértices de um grafo, sendo ligadas entre si por arestas apenas quando partilham a mesma frequência. A estrutura deverá permitir a procura em profundidade (DFS), procura em largura (BFS), a listagem de todos os caminhos possíveis entre duas antenas, e a identificação de pares de antenas com frequências distintas.

Estas funcionalidades têm como objetivo explorar as relações de interferência entre antenas, simulando um problema real com base na análise estrutural do grafo construído.

2. Implementação

2.1. Função inicializaGrafo

```c
/// @brief Inicializa o grafo (número de vértices) a 0
/// @param g
void inicializaGrafo(Grafo *g) {
    g->n_vertices = 0;
}
```

Figura 1 - Função inicializaGrafo

A função inicializaGrafo tem como principal objetivo preparar a estrutura de dados do grafo antes da sua utilização. Recebe como argumento um apontador para uma estrutura do tipo Grafo e define o campo n_vertices com o valor zero, o que representa que o grafo está inicialmente vazio, ou seja, ainda não contém qualquer antena registada. Esta etapa é essencial para garantir que a construção do grafo começa num estado limpo e controlado.

2.2. Função adicionaVertice

```c
/// @brief Adiciona uma nova antena (vértice) ao grafo.
/// @param g Apontador para o grafo
/// @param x Coordenada x da antena
/// @param y Coordenada y da antena
/// @param freq Frequência de ressonância da antena
/// @return ID atribuído ao novo vértice ou -1 se o grafo estiver cheio
int adicionaVertice(Grafo *g, int x, int y, char freq) {
    if (g->n_vertices >= MAX_VERTICES) return -1;
    int id = g->n_vertices;
    g->vertices[id].id = id;
    g->vertices[id].x = x;
    g->vertices[id].y = y;
    g->vertices[id].freq = freq;
    g->vertices[id].lista = NULL;
    g->n_vertices++;
    return id;
}
```

Figura 2 - Função adicionaVertice

A função verifica se o número atual de vértices atingiu o limite máximo permitido (MAX_VERTICES). Se sim, a função retorna -1, indicando que não é possível adicionar mais antenas. Cria uma variável (id) para usada como identificador único do novo vértice e para atribuir os valores das antenas e freq.

2.3. Função adicionaAresta

```c
/// @brief Adiciona uma aresta bidirecional entre dois vértices com a mesma frequência
/// @param g Apontador para o grafo
/// @param orig Vértice de origem
/// @param dest Vértice de destino
void adicionaAresta(Grafo *g, int orig, int dest) {
    if (orig < 0 || dest < 0 || orig >= g->n_vertices || dest >= g->n_vertices) return;
    if (g->vertices[orig].freq != g->vertices[dest].freq) return;

    Adj *nova = (Adj *)malloc(sizeof(Adj));
    nova->dest = dest;
    nova->prox = g->vertices[orig].lista;
    g->vertices[orig].lista = nova;

    // Aresta bidirecional
    Adj *nova2 = (Adj *)malloc(sizeof(Adj));
    nova2->dest = orig;
    nova2->prox = g->vertices[dest].lista;
    g->vertices[dest].lista = nova2;
}
```

Figura 3 - Função adicionaAresta

A função adicionaAresta é responsável por criar uma ligação (aresta) entre dois vértices do grafo, desde que ambos representem antenas com a mesma frequência. A ligação é feita de forma bidirecional, ou seja, cada vértice passa a referenciar o outro na sua respetiva lista de adjacências.

2.4. Função imprimeGrafo

```c
/// @brief Imprime todos os vértices do grafo e as suas ligações (arestas)
/// @param g Apontador
void imprimeGrafo(Grafo *g) {
    for (int i = 0; i < g->n_vertices; i++) {
        printf("Antena %d [%c] em (%d,%d) -> ",
            g->vertices[i].id, g->vertices[i].freq, g->vertices[i].x, g->vertices[i].y);
        Adj *aux = g->vertices[i].lista;
        while (aux != NULL) {
            printf("%d ", aux->dest);
            aux = aux->prox;
        }
        printf("\n");
```

Figura 4 - Função imprimeGrafo

A função percorre todos os vértices do grafo e apresenta, para cada um, a sua identificação, frequência e coordenadas, bem como os vértices a que está ligado através de arestas, percorrendo a lista de adjacência do vértice através do apontador (aux)

2.5. Função dfs_visit

```c
/// @brief Função auxiliar recursiva para DFS. Visita os vértices ligados ao atual
/// @param g Apontador
/// @param id Vértice atual
/// @param visitado Array de vértices já visitados
void dfs_visit(Grafo *g, int id, int *visitado) {
    visitado[id] = 1;
    printf("Visitado DFS: Antena %d [%c] em (%d,%d)\n",
           g->vertices[id].id,
           g->vertices[id].freq,
           g->vertices[id].x,
           g->vertices[id].y);

    Adj *adj = g->vertices[id].lista;
    while (adj != NULL) {
        if (!visitado[adj->dest]) {
            dfs_visit(g, adj->dest, visitado);
        }
        adj = adj->prox;
    }
}
```

Figura 5 - Função dfs_visit

O seu propósito é percorrer o grafo a partir de um vértice específico, explorando todos os vértices vizinhos de forma recursiva. Utiliza um array de

controlo (visitado[]) para garantir que cada vértice é visitado apenas uma vez, evitando ciclos ou repetições, através do apontador “adj”.

2.6. Função dfs

```c
/// @brief Inicia a procura em profundidade (DFS) a partir de uma antena
/// @param g Apontador
/// @param start_id Antena de partida
void dfs(Grafo *g, int start_id) {
    if (start_id < 0 || start_id >= g->n_vertices) {
        printf("ID inválido.\n");
        return;
    }

    int visitado[MAX_VERTICES] = {0};
    printf("DFS a partir da antena %d\n", start_id);
    dfs_visit(g, start_id, visitado);
}
```

Figura 6 - Função dfs

A função dfs (Depth-First Search) recebe o ID da antena de partida e utiliza um array (visitado[]) para controlar os vértices já percorridos. Esta função inicializa os dados necessários e chama recursivamente a função “dfs_visit”, que realiza a exploração dos vértices através das listas de adjacência.

2.7. Função bfs

```c
/// @brief Inicia a procura em largura (BFS) a partir de uma antena
/// @param g Apontador
/// @param start_id Antena de partida
void bfs(Grafo *g, int start_id) {
    if (start_id < 0 || start_id >= g->n_vertices) {
        printf("ID inválido.\n");
        return;
    }

    int visitado[MAX_VERTICES] = {0};
    int fila[MAX_VERTICES];
    int inicio = 0, fim = 0;

    fila[fim++] = start_id;
    visitado[start_id] = 1;

    printf("BFS a partir da antena %d\n", start_id);

    while (inicio < fim) {
        int atual = fila[inicio++];
        printf("Visitado BFS: Antena %d [%c] em (%d,%d)\n",
               g->vertices[atual].id,
               g->vertices[atual].freq,
               g->vertices[atual].x,
               g->vertices[atual].y);

        Adj *adj = g->vertices[atual].lista;
        while (adj != NULL) {
            if (!visitado[adj->dest]) {
                fila[fim++] = adj->dest;
                visitado[adj->dest] = 1;
            }
            adj = adj->prox;
        }
    }
}
```

Figura 7 - Função bfs

A função bfs (Breadth-First Search) inicia uma procura em largura a partir de um vértice específico, explorando o grafo nó a nó, ou seja, visitando primeiro todos os vizinhos diretos e só depois os vizinhos dos vizinhos. Verifica se o ID está dentro dos limites do grafo, criam-se 2 arrays: “visitado[]” para marcar os vértices já visitados e “fila[]” para manter a ordem de visita, “fila[fim++] =

start_id;” vértice de partida como visitado e incrementar o fim até ao fim da fila, impressão de cada antena e das suas informações (id, freq, coordenadas) e por último um percorrer final à lista de adjacências do vértice atual (se encontrar um vizinho não visitado, adiciona o mesmo no fim da fila “fila[fim++] = adj->dest;” e marca-o como visitado “visitado[adj->dest] = 1;”).

2.8. Função listarCaminhos

```c
/// @brief Lista todos os caminhos possíveis entre duas antenas com a mesma frequência
/// @param g Apontador
/// @param origem Antena de origem
/// @param destino Antena de destino
void listarCaminhos(Grafo *g, int origem, int destino) {
    if (origem < 0 || origem >= g->n_vertices || destino < 0 || destino >= g->n_vertices) {
        printf("ID inválido.\n");
        return;
    }

    if (g->vertices[origem].freq != g->vertices[destino].freq) {
        printf("Antenas com frequências diferentes não estão ligadas.\n");
        return;
    }

    int visitado[MAX_VERTICES] = {0};
    int path[MAX_VERTICES];
    printf("Caminhos entre %d e %d:\n", origem, destino);
    listarCaminhosAux(g, origem, destino, visitado, path, 0);
}
```

Figura 8 - Função listarCaminhos

A função listarCaminhos tem como objetivo encontrar e listar todos os caminhos possíveis entre duas antenas do grafo com a mesma freq calculando todas as sequências de vértices ligadas que ligam a antena de origem à de destino, sem passar duas vezes pelo mesmo vértice no mesmo caminho. É feita a validação dos Ids dentro dos limites do grafo, procura de caminhos só com antenas da mesma freq, instanciação dos 2 arrays auxiliares (visitado[] = vértices visitados; path[] = sequência de vértices do caminho atual) e chamada recursiva da função “listarCaminhosAux”

2.9. Função listarCaminhosAux

```c
void listarCaminhosAux(Grafo *g, int atual, int destino, int *visitado, int *path, int path_len) {
    visitado[atual] = 1;
    path[path_len++] = atual;

    if (atual == destino) {
        printf("Caminho: ");
        for (int i = 0; i < path_len; i++) {
            int id = path[i];
            printf("%d [%c](%d,%d)", id,
                   g->vertices[id].freq,
                   g->vertices[id].x,
                   g->vertices[id].y);
            if (i < path_len - 1) printf(" -> ");
        }
        printf("\n");
    } else {
        Adj *adj = g->vertices[atual].lista;
        while (adj != NULL) {
            if (!visitado[adj->dest]) {
                listarCaminhosAux(g, adj->dest, destino, visitado, path, path_len);
            }
            adj = adj->prox;
        }
    }

    visitado[atual] = 0; // backtrack
}
```

Figura 9 - Função listarCaminhosAux

A função listarCaminhosAux trata a lógica principal da procura de todos os caminhos possíveis entre dois vértices de um grafo, recorrendo a uma abordagem recursiva começando por marcar o vértice atual como visitado e adicioná-lo ao array “path[]” (visitado[atual] = 1; path[path_len++] = atual;).

Se o vértice atual for o destino, o conteúdo de “path[]” é impresso como um caminho válido. Caso contrário, percorre-se a lista de adjacências do vértice atual (Adj *adj = g->vertices[atual].lista;) e para cada vértice ainda não visitado, a função é chamada recursivamente. Após explorar todos os caminhos possíveis a partir de um vértice, desfaz-se a marcação de visita (visitado[atual] = 0;) para efetuar backtracking (permitindo que esse vértice possa ser reutilizado noutros caminhos)

3. Resultados

3.1. Matriz

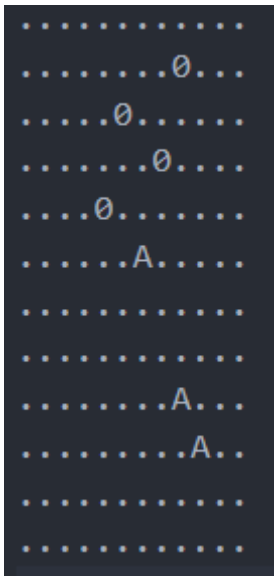

Figura 10 - Matriz escolhida para uso de teste

3.2. Teste Opção 1 – Imprimir grafo

```
--- Menu ---
1. Imprimir grafo
2. DFS a partir de antena
3. BFS a partir de antena
4. Listar caminhos entre duas antenas
0. Sair
Opcao: 1
Antena 0 [0] em (8,1) -> 3 2 1
Antena 1 [0] em (5,2) -> 3 2 0
Antena 2 [0] em (7,3) -> 3 1 0
Antena 3 [0] em (4,4) -> 2 1 0
Antena 4 [A] em (6,5) -> 6 5
Antena 5 [A] em (8,8) -> 6 4
Antena 6 [A] em (9,9) -> 5 4
```

Figura 11 - Primeira opção do Menu correspondente à impressão do grafo

3.3. Teste Opção 2 – DFS a partir de antena

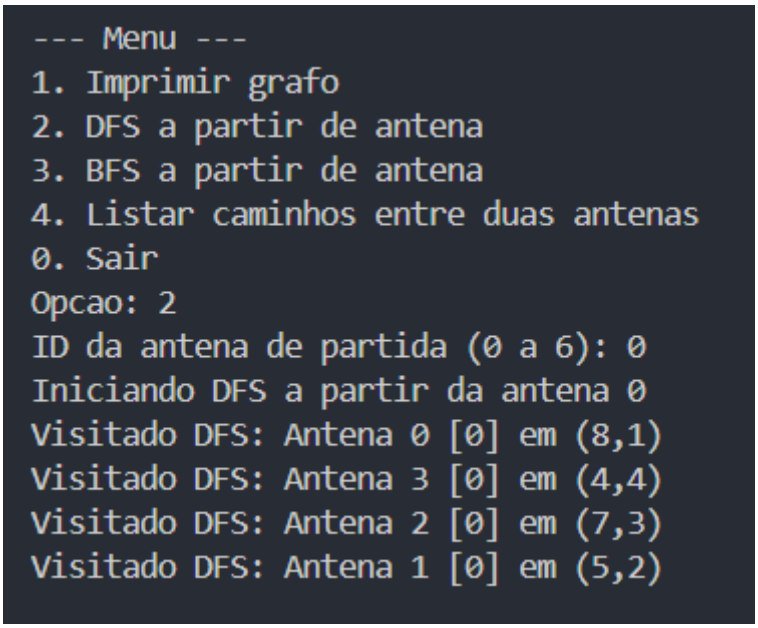

Figura 12 - Segunda opção do Menu correspondente à escolha de DFS a partir da antena com ID = 0 e frequência = 0

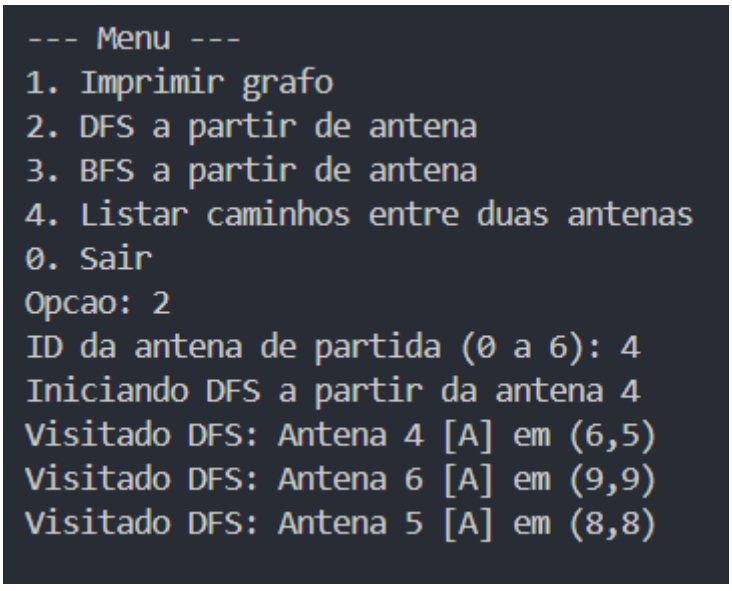

Figura 13 - Segunda opção do Menu correspondente à escolha de DFS a partir da antena com ID = 4 e freq = A

3.4. Teste Opção 3 – BFS a partir de antena

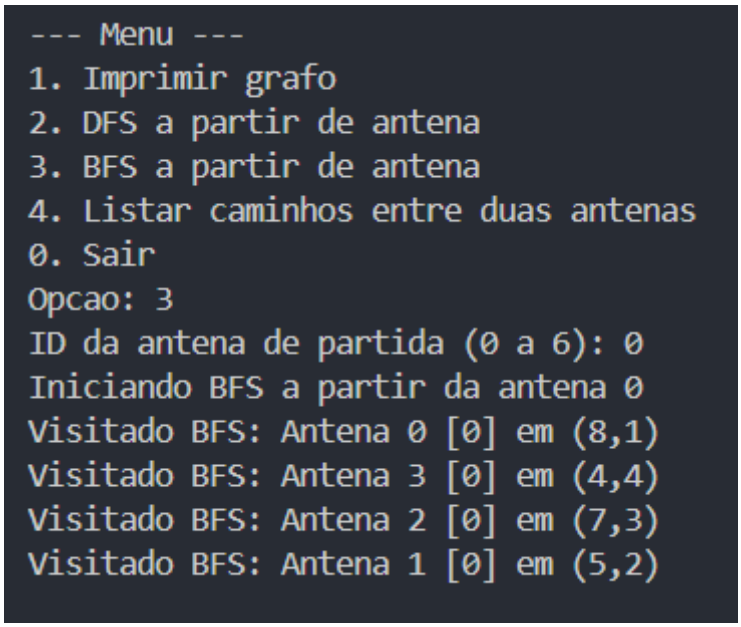

Figura 14 - Terceira opção do Menu correspondente à escolha de BFS a partir da antena com ID = 0 e freq = 0

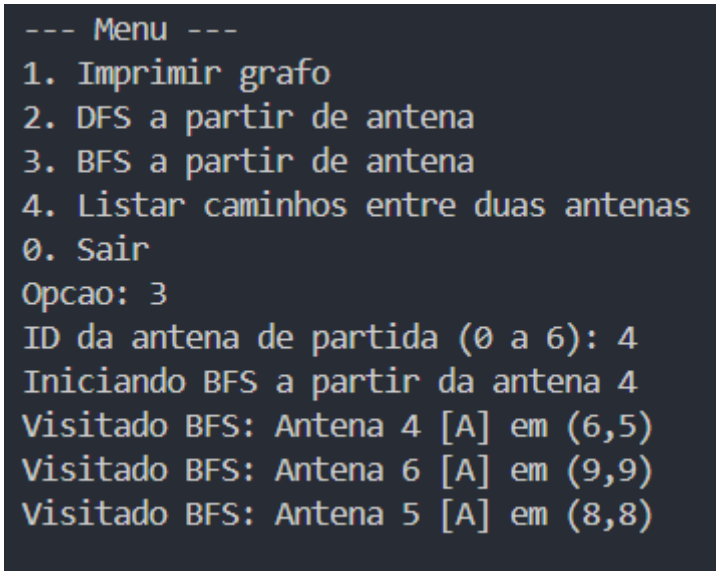

Figura 15 - Terceira opção do Menu correspondente à escolha de BFS a partir da antena com ID = 4 e freq = A

3.5. Teste Opção 4 – Listar caminhos entre duas antenas

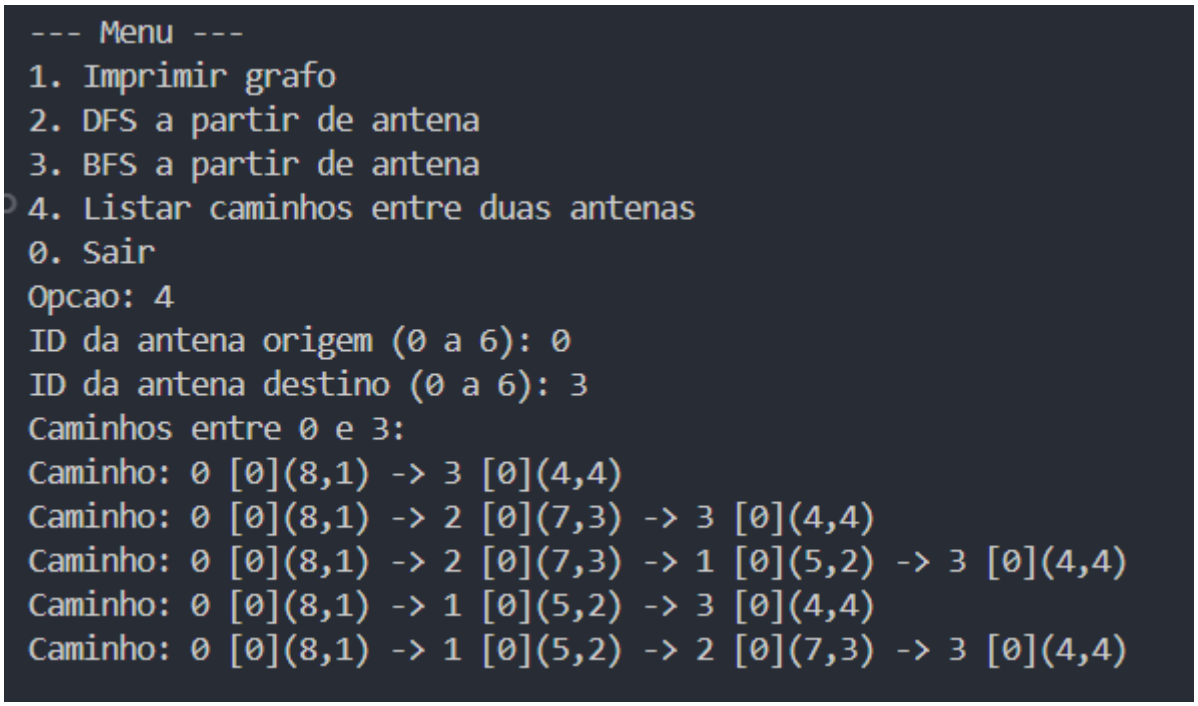

Figura 16 - Quarta opção do Menu correspondente à escolha de listar os caminhos entre duas antenas com ID origem = 0, ID destino = 3 e freq = 0

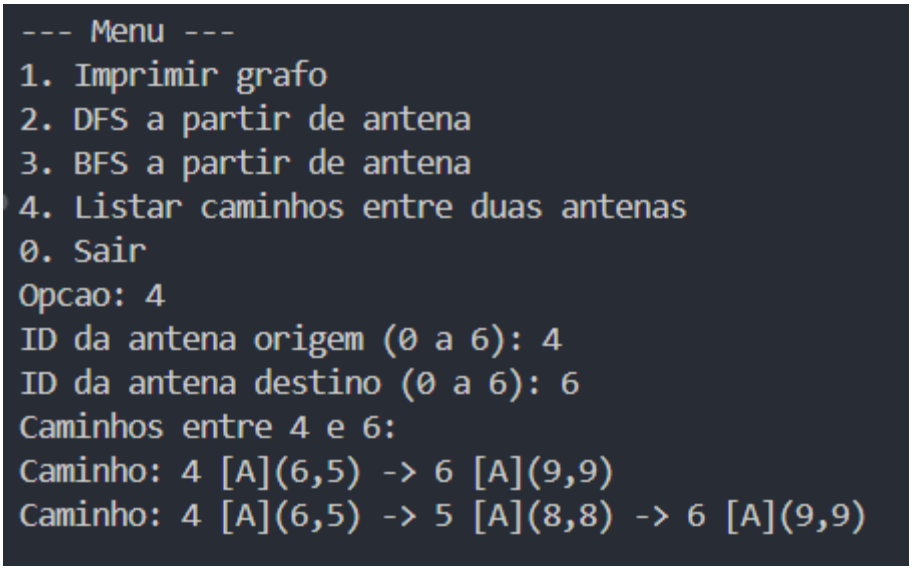

Figura 17 - Quarta opção do Menu correspondente à escolha de listar os caminhos entre duas antenas com ID origem = 4, ID destino = 6 e freq = A

4. Conclusão

Estas duas fases foram bastante enriquecedoras visto que serão não só uma mais valia para quem terá no próximo ano letivo a cadeira de Inteligência Artificial, visto que BFS e DFS serão alguns dos principais algoritmos lecionados, mas também para quem tem intenções de prosseguir com o Mestrado em Inteligência Artificial Aplicada, visto que reforça a importância de aplicação de conceitos teóricos em problemas comuns e complexos de Engenharia. É de louvar termos a oportunidade de não só podermos aprender os diversos conceitos mas também de aplicar os mesmos neste tipo de situações, como neste caso, o posicionamento de diversas antenas e as suas implicações/vantagens.

Apesar de necessitar de mais tempo para conseguir me aplicar mais ainda no trabalho, penso que no geral foi uma boa prestação e espero no futuro poder dar uso à matéria lecionada pelo Professor Luís Ferreira nesta cadeira de Estrutura de Dados Avançada, seja de formas profissionais ou académicas.

Muito Obrigado!

5. Bibliografia

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms, Third Edition*.

GoodRich, M. T., Tamassia, R., & Mount, D. M. (2011). *c_c-data-structures-and-algorithms-in-c*.

Sedgewick, R., & Wayne, K. (2011). *Algorithms FOURTH EDITION*.